
Electronic Thesis and Dissertation Repository

12-6-2013 12:00 AM

Redesign of Johar: a framework for developing accessible applications

Oladapo Oyebode
The University of Western Ontario

Supervisor
Dr. Jamie Andrews
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science
© Oladapo Oyebode 2013

Follow this and additional works at: <https://ir.lib.uwo.ca/etd>



Part of the [Graphics and Human Computer Interfaces Commons](#), and the [Software Engineering Commons](#)

Recommended Citation

Oyebode, Oladapo, "Redesign of Johar: a framework for developing accessible applications" (2013).
Electronic Thesis and Dissertation Repository. 1761.
<https://ir.lib.uwo.ca/etd/1761>

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlsadmin@uwo.ca.

REDESIGN OF JOHAR: A FRAMEWORK FOR DEVELOPING
ACCESSIBLE APPLICATIONS
(Thesis format: Monograph)

by

Oladapo Oyebode

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Masters of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Oladapo Oyebode 2013

Abstract

As the population of disabled people continues to grow, designing accessible applications is still a challenge, since most applications are incompatible with assistive technologies used by disabled people to interact with the computer. This accessibility issue is usually caused by the reluctance of software engineers or developers to include complete accessibility features in their applications, which in turn is often due to the extra cost and development effort required to dynamically adapt applications to a wide range of disabilities. Our aim to resolve accessibility issues led to the design and implementation of the “Johar” framework, which facilitates the development of applications accessible to both disabled and non-disabled users. In the Johar architectural model, the ability-based front-end user interfaces are called *interface interpreters*, while the application-specific logic or functionality implemented by application developers are called *applications* or *apps*. The seamless interaction between each interface interpreter and app is made possible by Johar.

In this thesis, we assure the quality of Johar by detecting and resolving many inconsistencies, omissions, irrelevancies, and other anomalies that can trigger unexpected or abnormal behaviour in Johar, and/or alter the smooth operation of interface interpreters and apps. Our approach to conducting the quality assurance involved reviewing the two components of Johar, `johar.gem` and `johar.idf`, by critically examining the functionality of classes in each component, including how classes interrelate and how functions are allocated or distributed among the classes. We also performed an exhaustive comparative review of four documents - IDF Format Specification document, XML Schema Document or XSD, the Interface Interpreter Specification document, and the `johar.idf` package - which are vital to the smooth running of all interface interpreters and apps. We also developed an automated testing tool in order to determine whether all errors or violations in an IDF (Interface Description File) are detected and reported.

As part of this thesis, we designed and implemented an interface interpreter, called Star that presents WIMP (Windows, Icons, Menus, and Pointers) graphical user interfaces to users, which is based on the “new version” of Johar. This new version evolved as a result of the redesign activities carried out on the Johar components and the various modifications effected during the quality assurance process. We also demonstrated the usage of Star on two apps to prove Johar’s ability to guarantee smooth interaction between interface interpreters and apps. Finally, in this thesis, we designed two other interface interpreters which will be implemented in the near future.

Keywords: Accessibility, Application Development, Software Framework

Co-Authorship Statement

The Johar framework was first designed, implemented, and presented at ASSETS in 2009 by my supervisor, Dr. Jamie Andrews, and his research assistant, Fatima Hussain. The quality assurance on Johar, discussed in Chapter 3, was carried out by both of us. I reviewed all the Johar-related documents (i.e. IDF Format Specification document, Interface Interpreter Specification document, XML Schema document, and the Johar packages) and submitted a report afterwards. The review report was then used by my supervisor in making appropriate corrections in the affected documents, which also include redesigning or restructuring the Johar packages. Furthermore, I implemented the automated testing tool in Java according to my supervisor's specification.

In Chapter 4, the specification documents (which describe both the GUI and behaviour) for the Star interface interpreter were written by my supervisor, which in turn were reviewed by me in order to report ambiguities and any conflict with the specification document for all interface interpreters (i.e. Interface Interpreter Specification document). I implemented the Star interface interpreter in Java, according to the requirements in those specification documents. Furthermore, the Appointment Calendar app that was used in our demonstration was first written by my supervisor for the 2009 version of Johar, and I had to rewrite some part of the app engine to conform with the new version of Johar. The Temperature Converter app, which was also used in our demonstration, was written by me.

Finally, in Chapter 5, I wrote the specification documents (which describe both the interface and behaviour) for Grupo and StarX interface interpreters.

Acknowledgements

I appreciate God for granting me knowledge, understanding, and strength in the course of my program at Western University.

I thank my supervisor, Dr. Jamie Andrews, for his wonderful support and guidance. Sir, you are truly a “Gem” and I am grateful for the opportunity to work with you on the “Johar” project.

I thank the Graduate Chair, Dr. Roberto Solis-Oba, and the Graduate Secretary, Mrs. Janice Wiersma, for making my student experience a memorable one. I also thank Mrs. Cheryl McGrath for her help in times of need.

Finally, I appreciate my parents, siblings, spouse, and friends for their inspiring words and support throughout my stay in Canada. Thank you all!

Contents

Abstract	ii
Co-Authorship Statement	iii
Acknowledgements	iv
List of Figures	x
List of Tables	xii
List of Appendices	xiii
1 Introduction	1
1.1 The Johar Framework	2
1.1.1 Interface Interpreter (IntI)	2
1.1.2 Application Engine	3
1.1.3 Interface Description File (IDF)	4
1.1.4 Software Architecture of Johar	7
1.2 Thesis Contribution	7
1.3 Thesis Outline	9
2 Background and Related Work	10
2.1 Assistive Technologies	10
2.2 Accessibility APIs	12
2.3 User Interface Architectural Models	13
2.4 User Interface Description Languages	15
2.5 Personalized User Interface Generators	17
2.6 Conclusion	19
3 Quality Assurance on Johar	20
3.1 Review and Redesign of Johar Components	20
3.1.1 The johar.gem package	20
The Review Process	22
The Redesign Process	23
3.1.2 The johar.idf package	23
The Review Process	24
The Redesign Process	24

3.2	Review of Johar-related Documents for Consistency	25
3.3	Test Infrastructure for IDFs	27
3.3.1	Generating Test Cases	28
	Procedure for Generating Valid Test Cases in TS 1	29
	Procedure for Generating Invalid Test Cases in TS 2	29
	Procedure for Generating Invalid Test Cases in TS 3	32
3.3.2	Running the Test Cases	35
3.3.3	Interpreting the Test Results	36
	The Error Log	36
	The Summary Report	39
4	The Star Interface Interpreter	41
4.1	Requirements Specification of Star GUI	41
4.1.1	The Main Panel	42
	The Menu Bar	42
	The Text Display Area	42
	The Table Area	43
	The Status Bar	43
4.1.2	The Command Dialog Box	43
	The Parameter Section of the Command Dialog Box	44
4.1.3	The Question Dialog Box	45
4.1.4	The Help Box	46
4.1.5	The Message Dialog Box	48
4.2	Design of Star	48
4.2.1	Components of Star	48
4.3	Implementation and Demonstration of Star	50
	Interacting with the Temperature Converter App	51
	Interacting with the Appointment Calendar App	58
4.4	Quality Assurance on Star	64
5	Other Interface Interpreters	66
5.1	Previous Interface Interpreters	66
5.2	The StarX Interface Interpreter	67
5.2.1	The StarX GUI	67
	Visual Cue for Focused Interface Widgets	68
	The Hotkeys Pop-up Table	69
5.2.2	Rationale for the Choice of Keyboard Shortcuts	69
5.3	The Grupo Interface Interpreter	70
5.3.1	Working with Tables	71
	Setting the Current Table	71
	Selecting Rows from the Current Table	71
	Deselecting Rows in the Current Table	73
	Displaying the Content of a Table	73
5.3.2	Accessing App Commands	73
	Specifying Parameters for App Commands	74

Executing App Commands	74
5.3.3 Accessing Help Contents	74
5.3.4 Testing Apps Using Grupo	75
6 Conclusion	77
6.1 Future Work	78
Bibliography	79
A Johar Interface Description File (IDF): Format Specification	84
A.1 Syntax	84
A.1.1 Syntax of Attribute Declarations	84
A.1.2 Example	85
A.1.3 Processing of Identifiers as Values	85
A.2 Allowed Attributes	86
A.2.1 Top-level Attributes	86
A.2.2 Sub-attributes of Command	88
A.2.3 Sub-attributes of Stage and Single-Stage Commands	91
A.2.4 Sub-attributes of Parameter	92
A.2.5 Sub-attributes of Question	99
A.2.6 Sub-attributes of CommandGroup	100
A.2.7 Sub-attributes of Table	101
A.2.8 Generated Attribute Values	102
A.3 Johar Booleans	102
A.4 Camel Case Translation	103
B Interface Interpreters (IntIs): Requirements Specification	104
B.1 Core Steps	104
B.2 Other Requirements	108
C Johar XML Schema Document	111
D Report On The Review Of Johar-Related Documents	118
E “Star” Interface Interpreter: Requirements Specification of the Star GUI	124
E.1 The Main Panel	124
E.2 The Command Dialog Box	125
E.3 Parameter Section of the Command Dialog Box	126
E.4 Repetition Section of the Parameter Section	127
E.5 Question Dialog Box	128
E.6 Help Box	129
E.6.1 Top-Level State	129
E.6.2 Command State	129
E.6.3 Parameter/Question State	130
F “Star” Interface Interpreter: Requirements Specification (Behaviour)	131

F.1	Top-Level Behaviour	131
F.2	Selecting a Command from a Menu	132
F.3	Question-and-Wrapup Procedure	133
	F.3.1 Question Dialog Cancel Button Action	133
	F.3.2 Question Dialog OK Button Action	134
F.4	Command Wrapup Procedure	134
F.5	Refreshing the Tables	135
F.6	The ShowTextHandler	135
F.7	The Command Dialog Box	136
	F.7.1 Creating the Command Dialog Box	136
	F.7.2 Initialize Stage Procedure	136
	F.7.3 Next Stage Procedure	137
	F.7.4 Previous Stage Procedure	137
	F.7.5 Wrap Up Stage Procedure	137
	F.7.6 Next Button Action	138
	F.7.7 Previous Button Action	139
	F.7.8 OK Button Action	139
	F.7.9 Cancel Button Action	139
F.8	The Parameter Section	140
	F.8.1 Add Another Button Action	140
	F.8.2 Move Up Button Action	140
	F.8.3 Move Down Button Action	140
	F.8.4 Delete Button Action	140
G	Some Source Code of “Star” Implementation	141
G.1	The <i>Star</i> class	141
G.2	The <i>CommandDialog</i> class	146
G.3	The <i>QuestionDialog</i> class	155
G.4	The <i>HelpBox</i> class	162
H	App Engine of the Temperature Converter App	172
I	Interface Description File (IDF) of the Appointment Calendar App	175
J	“StarX” Interface Interpreter: Requirements Specification of the StarX GUI	180
J.1	Main Panel	180
	J.1.1 The Menu Bar	181
	J.1.2 The Text Display Area	181
	J.1.3 The Table Area	181
J.2	The Command Dialog Box	182
	J.2.1 Parameter Section of the Command Dialog Box	182
	J.2.2 Repetition Section of the Parameter Section	183
J.3	Question Dialog Box	184
J.4	Help Box	184
	J.4.1 Top-Level State	185

J.4.2	Command State	185
J.4.3	Parameter/Question State	186
J.5	Keyboard Shortcuts for interacting with certain Widgets	187
J.5.1	Boolean Widget	187
J.5.2	Choice Widget and TableEntry Widget	187
J.5.3	Date Widget	188
J.5.4	File Widget	188
J.5.5	Number Widget	189
J.5.6	Text Widget	189
J.5.7	Time Widget	190
J.5.8	The Message Dialog Box	190
J.5.9	Hotkeys Pop-Up Table	191
K	“Grupo” Interface Interpreter: Requirements Specification	192
K.1	Commands in Grupo	192
K.1.1	The browse command	193
K.1.2	The help command	193
K.1.3	The table command	193
K.1.4	The select command	194
K.1.5	The deselect command	194
K.1.6	The command command	194
K.1.7	The param command	195
K.1.8	The ok command	195
K.2	Output Message Prefixes	195
K.3	The Input File	196
K.3.1	A Sample Input File	196
L	“Grupo” Interface Interpreter: Requirements Specification (Behaviour)	197
L.1	Top-Level Behaviour	197
L.2	Running Commands in an Input File	197
L.3	The Parse-and-Execute Command Procedure	198
L.4	Execute App Command Procedure	201
L.5	Execute Browse Command Procedure	202
L.6	Execute Help Command Procedure	202
L.7	Execute Exit App Procedure	203
L.8	The ShowTextHandler	204
	Curriculum Vitae	205

List of Figures

1.1	Architectural Model of the Johar Framework	3
1.2	User Interaction with Apps via Interface Interpreters (Scenario 1)	4
1.3	User Interaction with Apps via Interface Interpreters (Scenario 2)	5
1.4	The Intent-based Interaction Model	6
1.5	Software Architecture of the Johar Framework	8
2.1	Relationship between Assistive Technologies, Accessibility APIs, and Applications	12
2.2	The Seeheim Model	14
2.3	The Arch Model	15
3.1	The Class Diagram of the <code>johar.gem</code> Package	21
3.2	The Class Diagram of the <code>johar.idf</code> Package	25
3.3	A sample annotated IDF for generating test cases	28
3.4	Test Case 1 in TS 1	30
3.5	Test Case 2 in TS 1	31
3.6	Test Case 1 in TS 2	32
3.7	Test Case 2 in TS 2	33
3.8	Test Case 1 in TS 3	34
3.9	Test Case 2 in TS 3	35
3.10	Architecture of the Automated Testing Tool	37
3.11	Error Log for Test Case 1 in TS 1. [No error is detected]	38
3.12	Error Log for Test Case 1 in TS 2. [An error is detected]	38
3.13	Summary Report of the tests	39
4.1	The Main Panel of Star GUI	42
4.2	The Command Dialog Box of Star GUI	44
4.3	The Parameter Section of the Command Dialog Box	45
4.4	The Question Dialog Box of Star GUI	45
4.5	The Help Box of Star GUI [Top-Level State]	46
4.6	The Help Box of Star GUI [Command State]	47
4.7	The Help Box of Star GUI [Parameter/Question State]	47
4.8	The Message Dialog Box of Star GUI	48
4.9	The Class Diagram showing the key components of Star and the relationship among them	50
4.10	IDF for the Temperature Converter App	52
4.11	The XML equivalent of the Temperature Converter App's IDF	53

4.12	Main Panel of the Temperature Converter App	54
4.13	Main Panel of the Temperature Converter App [The <i>Convert</i> Menu]	55
4.14	Main Panel of the Temperature Converter App [The <i>Star</i> Menu]	55
4.15	The Command Dialog Box for the “Celsius to Fahrenheit” Command	56
4.16	Text Display Area shows the conversion result	56
4.17	Top-Level State of the Temperature Converter App’s Help Box	57
4.18	Command State of the Temperature Converter App’s Help Box	57
4.19	Parameter State of the Temperature Converter App’s Help Box	57
4.20	The Question Dialog Box confirming user’s intent to exit the App	58
4.21	Main Panel of the Appointment Calendar App	59
4.22	Main Panel of the Appointment Calendar App [The <i>Appointment</i> Menu]	59
4.23	Command Dialog Box for the “Add Appointment” Command	60
4.24	A notification and a new appointment are shown in the Text Display Area and Appointment table respectively	60
4.25	Weeks table indicates the existence of three appointments	60
4.26	Selecting the “Go To Date” Command	61
4.27	The Command Dialog Box for the “Go To Date” Command	61
4.28	Selecting an appointment from the Appointment table	62
4.29	Cancelling the selected appointment via the “Cancel Appointment” Command	62
4.30	Cancellation notification in the Text Display Area and deletion of appointment from the Appointment table	62
4.31	Top-Level State of the Appointment Calendar App’s Help Box	63
4.32	Selecting the Exit Command to terminate the Appointment Calendar App	63
4.33	A Message Dialog Box notifying the user of a successful termination of the App	64
5.1	A red border acting as a visual cue for a widget that currently has focus	68
5.2	The Hotkeys Pop-up Table	69
5.3	An input file containing Grupo commands for interacting with the Appointment Calendar App	72
5.4	Grupo accepts a test case, executes it against the App Engine via Johar, and displays output information on Standard Output for a Tester’s perusal	75
A.1	Example of IDF syntax.	85
A.2	Examples of camel-case translation.	103

List of Tables

2.1	An example of a blind user's capability captured in the Knowledge Base	18
3.1	Some results of the review of Johar-related documents for consistency	26
4.1	Outcome of reviewing Star Specification documents in conjunction with the Interface Interpreter Specification document	65
5.1	Prefix for each category of information displayed on the Standard Output	76
B.1	Bindings of Johar parameter types to Java types.	107
J.1	Keyboard shortcuts for the Menu Bar	181
J.2	Keyboard shortcuts for the Text Display Area	181
J.3	Keyboard shortcuts for the Table Area	182
J.4	Keyboard shortcut for the Cancel, Previous, Next and OK buttons	182
J.5	Keyboard shortcuts for the Parameter Section	183
J.6	Keyboard shortcuts for buttons in the Repetition Section of a Parameter Section	184
J.7	Keyboard shortcuts for the Question Dialog Box	184
J.8	Keyboard shortcuts for the Help Box buttons	185
J.9	Keyboard shortcuts for the Top-Level State's Commands table	185
J.10	Keyboard shortcuts for the Command State's Text Area	186
J.11	Keyboard shortcuts for the Command State's Parameters table	186
J.12	Keyboard shortcuts for the Command State's Questions table	186
J.13	Keyboard shortcuts for the Parameter/Question State's Text Area	187
J.14	Keyboard shortcuts for the Boolean Widget	187
J.15	Keyboard shortcuts for the Choice/TableEntry Widget	188
J.16	Keyboard shortcuts for the Date Widget	188
J.17	Keyboard shortcuts for the File Widget	189
J.18	Keyboard shortcuts for the Number Widget	189
J.19	Keyboard shortcuts for the Text Widget	190
J.20	Keyboard shortcuts for the Time Widget	190
J.21	Keyboard shortcut for the Message Dialog Box	191
K.1	Grupo Commands	193
K.2	Prefixes of output messages	196

List of Appendices

Appendix A Johar Interface Description File (IDF): Format Specification	84
Appendix B Interface Interpreters (IntIs): Requirements Specification	104
Appendix C Johar XML Schema Document	111
Appendix D Report On The Review Of Johar-Related Documents	118
Appendix E “Star” Interface Interpreter: Requirements Specification of the Star GUI . .	124
Appendix F “Star” Interface Interpreter: Requirements Specification (Behaviour)	131
Appendix G Some Source Code of “Star” Implementation	141
Appendix H App Engine of the Temperature Converter App	172
Appendix I Interface Description File (IDF) of the Appointment Calendar App	175
Appendix J “StarX” Interface Interpreter: Requirements Specification of the StarX GUI .	180
Appendix K “Grupo” Interface Interpreter: Requirements Specification	192
Appendix L “Grupo” Interface Interpreter: Requirements Specification (Behaviour) . . .	197

Chapter 1

Introduction

Accessibility in computing is concerned with the removal of barriers that exclude some group of people from using the computer. Designing software applications for accessibility is concerned with making programs and functionality available to a variety of users, whether disabled or non-disabled. Unfortunately, almost all software applications contain some barriers to people with disabilities [1]. Although there are assistive technologies (such as screen readers) to bridge this digital divide, many applications are incompatible with these technologies since they were designed for non-disabled users. This situation may continue to worsen, considering the increasing population growth of disabled people. For example, approximately 56.7 million Americans (18.7% of the civilian noninstitutional population) have some form of disability as at 2010, as opposed to the 54.4 million population in 2005 [2]. Moreover, the World Health Organization (WHO) estimated that 285 million people are visually-impaired globally in 2010 (out of which 39 million are blind) [3], and the global population of blind people is expected to increase to 78 million by 2020 [4].

Furthermore, in order to make an application accessible, the interface through which a user interacts with the application to undertake his or her tasks must be considered. This interface (popularly known as *user interface*) can be graphical, vocal or textual, but the decision about how it should be presented must be based on the ability of the user. For example, a blind user cannot read commands and information presented in graphical or visual format (e.g. menus, tables, buttons, icons, etc.); a deaf user cannot comprehend any information presented vocally (i.e. speech output); a low vision user cannot read any text presented in small font or notice a small control; a motor-impaired user will have difficulty controlling a pointer and clicking with the mouse; etc. Hence, an application becomes accessible if its user interface fits users' abilities and needs.

Some research efforts have already been directed towards making applications accessible to disabled and non-disabled users. These efforts led to the development of assistive

technologies (such as screen readers for blind users [5][6][7], screen magnifiers for low-vision users [8], pointing tools for motor-impaired users [9], speech-enabled systems for blind and low-vision users [10]), accessibility APIs (used by assistive technologies and applications to access interface components) [11][12][13], and personalized user interface generators which automatically generate user interfaces based on users' ability or preferences [14][15][16]. Unfortunately, assistive technologies are not compatible with all applications, and the generated user interfaces may not fit the ability and needs of every user.

Our determination to complement the efforts of other researchers in resolving accessibility issues led to the development of our *Johar framework* [17], which facilitates the development of applications accessible to both disabled and non-disabled people.

1.1 The Johar Framework

“Johar” is a framework that facilitates the development of applications that can be used by both disabled and non-disabled users [17]. The Johar framework promotes the separable interface theory [18] by providing a separation between the front-end user interface and the application-specific logic. Thus, experienced user interface designers can focus on developing user interfaces that fit the ability and needs of users, while application developers can focus on implementing the underlying functionality or application-specific logic of applications.

In our work, the ability-based front-end interfaces are called *Interface Interpreters*, while their designers are known as *Interface Interpreter developers*. The application-specific logic or functionality implemented by application developers are called *applications* or *apps*. The Johar framework binds each Johar interface interpreter to each Johar app [17]. Thus, improvements in interface interpreters automatically improve access to all applications, and improved applications are accessible to all users [17]. This is in contrast to other efforts in accessible software development, in which improvements often help one group of users to access one kind of application.

As shown in Figure 1.1, a user interacts with an app using an *Interface Interpreter* that is suitable. This interface interpreter first reads the *Interface Description File (IDF)* written by the app developer, and then communicates with the *application engine* through Johar.

1.1.1 Interface Interpreter (IntI)

The interface interpreter is the mediator between the user and a Johar app. An interface interpreter usually supports specific user groups (e.g. blind users, motor-impaired users, deaf

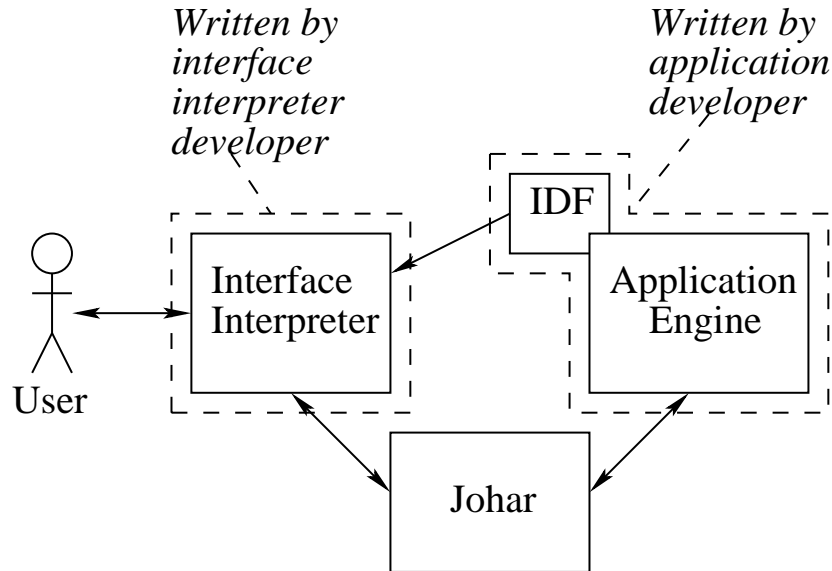


Figure 1.1: Architectural Model of the Johar Framework [17]

users, etc.). Thus, in order to access the desired app, each user must select an interface interpreter that suits his or her ability and needs (see Figure 1.2 and Figure 1.3). For example, a non-disabled user can select a graphical or WIMP¹ interface interpreter; a blind user can select a speech-driven/sound-based interface interpreter; a motor-impaired user can select an interface interpreter that support certain interaction techniques depending on their dexterity (e.g. scanning-based interaction, eye gaze, head gestures, voice-driven interaction, mouse-driven interaction, keyboard-driven interaction etc.); and so on.

Interface interpreters interact with various input/output devices (e.g. keyboard, camera, speaker, microphone, GPS device etc.) to get input data from users, and to present output data in a form that the user can perceive via their working senses (e.g. hearing, sight, etc.).

1.1.2 Application Engine

The application engine or *app engine* contains the application-specific logic, which is made up of various components and objects that perform series of computations on the input in order to transform it to an output. The app engine is the core of any Johar app. The app engine receives input data from the interface interpreter, and then performs computations on the data, thereby producing an output which is sent to the interface interpreter. The output can be in form of table data or text. During computation, the app engine may access external resources, such as

¹WIMP stands for Windows, Icons, Menus, and Pointers. The most prevalent of the graphical user interfaces (GUIs) is the WIMP interface [19].

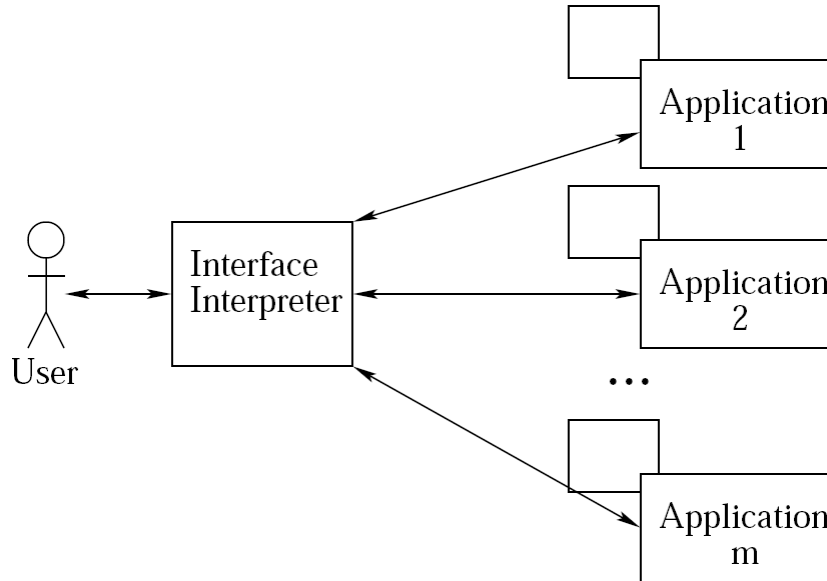


Figure 1.2: Each user can access all Johar apps via his or her selected Interface Interpreter [17]

databases, files, third-party APIs, etc.

1.1.3 Interface Description File (IDF)

The Johar IDF contains a high level description or definition of a user interface. Our approach to defining interfaces is based on the *Intent-based Interaction Model*, presented in Figure 1.4. In this model, every interaction session between a user and a Johar app can be described as a cycle which alternates between a phase in which the user specifies what they intend the application to do (Intent Specification), and a phase in which the app does it (Application Computation) [17]. This interaction cycle begins when an app is launched.

Whenever a user launches an app, the app engine may perform some initial computation and presents initial data to the user. Afterwards, the user is required to specify his or her intent depending on the task at hand. While specifying his or her intent, a user may browse the initial data (e.g. table data), select data (e.g. select single or multiple rows in a table), select a command (e.g. selecting a menu in a GUI or typing a command in a text-based/command-line interface), select parameters (which are inputs to a command), refine his or her intent by providing additional parameters, and confirm his or her intent (e.g. by answering an *are you sure* question). Once the intent specification is fully completed, the app performs some computation (via the app engine). Thus, activities 1 and 2 are application-specific and performed by the application engine, while activities 3, 4, and 5 take place at the interface-level through the use of an interface interpreter (by the user).

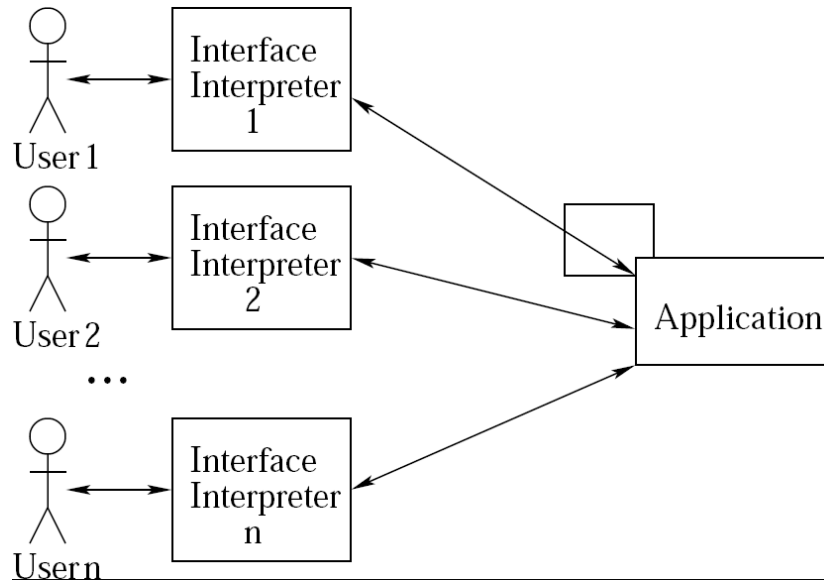


Figure 1.3: Each app can be accessed by all users via their selected Interface Interpreters [17]

The IDF Format

The objectives of any IDF are: (1) *to describe the nature of an application and the functionality available to users*; and (2) *to describe the nature of the data that will be exchanged between the user and the application*. In a bid to accomplish these objectives, the IDF is structured as a tree of attribute instances, where each attribute instance can have a *name* and *value* [17]. The attribute value can contain more attributes. Each attribute-value pair contributes towards achieving the objectives above. A brief description of some important attributes of an IDF is given below. Appendix A of this thesis shows all the attributes and their respective semantics and syntax.

- **ApplicationEngine:** This attribute specifies the class that contains the application-specific logic of the app.
- **Command:** This attribute represents the basic top-level unit of user-application interaction. A **Command** represents a service or functionality provided by the app. It also corresponds to a broad statement of intent the user can make. For example, in a graphical Word Processor, the “Copy” menu item is a command; while in a Unix terminal, “cp” is the equivalent command for “Copy”. The sub-attributes of **Command** specify the app engine method that determines if a command is active, the help messages, priority level or prominence of the command, questions to ask users, app

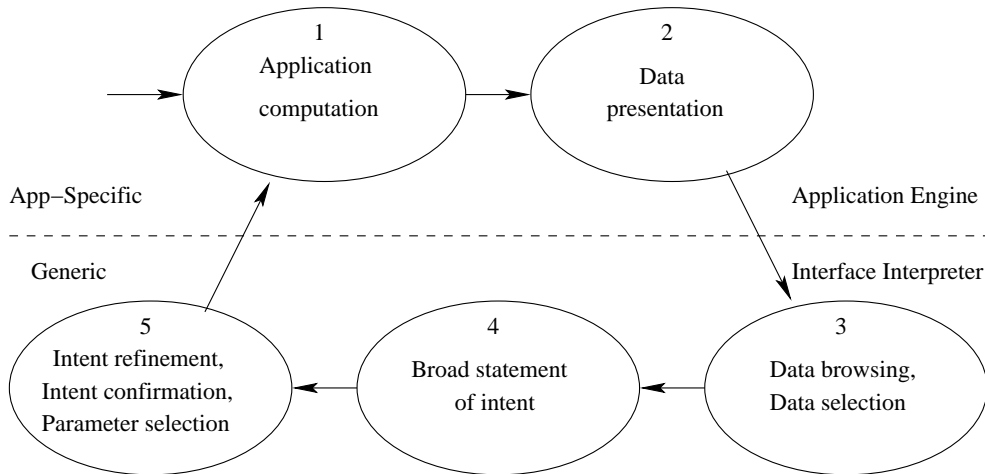


Figure 1.4: The Intent-based Interaction Model [17]

engine method that performs the actual computation, label of the command, app engine method that determines whether the app should quit after performing the computation, and stages of the command (for wizard-style interaction).

- **CommandGroup:** This attribute represents a group of related commands. For example, in a graphical text editor, the File menu is a command group. Its sub-attributes specify the label of the command group, and each command in the group.
- **Table:** This attribute represents a list of similar data entities (i.e. table) presented to the user by the app engine. Its sub-attributes specify the default table header, default column names, and a value stating whether the table is browsable or not. A browsable table is visible to the user, and rows can be selected from it.
- **Parameter:** This represents a piece of data which comes from the user and is relevant to the Command, such as an integer, floating-point number or string. It has many sub-attributes, but the important sub-attribute is the Type, which specifies the parameter type. The valid types are `int`, `float`, `boolean`, `text`, `file`, `tableEntry`, `choice`, `date`, and `timeOfDay`. A Parameter of type `tableEntry` must contain the `SourceTable` attribute.
- **Question:** This represents a question that should be asked of the user. The sub-attributes of Question are similar to most of the sub-attributes of Parameter, except that Question has a sub-attribute that specifies the app engine method which determines whether the question should be asked or not. Questions can be used to confirm a user's intent, before the required computation is performed.

The IDF is written in a domain-specific language which is more concise than XML. However, it is internally converted to XML during processing; thus, no technical skill is required (from an app developer) to write a complete IDF for an app.

1.1.4 Software Architecture of Johar

The current Johar framework is implemented in Java, and it consists of two main packages (`johar.gem` and `johar.idf`). The `johar.gem` package contains several classes that work together to ensure smooth dialogue between the interface interpreter and the application engine (such as facilitating exchange of data between them, and also invoking appropriate methods in the application engine to handle users' requests). The `johar.idf` provides APIs that are called by the interface interpreter to access the content (i.e. attributes and values) of an IDF.

Figure 1.5 shows the software architecture of the Johar framework. Although we hope to port Johar to other platforms in the future, we expect that the architecture will remain very similar.

1.2 Thesis Contribution

The overall goal of this thesis is to assure the quality of Johar, and to do so in a way that allows us to reflect and report on our quantitative and qualitative observations. To achieve this goal, we performed a comprehensive review of the entire Johar project so as to detect and correct inconsistencies, omissions, irrelevances, and other loopholes that can hinder successful deployment of Johar. We also implemented a complex interface interpreter and designed two additional interface interpreters. We can never say that any program is completely free of bugs, or that design documents are of the maximum quality. However, we believe that the result of our work is a set of documents and a working interface interpreter implementation that are, to the best of our knowledge, complete, internally consistent, and consistent with each other.

We started work with the review of Johar's components, which are the `johar.gem` and `johar.idf` packages. We discovered that the `johar.idf` package is poorly structured because only one class performs the overall function of the package. This kind of structure is bug-prone, encourages code duplication, and prevents code reuse. We addressed this issue by restructuring the `johar.idf` package into a suite of specialized and interrelated classes.

The next phase of our quality assurance process involves comparing four documents for consistency. The documents are: IDF Format Specification document², XML Schema

²defines the syntax and semantics of all the attributes an IDF can contain.

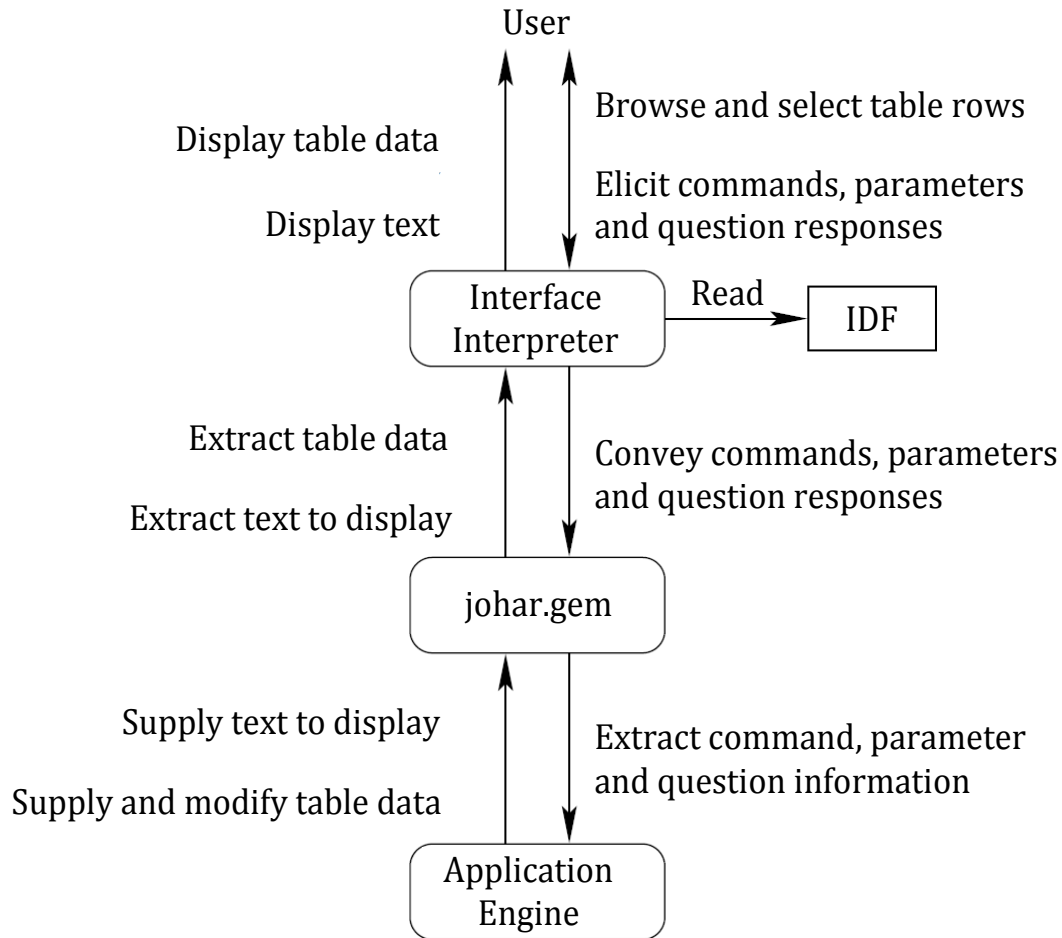


Figure 1.5: Software Architecture of the Johar Framework [17]

document³, the Interface Interpreter Specification document⁴, and the `johar.idf` package. The outcome of this review process revealed some discrepancies. We addressed these discrepancies by making appropriate changes to these documents and their dependencies. Furthermore, we decided to build a test infrastructure for IDFs, since they are critical to the smooth running of interface interpreters. The test infrastructure determines whether all IDF attributes are captured by the XML Schema and `johar.idf` package. It also verifies whether the XML Schema correctly validates an IDF, and whether all errors in an IDF are caught and reported.

Furthermore, since we have modified Johar in the course of our quality assurance activity, we needed to test the practicability of the framework by developing and running an interface interpreter and apps. Thus, we designed a new interface interpreter, called *Star*, which

³used for validating the XML-equivalent of an IDF.

⁴specifies the general behaviour of all interface interpreters.

provides a graphical user interface (GUI) through which users can access any Johar app. The design documents of *Star* describe both the look-and-feel and behaviour of *Star*. We then reviewed these design documents and the Interface Interpreter Specification document for consistency. After completing the review process and resolving inconsistencies and ambiguities, we implemented and tested *Star*. Also, we demonstrated the usage of *Star* with an app.

In the last phase of our work, we wrote design documents for two future interface interpreters - *StarX* and *Grupo*. The *StarX* interface interpreter extends *Star* by providing keyboard shortcuts through which users (especially users with limited hand use) navigate the entire graphical user interface. *Grupo* is a “batch-mode” interface interpreter that provides commands through which users carry out their tasks.

Finally, in the near future, we hope to release the first version of Johar as open source; and we expect researchers, experienced interface designers, and application developers around the world to explore Johar, and to build interface interpreters and apps on top of it. Thus, in not so distant future, we would have formed Johar interface interpreter and app *developers community*, as well as *users forum* that will serve as a platform on which Johar developers brainstorm, and on which app users share their views and experiences.

1.3 Thesis Outline

Chapter 2 contains the background and related work. Chapter 3 contains a detailed description of the quality assurance processes, and the resulting effects on the internal structure of Johar and its dependencies. Chapter 4 contains the specification, design and implementation details of our new interface interpreter for graphical interfaces, called *Star*. This chapter also contains a description of how we assure the quality of *Star*, and how we used *Star* to access (and interact with) a Johar app. Chapter 5 contains the specification of two other interface interpreters, called *Grupo* and *StarX*. The last chapter of the thesis, Chapter 6, contains the summary and conclusion.

Chapter 2

Background and Related Work

The background of this thesis is drawn from the answers to the following questions: What are the technologies currently used by disabled people to access applications? What techniques are adopted by these technologies to provide accessibility? Section 2.1 and Section 2.2 provide a summary of the research conducted to find answers to these questions.

Our research findings summarized in Sections 2.1 and 2.2 revealed the vital role a user interface plays in making an application accessible. This revelation led to the following questions: What is the general structure or model of a user interface? How can a user interface be designed to support users with diverse abilities? We conducted further research in order to answer these questions. Section 2.3 presents a summary of our findings for the first question, while Sections 2.4 and 2.5 are for the second question.

2.1 Assistive Technologies

People with disabilities - such as visual impairment, blindness, and motor impairment - can operate computers to perform their tasks via assistive technologies.

For computer users with low vision, the prominent assistive technology is screen enlargement software, such as ZoomText Magnifier [20] and MAGic [8], which increase the size of controls and texts on the screen so they can be clearly visible to users. Screen readers assist blind users in using applications on computers. Screen readers basically read computer screens and speak the text contents [21]. In other words, screen readers have the capability to analyze user interface components (menus, message/dialog boxes, text containers, etc.) and produce the speech output of their contents [22]. Most screen readers support Braille displays. Some of the prominent screen readers are Job Access With Speech (JAWS) [5], Non-Visual Desktop Access (NVDA) [6], Windows Eyes [7], ZoomText Reader [20], Narrator [23],

VoiceOver [24], Linux Screen Reader [25], and Orca [26]. These screen readers run on specific platforms; for example, the first five screen readers above run on Windows operating systems, VoiceOver runs on Apple-based operating systems (Mac OS X, iOS), while Linux Screen Reader and Orca run on GNOME-compliant operating systems (OpenSolaris, Ubuntu, etc.). JAWS, NVDA and VoiceOver are the commonly-used screen readers [27].

Assistive technologies for motor-impaired users also exist. Punyabukkana et al. [10] designed a sound-based input system that allow users to traverse, select, and activate commands in Windows GUI just by humming and making fricative sounds. The hum sound initiates traversal in a direction (e.g. moving from one menu to the other in the right direction), while the fricative sound stops the traversal and activates the selected command. The pitch of the hum sound is used to determine the direction of the traversal. These two sounds were also used in controlling the cursor to perform point-and-click tasks (e.g. clicking a button to launch an application or save a document) [28]. The hum sound controls the pointer, while the fricative sound clicks the control (e.g. button or icon) at the location of the pointer. Another assistive technology for the motor-impaired is ceCursor [9] which uses eye-gaze tracker in controlling a screen pointer on Windows. The user can move the cursor in one of four directions (up, down, left, right) by gazing in that direction, and clicking of interface controls is achieved whenever the user looks at the centre of the cursor for 1 second.

Biswas et al. presented new systems to assist computer users with severe motor impairments [29][30]. The first is the cluster scanning system which identifies potential targets (e.g. buttons, icons, menus, and other widgets) on the screen and then groups them into clusters based on their locations. The user selects the cluster containing the target using a push button switch, and then the system iteratively divides the selected cluster into smaller clusters, until only one cluster remains (which contains the target). The second system combines eye-gaze tracking and scanning techniques to achieve greater efficacy. The eye-gaze tracker moves the pointer closer to (or on) the target, and then the system switches to the 8-directional scanning mode (after a key press from the user) to focus and click on the target.

Although these assistive technologies have helped computer users with impairments to interact with applications on computers, most high-quality commercially-available screen readers and magnifiers are expensive, as well as eye-gaze trackers and switches (which are not widely available). Also, most screen readers do not provide access to all applications, since they have difficulty interpreting some widgets (e.g. custom controls) [31]. Finally, most generic accommodation techniques for users with motor-impairments have difficulty working with applications with complex GUI layout and small interface controls.

2.2 Accessibility APIs

Accessibility APIs (Application Programming Interfaces) mediate between assistive technologies and applications. As shown in Figure 2.1, a user interacts with an application via an assistive technology, which in turn relies on an accessibility API to perform its function. Accessibility APIs enable assistive technologies to successfully identify, access, and manipulate elements of an application’s user interface.

Microsoft developed two accessibility APIs that allow assistive technologies to access applications running on Windows operating systems. The first API is called Microsoft Active Accessibility (MSAA) [11], which is based on the Component Object Model (COM) technology. It provides API elements which contain methods for exposing information about user interface elements. Thus, assistive technologies (via the OLEACC library) can interact with and retrieve the content of standard or common interface controls for any Windows application. The `IAccessible` interface of MSAA allows applications to further expose information about interface elements that are not available to assistive technologies by default (e.g. custom controls and windowless elements). Assistive technologies are notified (via `WinEvents` or Windows Events) of changes in interface elements. They retrieve the contents of (and other information about) interface elements through the properties exposed to them (e.g. Name, Role, Location, Value, Description, Help, etc.), and can also navigate the tree of interface elements (i.e. object model) in order to understand the structure of the user interface. An example of a screen reader using MSAA is JAWS [5].

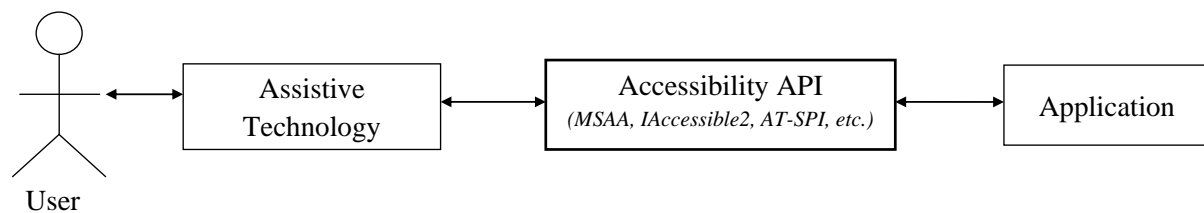


Figure 2.1: Relationship between Assistive Technologies, Accessibility APIs, and Applications

The latest API developed by Microsoft is UI Automation [12]. UI Automation offers support for new user interface elements, exposes a richer set of properties and control patterns for UI elements, offers flexibility in navigating the object model (via scoping and filtering), and allows application developers to define custom control patterns, properties, and events.

Another post-MSAA API is `IAccessible2` [13], which supports rich-text controls, Web 2.0 technologies, tables, and spreadsheets. This API is useful for rich document applications, such as Word Processors, Spreadsheet packages, Web browsers using AJAX and DHTML, and so on. `IAccessible2` supports applications running on Windows and Unix operating systems.

Other accessibility APIs are the ATK (Accessibility Toolkit) and AT-SPI (Assistive Technology Service Provider Interface) [32] which provide access to both GNOME and non-GNOME based applications on Unix platforms; OS X Accessibility Protocol [33] which supports applications on Mac OS X; and the Java Accessibility API (`javax.accessibility`) [34] which support Java applications.

Accessibility APIs are not capable of exposing the underlying functionality of applications to assistive technologies. Hence, users with disabilities, especially blind users, may experience difficulty building accurate mental models of an application since information collected by screen readers (for instance) at the interface level might not be sufficient to understand what the application is doing or can do. Moreover, developers, who have designed their GUIs mostly for non-disabled users, will need to expend extra development cost and effort in making their custom GUI controls accessible.

2.3 User Interface Architectural Models

Earlier methods of system development require that the user interface and the system's functionality be managed in the same component. However, the resulting effect of this approach is the production of systems that resist modification and present difficulty in providing for human factors [35]. The panacea to this problem is postulated to be Separable Interface Theory, which promotes the separation of the user interface from the functional aspects of a system [18]. This theory has become a building block on which systems are developed.

One of the most influential models that promote separable interface theory is the Seeheim model [36]. As shown in Figure 2.2, the Seeheim model has three components: presentation component, dialogue control, and the application interface model. The presentation component generates the physical user interface that the user interacts with. The presentation component interacts with the input/output devices to receive input data from users and to display output data to users. The presentation component sends input data to (and receives output data from) the application interface model via the dialogue control. In other words, data representing a user's requests are channeled through the dialogue control to the appropriate procedure in the application program, which performs necessary computations on the data, and then sends the results to the presentation component through the dialogue control. The application interface model represents the application, including the data objects, procedures, and constraints. These constraints are restrictions on the input supplied by users.

Similar to the Seeheim model, but with slightly different structure, is the Arch model [37].

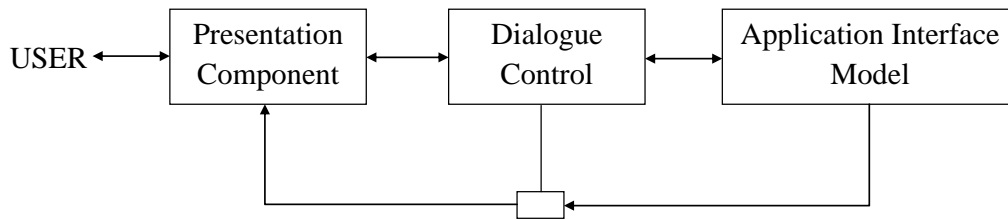


Figure 2.2: The Seeheim Model [36]

The goal of developing the Arch model is to generate user interfaces that manage the interaction between two externally-provided components - application domain functionality and the UI toolkits [37]. Five components make up the Arch model, with each serving a different purpose (as shown in Figure 2.3). The presentation component generates the virtual representation of the user interface (using presentation objects), while the interaction toolkit component generates the physical interface (using interaction objects supplied by UI Toolkit packages or software, e.g. Java Swing GUI toolkit, Google maps driver, speech drivers, etc.). The interaction objects in the physical interface correspond to the presentation objects in the virtual interface. Application-specific functions (including managing domain data) are handled by both the domain-specific and domain-adaptor components through the domain objects. The domain-adaptor component provides services related to the presentation of information, while the domain-specific components perform operations or computations that do not relate to the user interface. The dialogue component coordinates the interaction between the presentation component and the domain-adaptor component, in terms of data exchange and sequencing of tasks.

Other interface architectural models that complement the Seeheim model are Model-View-Controller (MVC) [38] and Passive-View-Command (PVC) [39].

Our work fits the separable interface theory, since the front-end interface interpreters are separated from the application engine, and the communication between both are facilitated by the Johar framework.

Alonso et al. [40] aimed to develop a user interface model for the blind. The model is built upon six human-computer interface models by supplementing each of them with additional requirements for blind users. The models are task model, domain model, dialog model, presentation model, platform model, and user model [40]. The summary of each model's additional requirements for the blind are: (1) the tasks described in the task model should fit the user's ability; (2) the domain model should define a sequence of windows and window

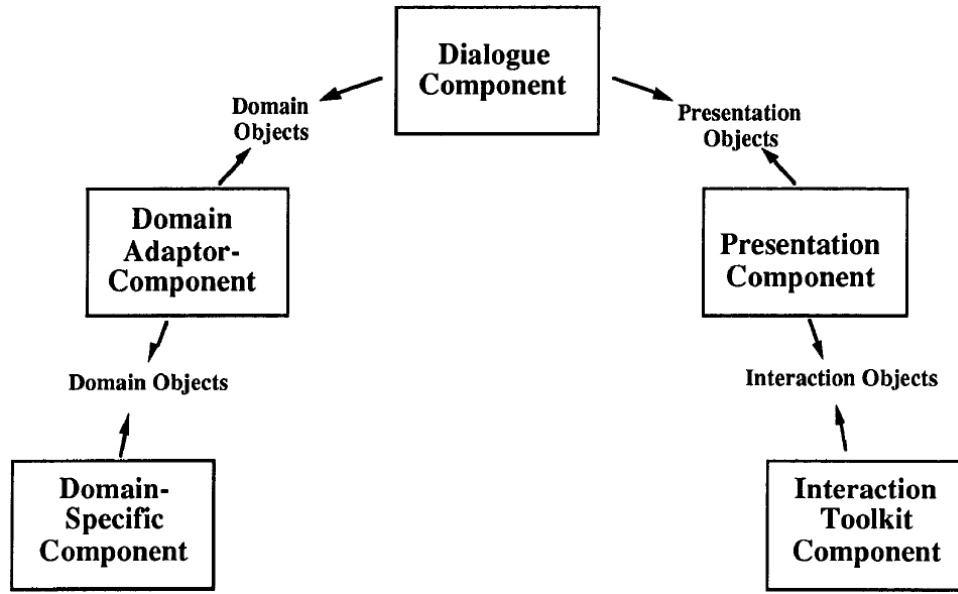


Figure 2.3: The Arch Model [37]

components that conform to 1-dimensional navigation and also provide speech and Braille output to support user interaction; (3) the dialog model should support keyboard or equivalent as interaction medium for input, while output messages should be received via speech synthesizers or Braille displays; (4) the presentation model should also define the speech and Braille output for each interface object, as well as the different levels of user experience; (5) the platform model should define the capabilities of speech and Braille output provided by standard APIs [40]; and (6) the user model should include certain configuration parameters, such as the speech parameters, Braille parameters, and detail level of output messages [40]. Our work complements that of Alonso et al, since their model can be used to build interface interpreters for the blind.

2.4 User Interface Description Languages

User interface management systems require a description of the user interfaces to be implemented in order to automatically (or semi-automatically) generate the interfaces [36]. This description of user interfaces can be achieved via user interface description languages (UIDLs). UIDLs allows interface designers to describe a user interface using high-level constructs or syntax which abstract away implementation details as well as details of input and output devices [41]. Most UIDLs are XML-compliant languages (in order to take advantage of XML's platform-independence), which can then be read directly by renderers or

interface generators to produce desired user interface for target platforms.

We studied four UIDLs, namely UIML (User Interface Markup Language) [42], XIML (eXtensible Interface Markup Language) [43], USIXML (USer Interface eXtensible Markup Language) [44], and AUIML (Abstract User Interface Markup Language) [45] in order to understand their approach to specifying user interfaces. In UIML, user interface elements (or parts) and their associated content (e.g. text, images, sound, etc.), presentation style (e.g. layout, colour, font, size, etc.) and interaction behaviour are represented using customized tags and vocabularies. The vocabularies are supplied by the User Interface toolkit(s) (e.g. Voice toolkit - for voice enabled interfaces, HTML toolkit - for web interfaces, Java AWT/Swing - for graphical user interfaces, etc.) required for presentation or rendering. UIML can also specify the connection of the user interface to the application logic.

Unlike UIML, USIXML describes a user interface in four levels of abstractions depending on the *context of use* [44]. The levels in order of abstraction (high to low) are *tasks and concepts*, *abstract user interface (AUI)*, *concrete user interface (CUI)*, and *final user interface (FUI)*. The tasks and concepts level views a user interface as composed of interrelated tasks and the domain (i.e. representation of real-world objects) in which the tasks are undertaken. The AUI level views a user interface as composed of interrelated abstract interaction objects, which can be grouped into various interaction spaces (i.e. containers). The CUI level describes the appearance (using concrete interaction objects which are closer to the physical widgets) and user's modality of interaction (graphical or auditory) with the interface. The FUI level presents the physical user interface shown to the user.

Similar to USIXML is XIML, which groups user interface components into abstract and concrete. The abstract components are *task*, *domain*, and *user*. The *task* and *domain* components are similar to *tasks and concepts* in USIXML, while the *user* component defines the characteristics of each user group or individual user. Furthermore, the concrete components in XIML are *dialog* and *presentation*, which are similar to the *concrete user interface* in USIXML. The presentation component describes the appearance of the user interface, while the dialog component describes the modality of interaction. Like USIXML, the relationship between interface elements within each component or across components is also captured. Unlike USIXML, XIML does not describe the platform-dependent and toolkit-dependent representation of the final interface shown to the user.

Finally, IBM developed AUIML for internal use with a view to defining user interfaces based on the purpose or intent of interaction, rather than appearance [45]. But AUIML, like other languages mentioned above, describes the look or layout of the user interface using toolkit-independent objects, such as GROUP, TREE, etc [46]. Similar to UIML, AUIML

specifies actions that associate application routines with these objects [47].

2.5 Personalized User Interface Generators

Personalized User Interface Generators are developed by researchers to automatically generate user interfaces tailored to individual users' ability, preferences, and needs. The fundamental requirement for all user interface generators is an abstract description of the interface they need to generate, and this is achievable using any of the available UIDLs. Some of the relevant user interface generators we studied are SUPPLE [14], SUPPLE++ [15], and EGOKI [16].

SUPPLE and SUPPLE++ generate user interfaces to support motor-impaired users in performing point-and-click tasks (i.e. tasks involving clicking, dragging, pointing, list selection, etc.). SUPPLE relies on an *interface specification*, *device model*, *usage model*, and a *cost function* in order to generate the desired user interface. The *interface specification* describes the types of data (including associated constraints) to be exchanged between the application and the user, as well as action types (which are useful for invoking application methods at run-time). The *device model* is a representation of available widgets, as well as device-specific constraints. The *usage model* represents usage patterns or traces generated as the user interacts with the user interface. These usage patterns are used by SUPPLE in determining the prominence or priority of interface widgets, during the layout process or customization. The *cost function* determines the presentation style and quality of the user interface. For SUPPLE, the model of user preferences is used as the cost function. In order to obtain the model of user preferences, each user interacts with a preference-elicitation tool (developed by the authors, called ARNAULD) which asks the user to choose from queries showing two functionally equivalent but different user interface fragments (for example, a user may be asked to choose between list box and combo box, radio buttons and combo box, check boxes and list box, etc.). Alternatively, the ARNAULD system may show the user a preview of how the user interface for an application will look when generated; afterwards, the user is asked to suggest improvements to this user interface. The user is also allowed to customize the user interface at run-time (such as re-arranging widgets or removing widgets). The authors of SUPPLE conducted experiments by generating user interfaces for non-disabled users, motor-impaired users, and users with slight vision impairment [14][48].

Similar to SUPPLE is SUPPLE++, which personalizes interfaces based on users' motor abilities, instead of user preferences. To achieve this, SUPPLE++ has a built-in ABILITY MODELER, which builds a model of users' motor abilities after observing their performance on pointing, dragging, list selection, and multiple clicking tasks. This ability model is then

used as the cost function to generate the ability-based user interface [15] [49].

EGOKI generates user interfaces that provide access to ubiquitous services. It requires a UIML document, and the target user’s capability as inputs. The UIML document specifies the structure, style, content, and behaviour of interface elements. Moreover, the UIML document contains all functionality provided by the service, as well as the media types (i.e. audio, video, text, image, etc.) for the resources supplied by the user interface toolkit. The service functionality and media types are assigned priorities in order to guide selection of interface resources or widgets during layout. Furthermore, EGOKI maintains a knowledge base containing information about each media type and its associated resources and possible adaptations, based on user capabilities. Each user capability is formed by combining each interaction modality¹ and each capability level². For instance, Table 2.1 shows an example of the capability of blind users, and the corresponding media type, resources, and adaptations.

Modality					Media Type	Resource	Adaptation
<i>Vision</i>	<i>Auditory</i>	<i>Speech</i>	<i>Motor</i>	<i>Cognitive</i>			
Null	High	High	High	High	Audio	Text	Navigation Support, Text Header
					Video	Text	
					Image	Text	
					Text	Text	

Table 2.1: An example of a blind user’s capability captured in the Knowledge Base

Furthermore, the EGOKI system begins the user interface generation process by selecting the appropriate resources and adaptations (using the target user’s capability to search the knowledge base for a match) for each functionality provided by the service. It then transforms the abstract user interface (described in the UIML document) into the final user interface, after confirming that the selected resources are available in the UI toolkit specified in the UIML document. The final user interface generated is in XHTML format, which can easily be rendered for display on target devices. Any extra adaptation or customization can be applied to the user interface using CSS (Cascading Style Sheets).

Our work complements those of SUPPLE and SUPPLE++, since the resources, such as those used in achieving automatic and user-driven customization of interface elements, can be incorporated into graphical interface interpreters. Finally, in contrast to our work, EGOKI’s final user interface can only be rendered as a Web interface on target devices, which may not

¹Vision, Auditory, Motor, Speech, and Cognitive [16]

²High, Low, and Null [16]

be suitable for blind users. Even though screen readers are available for web interfaces, their verbosity while reading screen contents can slow down blind users, thereby increasing task completion time.

2.6 Conclusion

From our research findings presented above, the following shortcomings are evident. The Johar framework is aimed at addressing these issues.

- (1) Assistive Technologies are not compatible with all applications, and also not suitable for all disabled users.
- (2) Most interface description languages require developers to describe specific patterns of user interactions (i.e. structure and appearance of interface elements) in their user interface definitions. This hinders accessibility, since some group of users may prefer some other interface structure or appearance not captured in these definitions. For example, some blind users may prefer a user interface that allows them to issue simple text commands to locate and open existing files in a Text Editor, instead of using screen readers to navigate a user interface composed of a dialog box and containers (for listing drives, directories, and files). This is due to the fact that screen readers will spend a lot of time reading the containers and their contents (the drives, directories, and files), and any other widgets in the dialog box. Certainly, this process will cause frustration for blind users, since it will hinder them from performing their intended tasks quickly.
- (3) It is very difficult to develop user interface generators that personalize user interfaces for all users, considering the wide range of user abilities and the dynamism involved. Users' abilities change and new abilities are created with time; thus, the development (and maintenance) cost and effort required to keep modifying these interface generators to cater for new or changes in abilities will be very difficult to bear.

The first and third shortcomings are addressed by developing interface interpreters that fit the abilities and needs of a variety of user groups (such as non-disabled users, blind users, motor-impaired users, low-vision users, deaf users, and so on). The second issue is addressed with our Interface Description File (IDF) which is not tied to any specific user interaction pattern, and offers the benefit of being readable by any interface interpreter.

Chapter 3

Quality Assurance on Johar

In this chapter, we extensively discuss our quality assurance process targeted towards the detection and correction of inconsistencies, omissions, irrelevances, and other loopholes that could hinder successful deployment of Johar.

The quality assurance process is aimed at improving the two core components of Johar (i.e. `johar.gem` and `johar.idf`), as well as detecting and correcting flaws in the IDF Format Specification document, IDF's XML Schema document, and the Interface Interpreter Specification document, which are all critical to the success of the Johar project. Furthermore, error detection and reporting tests were carried out on IDFs via our automated testing tool developed for this purpose.

3.1 Review and Redesign of Johar Components

There are two main components of Johar: `johar.gem` and `johar.idf` (see Section 1.1.4). Since the current Johar implementation is in Java, these components are packages containing several interrelated classes that facilitate the development of interface interpreters and apps.

3.1.1 The `johar.gem` package

The `johar.gem` package is responsible for the following activities:

- (1) Invoking methods in the app engine;
- (2) Communicating input data (i.e. commands, parameter values, and question responses elicited by the user via the interface interpreter) to the app engine, and also communicating output data from the app engine to the interface interpreter.

Figure 3.1 shows all the classes in the `johar.gem` package. *GemSetting* interfaces with interface interpreters, while *Gem* interfaces with app engines. In other words, an interface

interpreter sends user inputs to and receives output data from an app engine via *GemSetting*, while the app engine receives user inputs from and sends output data to the interface interpreter via *Gem*.

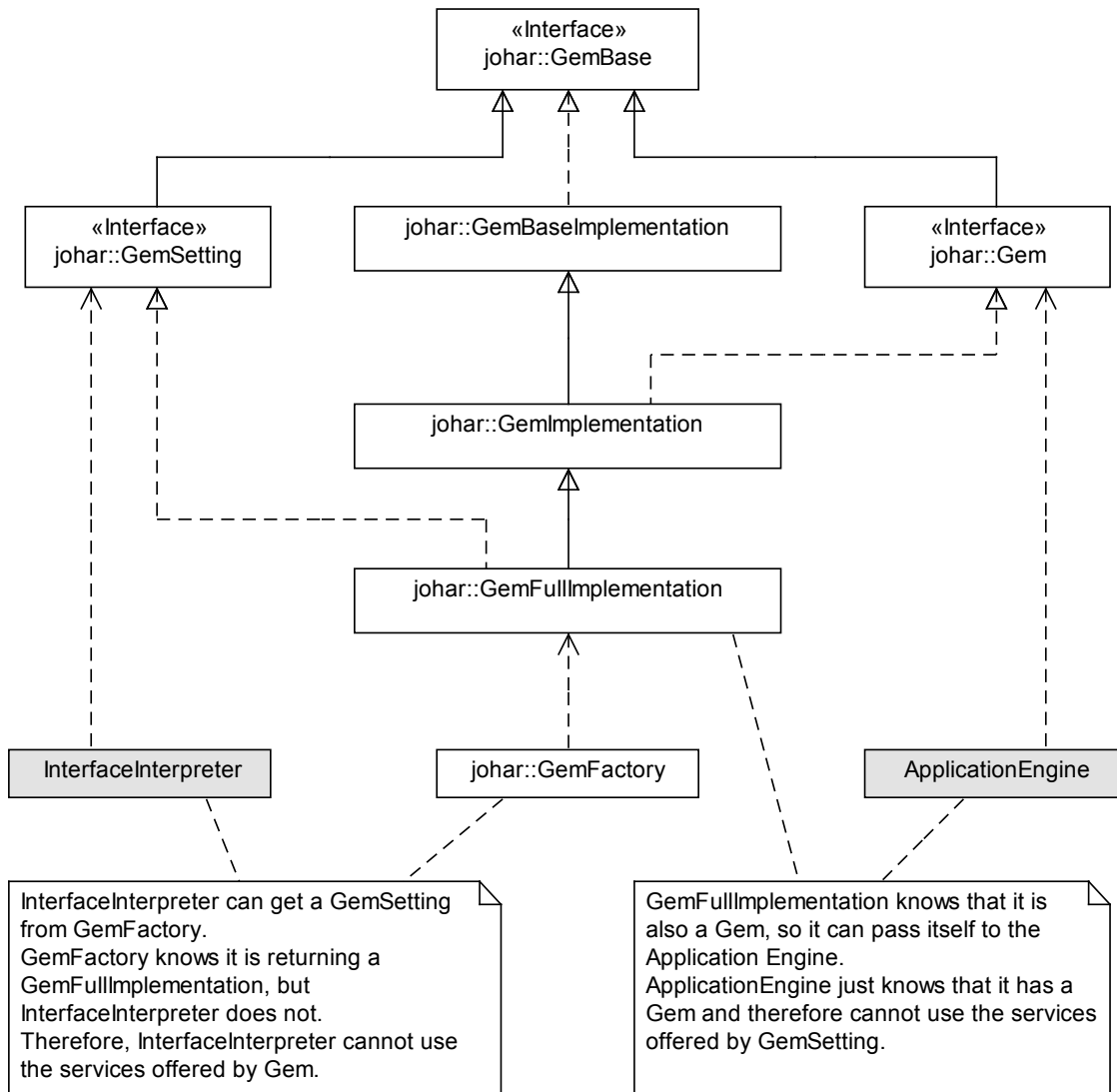


Figure 3.1: The Class Diagram of the `johar.gem` Package

The *GemFullImplementation* and *GemImplementation* are the brains behind the activities of *GemSetting* and *Gem* respectively. Specifically, *GemSetting* forwards the commands, parameter values, and question responses it received from the interface interpreter to *GemFullImplementation*, which then invokes the appropriate method in the app engine for computation to begin, and also makes the parameter values and question responses available to *GemImplementation* for use by the app engine through *Gem*. Furthermore, *Gem* forwards the table data or text it received from the app engine to *GemImplementation*, which then

makes the data accessible to *GemFullImplementation* for use by the interface interpreter through *GemSetting*.

The Review Process

Our review of the 2009 version of *johar.gem* package is aimed at determining whether the two activities outlined in subsection 3.1.1 are effectively carried out by the package. To achieve this aim, we reviewed the functionality of all the classes in this package.

The outcome of our review revealed the critical shortcomings outlined below. Afterwards, we explain the cause and effect of these shortcomings on the Johar project.

- (1) *GemSetting* did not have the feature needed to automatically initialize all tables that are supposed to hold table data coming from the app engine, when an interface interpreter is started by the user.
- (2) *GemSetting* did not have the feature needed to automatically validate all the app engine methods used in an IDF against the actual methods in the app engine.
- (3) *GemFullImplementation* was passive in determining which app engine method to invoke or trigger for computation to begin. It relies on *GemSetting* to provide the method name, which means *GemSetting* must get this information from the interface interpreter.
- (4) *GemFullImplementation* did not validate each parameter value received from *GemSetting* against the parameter type in the IDF. This means that *GemFullImplementation* could accept an integer value for a parameter declared as `float` in the IDF.

These shortcomings occurred because the 2009 version of *johar.gem* package did not access the IDF at all. As stated in chapter 1, the IDF provides information about the nature of an app (such as app name, name of app engine, app engine methods that perform specific functions, etc.), functionality or features available to users (such as commands, help facility, table browsing, parameter selection, answering questions, etc.), and the type of data to be exchanged between interface interpreters and the app (such as textual data, boolean values, integer values, files, calendar date and time, etc.). Unfortunately, as important as these pieces of information are, the 2009 version of *johar.gem* package had no knowledge of them.

The major effect or consequence of this knowledge gap is that the interface interpreter, which has access to the content of an IDF, must bridge the gap by initializing all tables at launch time, validating all app engine methods used in the IDF (which can only be achieved by collaborating with *GemSetting*), informing *GemSetting* which app engine method to invoke in order to perform specific task, and validating all parameter values against their corresponding

types in the IDF. This means extra work for interface interpreter developers, since they will have to worry about these issues, rather than focusing on developing quality ability-based user interfaces. In addition, the performance of interface interpreters is reduced by these extra checks and balances.

The Redesign Process

Our redesign process was aimed at restructuring the *GemSetting* and *GemFullImplementation* classes in the 2009 version of `johar.gem` package such that they could make certain decisions based on the content of IDFs without having to contact the interface interpreters for information.

We added new methods and rewrote existing methods in *GemFullImplementation* so as to address the shortcomings. These new and modified methods handle the following: initializing the app engine and all tables mentioned in the IDF, validating all the app engine methods used in an IDF against the actual methods in the app engine, automatically determining and invoking app engine methods to perform specific tasks (e.g. invoking command methods specified in the IDF, invoking methods to compute and retrieve default values for parameters, etc.), and validating each parameter value against its type in the IDF. These methods are accessible to the interface interpreter through *GemSetting*, and can be called when needed.

Thus, the interface interpreter developers can now focus majorly on the design of user interfaces. Also, the performance of interface interpreters is improved, thereby boosting user experience and satisfaction.

3.1.2 The `johar.idf` package

The `johar.idf` package provides access to the content of IDFs. Specifically, it is responsible for the following tasks:

- (1) Converting IDF to an XML document;
- (2) Validating the XML document in (1) above using an XML Schema Document (XSD) developed for this purpose;
- (3) Checking for other constraints¹ not captured in the XSD;
- (4) Reading the IDF's attribute instances contained in the validated XML document, and making them available for external access.

¹A summary of these other constraints is presented in subsection 3.3.2 of this chapter. The full details reside in the IDF Format Specification document which will be found in Appendix A of this thesis.

The Review Process

Our review at this phase covers the structure of the 2009 version of `johar.idf` package, in terms of how classes interrelate and how functions are allocated among the classes. The second phase of this review was conducted in conjunction with three other documents, as discussed in Section 3.2.

The outcome of our review revealed that only one class, called *IDF*, handled all the four major tasks outlined in subsection 3.1.2 above. Considering the fact that attributes in an IDF are structured in a tree-like manner (to showcase parent-child relationships among the attributes), having a single class to read and provide access to an IDF's content can obviously lead to code duplication, as well as difficulty in debugging, managing, maintaining, and reusing codes.

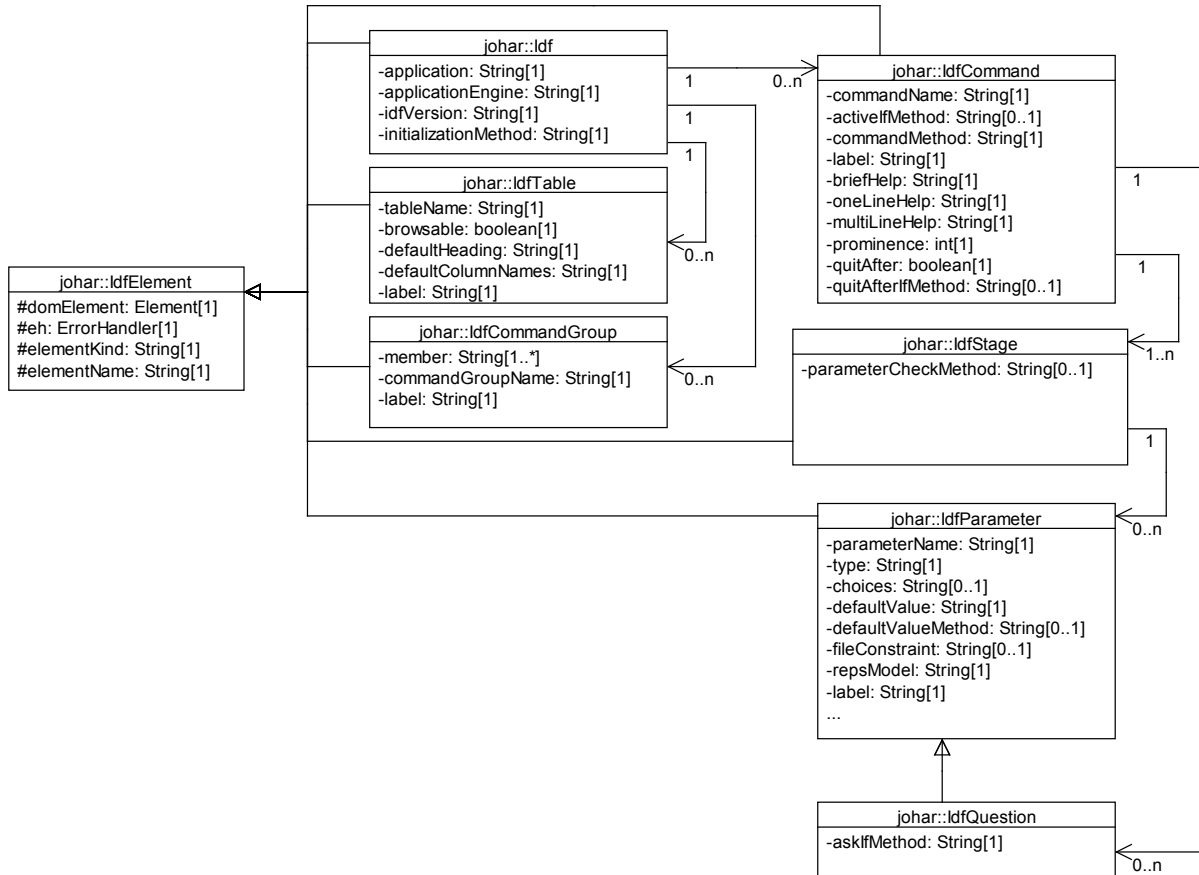
The Redesign Process

We decided to split the single class into multiple and manageable classes each of which performs a specific function and interacts with other classes. The class diagram in Figure 3.2 shows the new design of the `johar.idf` package.

The technique we used in creating the classes are as follows:

- For each attribute in the IDF that has sub-attributes (such as *Table*, *Command*, *CommandGroup*, *Stage*, *Parameter*, and *Question*), a class was created to provide access to that attribute and all its sub-attributes (including their respective values). The resulting classes are *IdfTable*, *IdfCommand*, *IdfCommandGroup*, *IdfStage*, *IdfParameter*, and *IdfQuestion*.
- The *Idf* class was created to convert an IDF document to an XML document, to validate the XML document against the XSD, and to check for violations against certain constraints specified in the IDF Format Specification document. In addition, the *Idf* class retrieves information about top-level attributes in the IDF. This information is: name of application, name of application engine, IDF version, application engine's initialization method, name of each command, name of each table, and name of each command group.
- The *IdfElement* class was created to expose all the elements, attributes, and values in the XML document. Thus, all other classes extract needed information from this class.

This new design makes the codes less error-prone, makes debugging easy, prevents code duplication, and makes code maintenance easy.

Figure 3.2: The Class Diagram of the `johar.idf` Package

3.2 Review of Johar-related Documents for Consistency

In furtherance of our quality assurance activity, we reviewed the following important documents for consistency. We conducted this review in March 2013. The latest version of the first three documents will be found in Appendix B, A, and C of this thesis respectively.

- **Interface Interpreter Specification document:** specifies the general behaviour of all interface interpreters.
- **IDF Format Specification document:** defines the syntax and semantics of all the attributes an IDF can contain, including the constraints on attributes and values.
- **XML Schema document (XSD):** used for validating the XML equivalent of an IDF.
- **`johar.idf` package:** provides access to the content (i.e. attributes and their values) of an IDF.

The last three documents are obviously related (since they involve *specifying*, *validating*, and *exposing the content* of IDFs). The first document (i.e. Interface Interpreter Specification document) is also related to other documents, since each interface interpreter accepts an IDF as input. Hence, all these documents must be consistent or harmonious with one another.

Following standard practices in software specification, each of the documents was mostly made up of short numbered paragraphs. The aim of this review is to verify whether every sentence in the IDF Format Specification document is traceable to some requirements or code in one of the other documents. Thus, we worked through each document paragraph by paragraph, and took note of every sentence (including the section number and the attribute concerned) in the IDF Format Specification document that cannot be traced to some requirements or code in one of the other documents.

Table 3.1 shows part of the results of our review process. The full results can be found in Appendix D of this thesis. For clarity, classes in the `johar.idf` package and the XML Schema Document (`johar.xsd`) are italicized in the table, while attributes are printed in Typewriter font.

Section	Attribute	Sentence	Violation
2.1	<code>InitializationMethod</code>	The name of the application engine method used to initialize the engine.	This attribute is not available in current <i>johar.xsd</i> version, and not captured in <i>Idf</i> .
2.2	<code>Prominence</code>	Default value: 2000	Default value specified in <i>IdfCommand</i> is 1000.
2.4	<code>FileConstraint</code>	Multiplicity: For parameters of type file, 0 or 1.	In <i>IdfParameter</i> , the multiplicity specified is 1.

Table 3.1: Some results of the review of Johar-related documents for consistency

The first row of Table 3.1 specifies that the `InitializationMethod` attribute, which was defined in Section 2.1 of the IDF Format Specification document, was not implemented in both the XML Schema document (*johar.xsd*) and the *Idf* class of `johar.idf` package. The implication of this omission is that whenever the *Idf* class attempts to validate the XML equivalent of an IDF containing the `InitializationMethod` attribute, an error occurs since `InitializationMethod` is never captured in *johar.xsd*. In addition, since the *Idf* class is meant to provide information about an app engine's initialization method (which is the value

of the `InitializationMethod` attribute in the IDF) when requested by the *GemFullImplementation* class of `johar.gem` package, it becomes impossible for *GemFullImplementation* to successfully initialize the app engine whenever an interface interpreter is launched by the user.

As shown in Appendix D of this thesis, a total of 24 consistency violations were reported, with 4 additional consistency-related observations. Sections 2.1 to 2.7 in the IDF Format Specification document were the problematic sections, with section 2.4 (titled “Sub-Attributes of Parameter”) having the highest number of consistency violations. We addressed all consistency violations by effecting appropriate changes in the documents concerned. For example, the first consistency violation in Table 3.1 occurred in the *Idf* class of `johar.idf` package and the XML Schema document (i.e. *johar.xsd*); in this case, we inserted some code into the *Idf* class to capture the missing `InitializationMethod` attribute and also added a schema element for the `InitializationMethod` attribute in *johar.xsd*. In addition, since the value of `InitializationMethod` attribute is needed to initialize an app engine, we added a method call (through which the attribute’s value can be retrieved from the *Idf* class) to the *GemFullImplementation* class of `johar.gem` package.

Thus, the results of this review assisted us in fixing code-related issues in the Johar packages, modifying and/or adding requirements to the IDF Format and Interface Interpreter Specification documents, as well as fixing omissions and bugs in the XML Schema document.

3.3 Test Infrastructure for IDFS

Although we have manually identified and fixed various IDF-related issues during our review processes, we still need to answer the following question:

- Given any IDF, can Johar detect and report all the invalid entries in it?

Answering the question above requires correctly providing answers to the following questions:

- Is the IDF correctly transformed to an XML document?
- Is the XML Schema Document (XSD) accurately validating IDFS?
- Are other constraints not captured in the XSD checked by relevant classes in `johar.idf` package in order to detect and report violations?

Even though we can guarantee that Johar detects, flags invalid entries (such as invalid attributes, wrong attribute values, omission or wrong placement of vital special characters, etc.) as errors, and provides detailed error reports or messages, we cannot ascertain that “all”

errors are detected and reported. This issue prompted us to develop an automated testing tool which accepts an annotated IDF as input, automatically generates many test cases from the IDF, runs all the test cases, and then produces error reports with which accurate decisions (such as answering all the questions above) can be made.

```

Application = ContactsManager //Required
ApplicationEngine = ContactsManager //Optional
IdfVersion = "1.0" //Required
InitializationMethod = initContactsManager //Optional

CommandGroup contacts = {
  Member = addContact //Required
}
Command addContact = {
  BriefHelp = "Adds a New Contact" //Optional
  Stage contactProfile = {
    Parameter fullName = {
      Type = text //Required
      MaxNumberOfChars = 0 //Forbidden
      Label = "Name (Last, First)" //Optional
    }
    Parameter age = {
      Type = int //Required
      MinValue = 18 //Optional
      MaxValue = 15 //Forbidden
    }
    ParameterCheckMethod = checkInputValues //Optional
  }
  Question addContactQuestion = {
    Type = boolean //Required
    Label = "Continue with adding contact?" //Optional
    AskIfMethod = confirmAddAction //Required
  }
}

```

Figure 3.3: A sample annotated IDF for generating test cases

3.3.1 Generating Test Cases

The test cases are generated automatically from an annotated IDF. By “annotated” we mean appending a comment to each attribute instance in the IDF. Each comment is preceded with a “//” followed by the text “Optional”, “Required”, or “Forbidden”. The “//Optional” and “//Required” comments are used to identify optional and required attribute instances in the IDF, as defined in the IDF Format Specification document. The “//Forbidden” comment, on

the contrary, is used to identify attribute instances or entries in the IDF that violate some requirements or constraints in the IDF Format Specification document. A sample annotated IDF is shown in Figure 3.3.

Our test objective is to generate many valid and invalid test cases from the annotated IDF, and then observing the results after running them. Under normal circumstances, the valid test cases should not produce any error, while each invalid test case should produce only one error (or related errors in some cases). Thus, a valid test case that produces an error is a *failed test case*; similarly, an invalid test case that does not produce an error is a *failed test case*. However, failed test cases are very important to us because they reveal bugs in the `johar.idf` package or the XML Schema Document (XSD), or faults in the IDF Format Specification document.

Furthermore, we grouped our test cases into three test suites - TS 1, TS 2, and TS 3. TS 1 contains valid test cases, while TS 2 and TS 3 each contains invalid test cases.

Procedure for Generating Valid Test Cases in TS 1

- (1) Generate each valid test case by deleting all the *Forbidden* attribute instances and one *Optional* attribute instance from the annotated IDF;
- (2) Delete all comments or annotations from the test cases.

Figures 3.4 and 3.5 show two test cases in TS 1, generated from the annotated IDF in Figure 3.3. In Test Case 1, the first *Optional* attribute instance (i.e. `ApplicationEngine = ContactsManager`) and all *Forbidden* instances were deleted from the annotated IDF; while in Test Case 2, the second *Optional* attribute instance (i.e. `InitializationMethod = initContactsManager`) and all *Forbidden* instances were deleted. Five other test cases can be generated by deleting one of the remaining *Optional* attribute instances (and all *Forbidden* instances) for each test case. The procedure for generating test cases for each test suite is given below.

Procedure for Generating Invalid Test Cases in TS 2

- (1) Generate each invalid test case by deleting all the *Forbidden* attribute instances and one *Required* attribute instance from the annotated IDF;
- (2) Delete all comments or annotations from the test cases.

Figures 3.6 and 3.7 show two test cases in TS 2, generated from the annotated IDF in Figure 3.3. In Test Case 1, the first *Required* attribute instance (i.e. `Application = ContactsManager`) and all *Forbidden* instances were deleted from the annotated IDF; while in Test Case 2, the

```
Application = ContactsManager
IdfVersion = "1.0"
InitializationMethod = initContactsManager

CommandGroup contacts = {
  Member = addContact
}
Command addContact = {
  BriefHelp = "Adds a New Contact"
  Stage contactProfile = {
    Parameter fullName = {
      Type = text
      Label = "Name (Last, First)"
    }
    Parameter age = {
      Type = int
      MinValue = 18
    }
    ParameterCheckMethod = checkInputValues
  }
  Question addContactQuestion = {
    Type = boolean
    Label = "Continue with adding contact?"
    AskIfMethod = confirmAddAction
  }
}
```

Figure 3.4: Test Case 1 in TS 1

```
Application = ContactsManager
ApplicationEngine = ContactsManager
IdfVersion = "1.0"

CommandGroup contacts = {
  Member = addContact
}
Command addContact = {
  BriefHelp = "Adds a New Contact"
  Stage contactProfile = {
    Parameter fullName = {
      Type = text
      Label = "Name (Last, First)"
    }
    Parameter age = {
      Type = int
      MinValue = 18
    }
    ParameterCheckMethod = checkInputValues
  }
  Question addContactQuestion = {
    Type = boolean
    Label = "Continue with adding contact?"
    AskIfMethod = confirmAddAction
  }
}
```

Figure 3.5: Test Case 2 in TS 1

```

ApplicationEngine = ContactsManager
IdfVersion = "1.0"
InitializationMethod = initContactsManager

CommandGroup contacts = {
  Member = addContact
}
Command addContact = {
  BriefHelp = "Adds a New Contact"
  Stage contactProfile = {
    Parameter fullName = {
      Type = text
      Label = "Name (Last, First)"
    }
    Parameter age = {
      Type = int
      MinValue = 18
    }
    ParameterCheckMethod = checkInputValues
  }
  Question addContactQuestion = {
    Type = boolean
    Label = "Continue with adding contact?"
    AskIfMethod = confirmAddAction
  }
}
}

```

Figure 3.6: Test Case 1 in TS 2

second *Required* attribute instance (i.e. `IdfVersion = "1.0"`) and all *Forbidden* instances were deleted. Five other test cases can be generated by deleting one of the remaining *Required* attribute instances (and all *Forbidden* instances) for each test case.

Procedure for Generating Invalid Test Cases in TS 3

- (1) Generate each invalid test case by deleting all but one *Forbidden* attribute instance from the annotated IDF. In other words, if there are n *Forbidden* attribute instances in the annotated IDF, then delete $n-1$ *Forbidden* attribute instances for each test case;
- (2) Delete all comments or annotations from the test cases.

Figures 3.8 and 3.9 show two test cases in TS 3, generated from the annotated IDF in Figure 3.3. In Test Case 1, the first *Forbidden* attribute instance (i.e. `MaxNumberOfChars = 0`) was not

```
Application = ContactsManager
ApplicationEngine = ContactsManager
InitializationMethod = initContactsManager

CommandGroup contacts = {
    Member = addContact
}
Command addContact = {
    BriefHelp = "Adds a New Contact"
    Stage contactProfile = {
        Parameter fullName = {
            Type = text
            Label = "Name (Last, First)"
        }
        Parameter age = {
            Type = int
            MinValue = 18
        }
        ParameterCheckMethod = checkInputValues
    }
    Question addContactQuestion = {
        Type = boolean
        Label = "Continue with adding contact?"
        AskIfMethod = confirmAddAction
    }
}
```

Figure 3.7: Test Case 2 in TS 2

deleted, but all other *Forbidden* attribute instances were deleted from the annotated IDF; while in Test Case 2, the second *Forbidden* attribute instance (i.e. `MaxValue = 15`) was not deleted but all other *Forbidden* attribute instances were deleted. Note that the first of these attribute instances is marked as “Forbidden” because the expected value of `MaxNumberOfChars` must be greater than or equal to 1, and that the second is marked as “Forbidden” because `MaxValue` must not be less than `MinValue` (as stated in the IDF Format Specification document).

```
Application = ContactsManager
ApplicationEngine = ContactsManager
IdfVersion = "1.0"
InitializationMethod = initContactsManager

CommandGroup contacts = {
  Member = addContact
}
Command addContact = {
  BriefHelp = "Adds a New Contact"
  Stage contactProfile = {
    Parameter fullName = {
      Type = text
      MaxNumberOfChars = 0
      Label = "Name (Last, First)"
    }
    Parameter age = {
      Type = int
      MinValue = 18
    }
    ParameterCheckMethod = checkInputValues
  }
  Question addContactQuestion = {
    Type = boolean
    Label = "Continue with adding contact?"
    AskIfMethod = confirmAddAction
  }
}
```

Figure 3.8: Test Case 1 in TS 3


```

Application = ContactsManager
ApplicationEngine = ContactsManager
IdfVersion = "1.0"
InitializationMethod = initContactsManager

CommandGroup contacts = {
  Member = addContact
}
Command addContact = {
  BriefHelp = "Adds a New Contact"
  Stage contactProfile = {
    Parameter fullName = {
      Type = text
      Label = "Name (Last, First)"
    }
    Parameter age = {
      Type = int
      MinValue = 18
      MaxValue = 15
    }
    ParameterCheckMethod = checkInputValues
  }
  Question addContactQuestion = {
    Type = boolean
    Label = "Continue with adding contact?"
    AskIfMethod = confirmAddAction
  }
}
}

```

Figure 3.9: Test Case 2 in TS 3

3.3.2 Running the Test Cases

As shown in Figure 3.10, the automated testing tool (which is written in Java) runs the test cases, and generates an error log for each test case, as well as a summary report of the tests. The Execution module of the automated testing tool runs each test case using the algorithm below:

```

For each test suite
  For each test case in test suite
    Convert test case to XML document
    Validate XML document against the XSD
    Check for violations against other constraints
    Generate Error Log

```

Next

Next

The automated testing tool interacts with the `johar.idf` package for the purpose of generating and validating each XML document against the XSD, including checking for violations against certain constraints. These constraints are defined in the IDF Format Specification document, and are summarized below:

- Both the minimum value and maximum value of certain attributes must be within specified thresholds, and the minimum value must be less or equal to the maximum value;
- A source table must be provided for parameters of type `tableEntry`;
- All command and parameter names must be unique;
- All help messages must not exceed a particular length;
- A command must belong to only one command group;
- The minimum and maximum number of occurrence of certain attributes (especially attributes with no sub-attributes) must not be exceeded.

3.3.3 Interpreting the Test Results

The expected test results are the *Error Log for each test case*, and the *Summary Report*.

The Error Log

The Error Log is a text file containing error reports generated while running each test case. The Error Log's content is structured into three parts:

- (1) *Action Taken on the Annotated IDF*: This part describes the attribute instance(s) removed from the annotated IDF to form the test case. We used this information to determine whether the error(s) detected (and reported) were in agreement with what we expected.
- (2) *IDF to XML Conversion Errors*: This part contains all the errors that occurred while converting the test case to an XML document. We used this error report to locate and correct bugs in the IDF to XML conversion module.

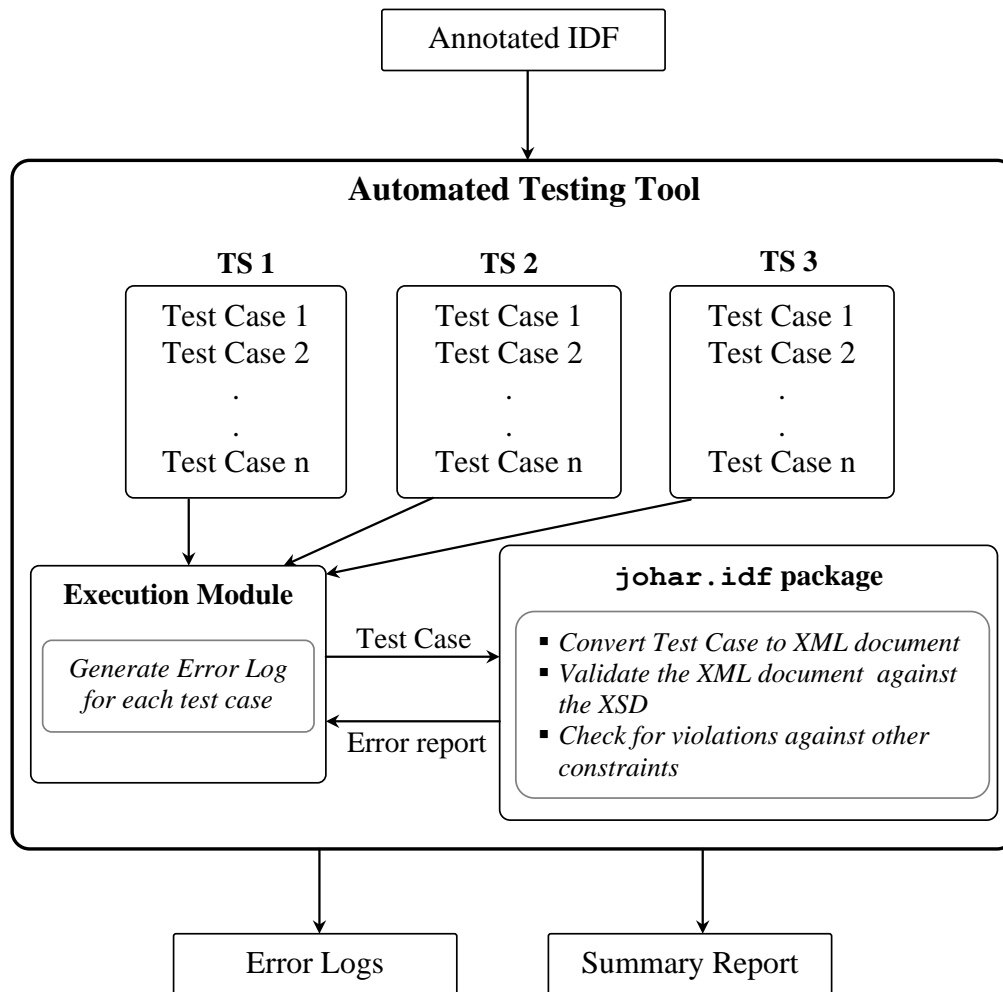
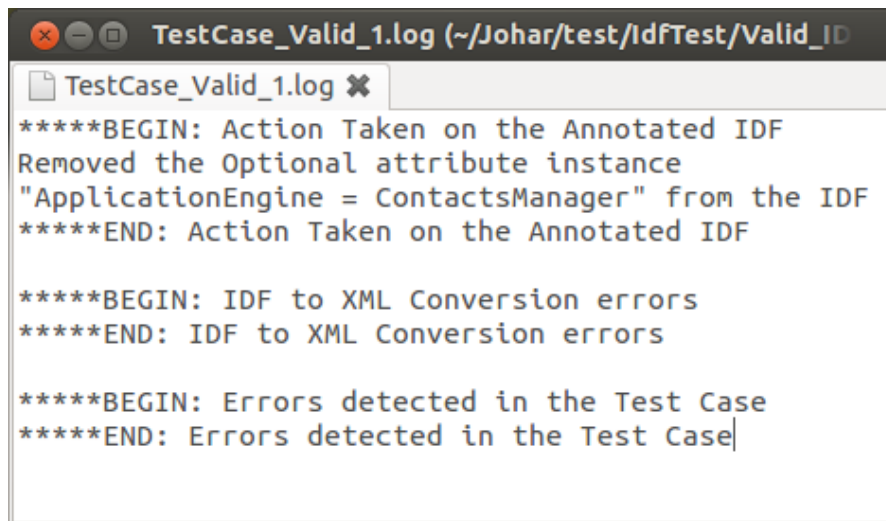


Figure 3.10: Architecture of the Automated Testing Tool

- (3) *Errors detected in the Test Case:* This part contains all XML validation errors (produced while checking the XML document against the XSD), and violations against other constraints specified in the IDF Format Specification document. These errors are linked to invalid entries in the test case. We used this error report to determine whether all the invalid entries in the test case were detected and reported.

The error logs for the test cases in Figures 3.4 and 3.6 are shown in Figures 3.11 and 3.12 respectively.



```

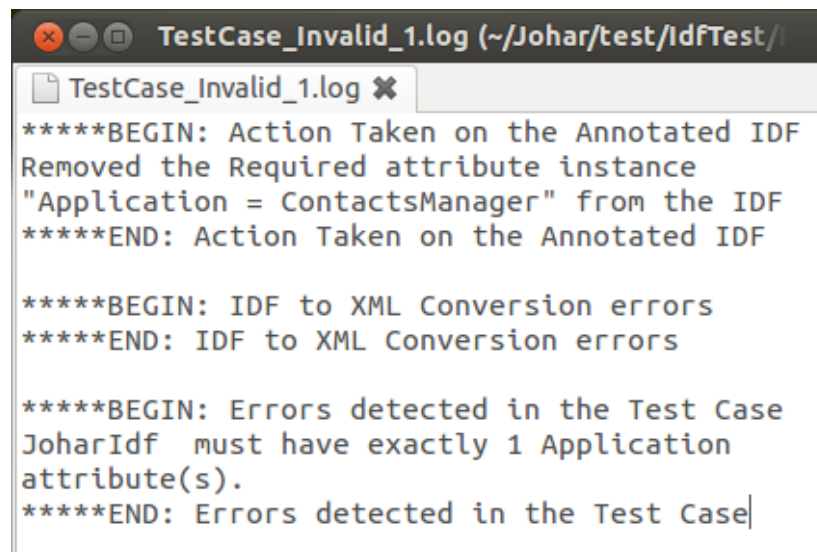
*****BEGIN: Action Taken on the Annotated IDF
Removed the Optional attribute instance
"ApplicationEngine = ContactsManager" from the IDF
*****END: Action Taken on the Annotated IDF

*****BEGIN: IDF to XML Conversion errors
*****END: IDF to XML Conversion errors

*****BEGIN: Errors detected in the Test Case
*****END: Errors detected in the Test Case|

```

Figure 3.11: Error Log for Test Case 1 in TS 1. [No error is detected]



```

*****BEGIN: Action Taken on the Annotated IDF
Removed the Required attribute instance
"Application = ContactsManager" from the IDF
*****END: Action Taken on the Annotated IDF

*****BEGIN: IDF to XML Conversion errors
*****END: IDF to XML Conversion errors

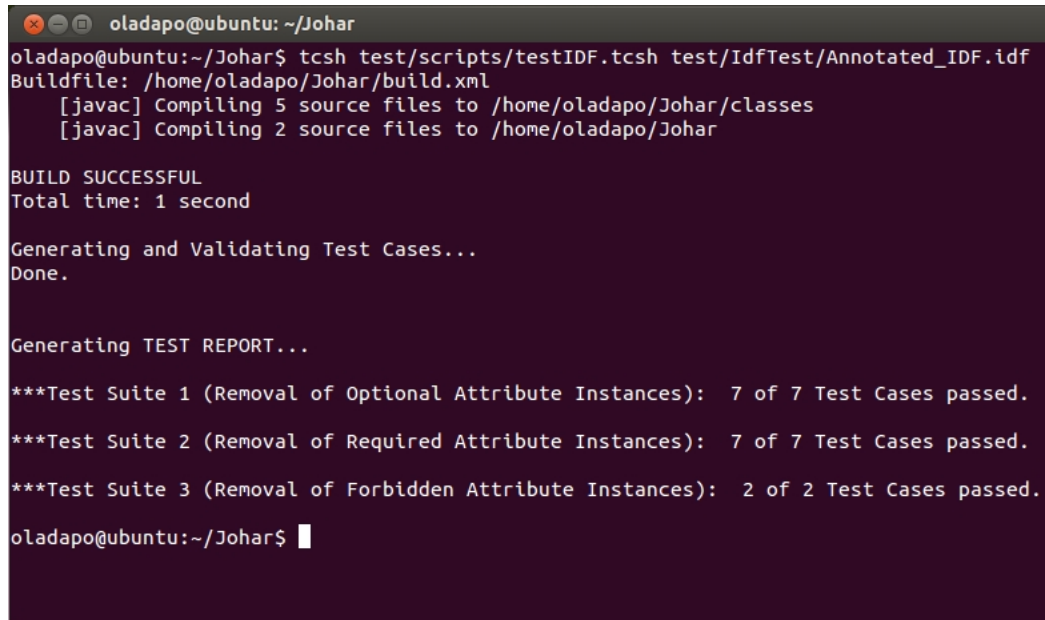
*****BEGIN: Errors detected in the Test Case
JoharIdf must have exactly 1 Application
attribute(s).
*****END: Errors detected in the Test Case|

```

Figure 3.12: Error Log for Test Case 1 in TS 2. [An error is detected]

The Summary Report

The summary report, shown in Figure 3.13, presents the overall test result at a glance. From this report, we can easily determine the total number of test cases generated, the number of test cases that passed or failed, and the test cases that failed (if any) for each test suite.



```

oladapo@ubuntu: ~/Johar
oladapo@ubuntu:~/Johar$ tclsh test/scripts/testIDF.tclsh test/IdfTest/Annotated_IDF.idf
Buildfile: /home/oladapo/Johar/build.xml
[javac] Compiling 5 source files to /home/oladapo/Johar/classes
[javac] Compiling 2 source files to /home/oladapo/Johar

BUILD SUCCESSFUL
Total time: 1 second

Generating and Validating Test Cases...
Done.

Generating TEST REPORT...

***Test Suite 1 (Removal of Optional Attribute Instances): 7 of 7 Test Cases passed.
***Test Suite 2 (Removal of Required Attribute Instances): 7 of 7 Test Cases passed.
***Test Suite 3 (Removal of Forbidden Attribute Instances): 2 of 2 Test Cases passed.

oladapo@ubuntu:~/Johar$

```

Figure 3.13: Summary Report of the tests

Determining test cases that passed or failed

Recall from subsection 3.3.1 that TS 1 contains valid test cases which should not produce error when executed; while TS 2 and TS 3 both contain invalid test cases that should each produce only one error (or related errors in some cases) when executed. Thus, we determine passed and failed test cases as follows:

- For each test case in TS 1,
 - If test case's error log contains no error, then
 - Flag test case as "Passed"
 - Otherwise,
 - Flag test case as "Failed"
- Next
- For each test case in TS 2,
 - If test case's error log contains no error, then
 - Flag test case as "Failed"

```
        Otherwise,  
            Flag test case as "Passed"  
    Next  
  
    • For each test case in TS 3,  
        If test case's error log contains no error, then  
            Flag test case as "Failed"  
        Otherwise,  
            Flag test case as "Passed"  
    Next
```

For example, the valid test cases in Figures 3.4 and 3.5 *passed* because their error logs (one of which is shown in Figure 3.11) contain no error message. Moreover, the invalid test cases in Figures 3.6, 3.7, 3.8, and 3.9 also *passed* since their error logs (one of which is shown in Figure 3.12) each contain an error message.

In conclusion, Figure 3.13 is a demonstration of the automated testing tool. The testing tool accepts the annotated IDF in Figure 3.3 as input, and then proceeds to generate and validate the valid and invalid test cases. The summary report shows that 7 valid test cases are in TS 1 (two of which are shown in Figures 3.4 and 3.5), 7 invalid test cases are in TS 2 (two of which are shown in Figures 3.6 and 3.7), and 2 invalid test cases are in TS 3 (as shown in Figures 3.8 and 3.9). The report further reveals that all the test cases in TS 1, TS 2, and TS 3 passed the tests.

In order to achieve our test objective which is to determine whether all errors in an IDF are detected and reported, we wrote different annotated IDFs and fed each of them to the automated testing tool as input. Prior to commencing the tests, we checked the annotated IDFs to make sure that they (collectively) cover all the attributes an IDF can contain, as defined in the IDF Format Specification document. After running all the test cases generated from the annotated IDFs, the automated testing tool revealed several failed test cases. The Error Log of each failed test case contained between 1 to 10 faults (or errors) which are linked to bugs in the `johar.idf` package and/or the XML Schema document (`johar.xsd`). We resolved all the bugs and repeated the tests to verify whether all the test cases would pass. As expected, all the test cases passed.

The latest source files of the Johar framework are available in our online repository ².

²<https://github.com/jamieandrews/johar-git>

Chapter 4

The Star Interface Interpreter

In this chapter, we discuss a new interface interpreter which is based on the new version of Johar. This interface interpreter is called Star, and its overall purpose is to present WIMP (Windows, Icons, Menus, and Pointers) graphical user interfaces to users. Star supports most of the common interaction styles evident in GUIs, such as menu selection, data entry and selection, table browsing, wizard-style interaction, customization of widgets (i.e. addition, rearrangement and deletion of widgets), and so on.

Star itself is not an accessible application, but a means through which application developers present their apps to users. Most application developers will preview and test their apps using Star in order to ascertain that their apps are working as expected and that all functionality or features have been implemented. Thus, successful development of Star is very important to the overall success of the Johar project. We therefore focused on Star as the interface interpreter with the highest priority for implementation. Other interface interpreters, including prototypes providing accessibility to some groups of disabled users, will be discussed in Chapter 5.

This chapter covers the specification and design of Star, as well as its implementation and demonstration.

4.1 Requirements Specification of Star GUI

In this section, we describe the features available in a Star graphical user interface. A more technical description of these features is presented in the Star GUI Specification document¹.

¹Available in Appendix E of this thesis.

4.1.1 The Main Panel

The Main Panel, in Figure 4.1, is displayed when the user launches an application using Star. It encompasses four components: the Menu Bar, Text Display Area, Table Area, and the Status Bar.

The Menu Bar

The Menu Bar contains application menus labelled with the names of command groups in the IDF. Each menu consists of menu items labelled with the names of commands in the IDF. Thus, these menu items are the commands through which the user can specify his/her intent. For example, the user can specify his/her intent to print the current document by selecting the “Print” menu item. Furthermore, the Menu Bar has a special menu, called *Star*, whose menu items represent services provided by the Star interface interpreter for all Johar applications (e.g. *Help* is a menu item under the *Star* menu).

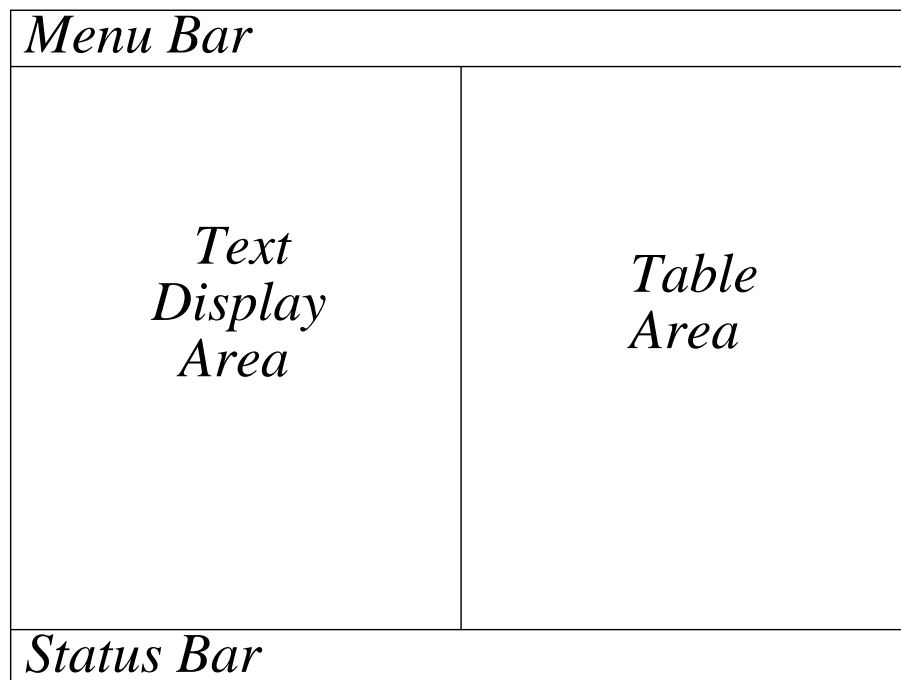


Figure 4.1: The Main Panel of Star GUI

The Text Display Area

This is a scrollable text widget that displays textual data requested by the user, and/or textual data representing the output of a computation. The Text Display Area also displays notifications sent from the application engine. The contents of the Text Display Area are

separated from one another via a horizontal line appended to each message or data. The Text Display Area is located to the left of the Table Area under the Menu Bar.

The Table Area

This is a scrollable widget that displays browsable tables, which are populated with data by the application engine, and whose rows are selectable by the user. The Table Area has tabs, one for each table. Selected rows of data are values of parameters of type `tableEntry` whose source table is the table containing the selected rows. Thus, the selected rows can be given to a command as input values for use by the application engine during computation. For example, suppose the Table Area contains an *Appointment* table where each row represents an appointment; a user can select an appointment from the table, and then select the “Cancel Appointment” command to cancel that appointment.

The Status Bar

The Status Bar usually displays short messages indicating the completion of an operation and/or simple success and failure information. It can also be used to display any low-priority message to the user. The Status Bar is located at the bottom of the Main Panel.

4.1.2 The Command Dialog Box

The Command Dialog Box in Figure 4.2 is displayed when the user selects a menu item or command under an application menu. The Command Dialog Box contains a section for each parameter of the command, as specified in the IDF.

Each parameter section contains one of the various data entry widgets, depending on the parameter type (which can be either `text`, `boolean`, `int`, `float`, `choice`, `file`, `date`, or `timeOfDay`). Moreover, each parameter section is located in a particular stage of the Command Dialog Box, depending on which stage of the command the parameter belongs to in the IDF. In other words, if a parameter belongs to the second stage of the selected command (as specified in the IDF), then the data entry widget for that parameter will be placed in the second stage of the Command Dialog Box. Thus, the Command Dialog Box is structured like a Wizard, where the user can navigate from one stage to another using the “Previous” and “Next” buttons. These two buttons are shown automatically if the selected command has more than one stage, but hidden if only one stage exists. Furthermore, the user can prematurely terminate the command by clicking the “Cancel” button, or complete his/her task by clicking the “OK” button.

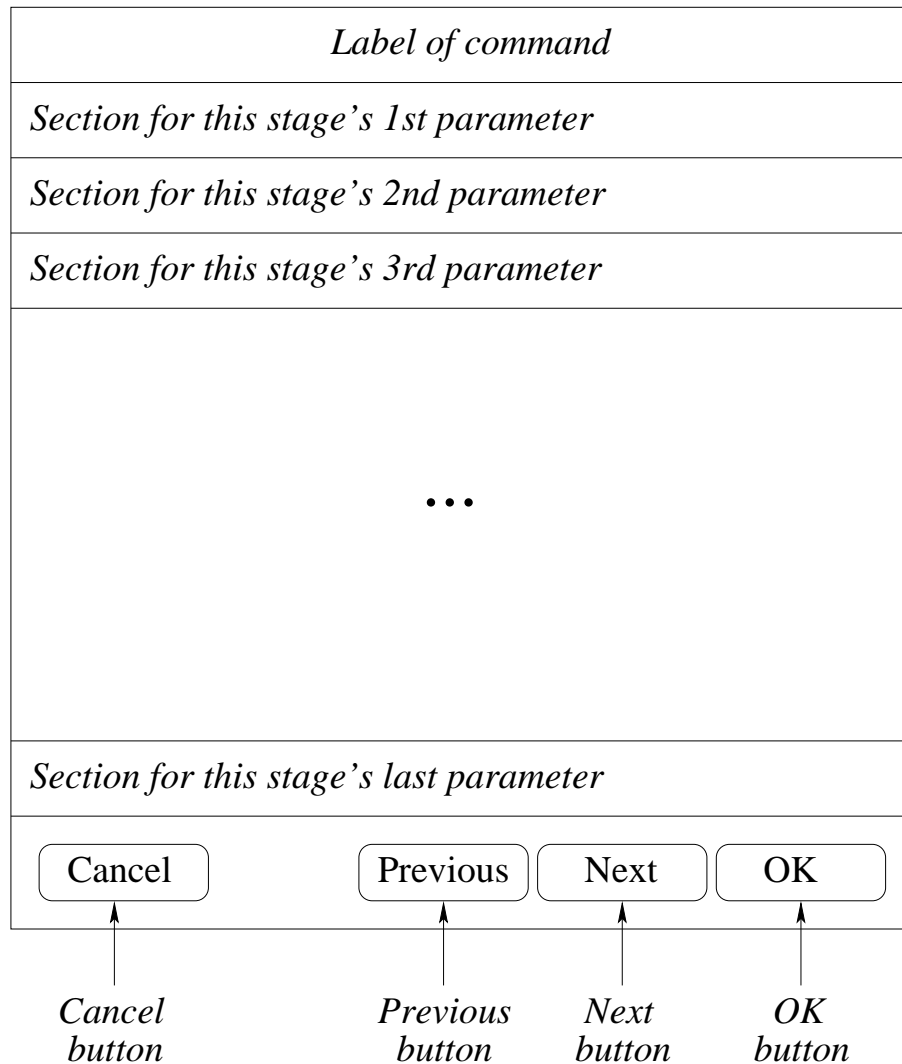


Figure 4.2: The Command Dialog Box of Star GUI

The Parameter Section of the Command Dialog Box

The parameter section of the Command Dialog Box, shown in Figure 4.3, contains the parameter's label (as specified in the IDF) and the data entry widget for the parameter. Based on the number of repetitions specified in the IDF for the parameter, there can be more than one data entry widget. Examples of data entry widgets include text boxes and drop-down menus. Also, based on the minimum number of repetitions specified in the IDF, some data entry widgets may be visible at first, but more can be added by clicking the “+” button. The user is allowed to delete each unwanted widget down to the minimum number of repetitions (by clicking the “X” button), and allowed to add more widgets up to the maximum number of repetitions (as specified in the IDF).

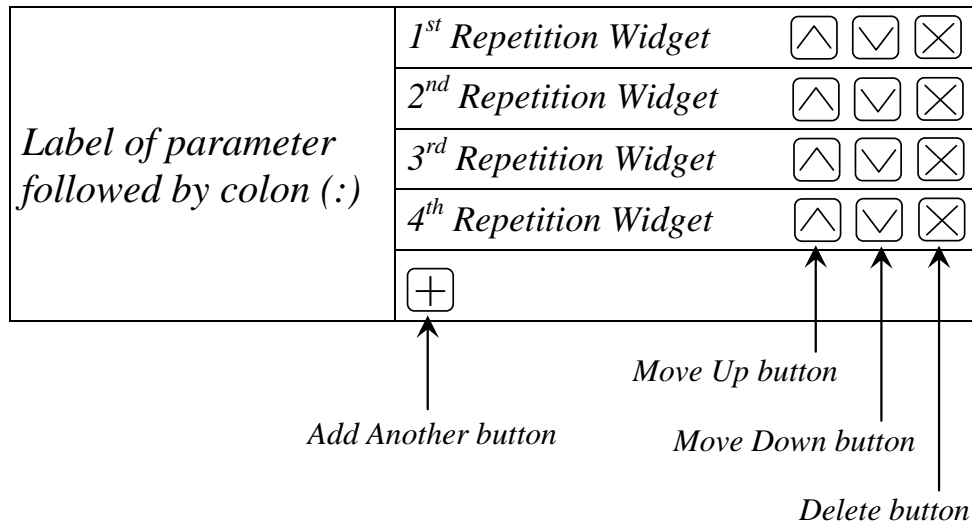


Figure 4.3: The Parameter Section of the Command Dialog Box

Finally, if parameter values are to follow a particular order, the user is allowed to rearrange the widgets accordingly by clicking the “^” and “v” buttons.

4.1.3 The Question Dialog Box

The Question Dialog Box, shown in Figure 4.4, is used to ask questions. It is composed of the question’s label (as specified in the IDF), a widget for supplying value in response to the question, an “OK” button, and a “Cancel” button. The “OK” button confirms the user’s response, while the “Cancel” button discards the question.

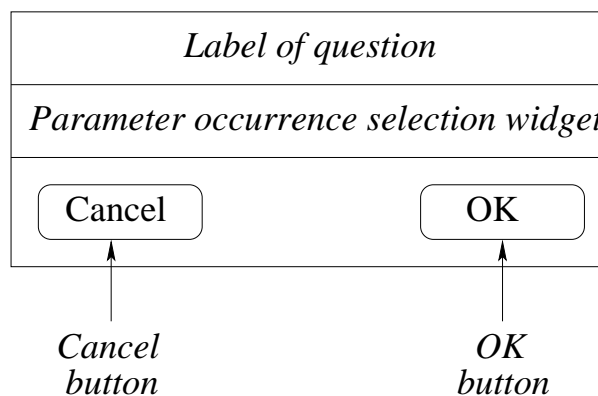


Figure 4.4: The Question Dialog Box of Star GUI

4.1.4 The Help Box

When the user selects *Help* under the *Star* menu, the Help Box is displayed. As shown in Figures 4.5, 4.6, 4.7, the Help Box has three states: the Top-Level state, the Command state, and the Parameter/Question state.

The Top-Level State of the Help Box shows the `OneLineHelp` for each command, as specified in the IDF². The Command State of the Help Box shows the detailed help information (or `MultiLineHelp`³) for the command selected in the Top-Level State. In addition, the Command State shows the `OneLineHelp` for each parameter and question of the command⁴. The Parameter/Question State of the Help Box shows the `MultiLineHelp` for the parameter or question selected in the Command State.

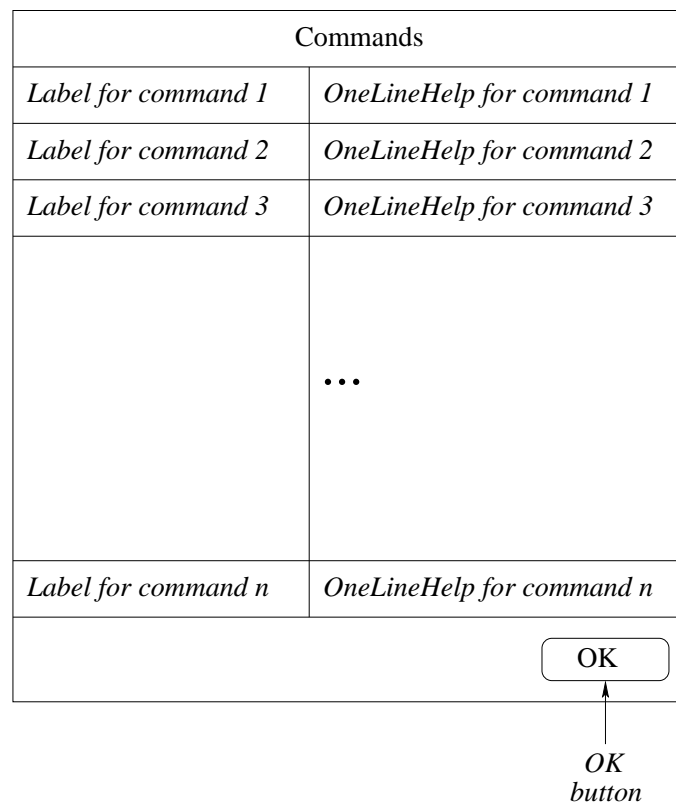


Figure 4.5: The Help Box of Star GUI [Top-Level State]

²The Label or `BriefHelp` of the command is displayed if there is no `OneLineHelp` specified for that command.

³The Label, `BriefHelp`, or `OneLineHelp` of the command is displayed if there is no `MultiLineHelp` for that command in the IDF.

⁴The Label or `BriefHelp` of the parameter/question is displayed if there is no `OneLineHelp` for that parameter/question in the IDF.

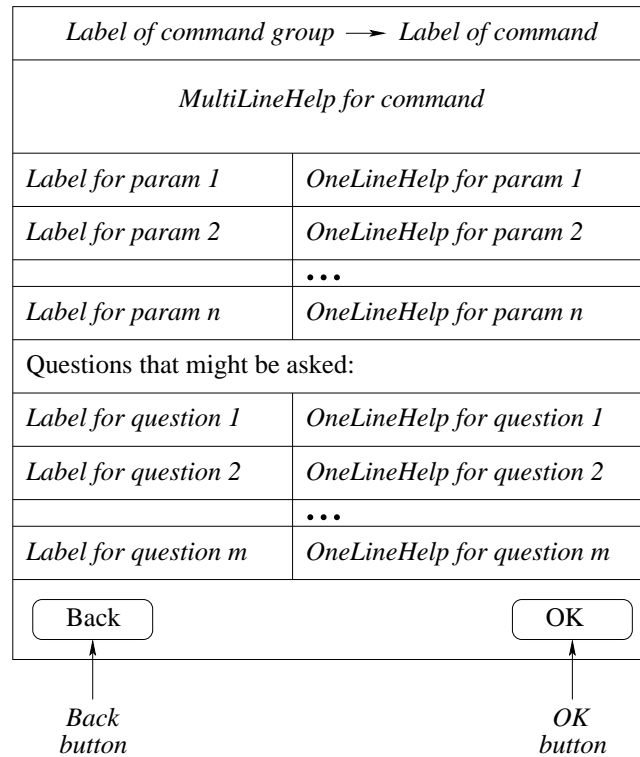


Figure 4.6: The Help Box of Star GUI [Command State]

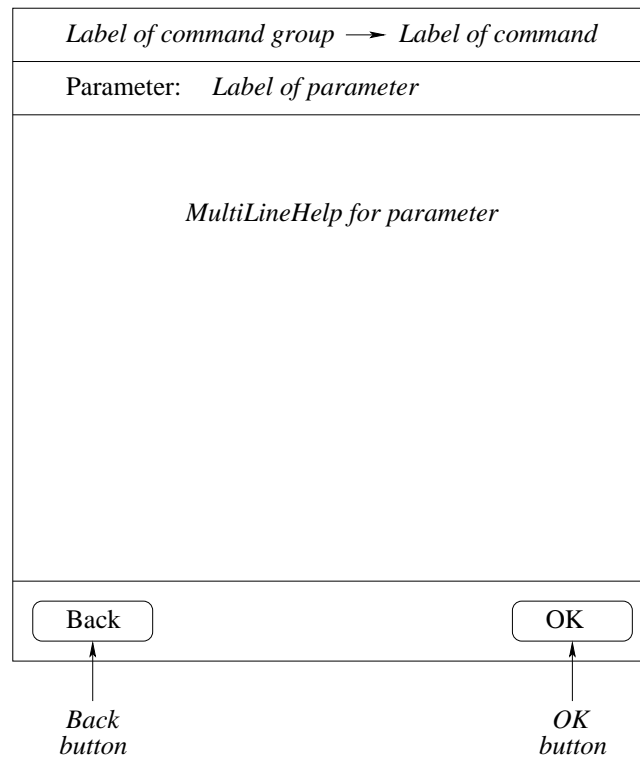


Figure 4.7: The Help Box of Star GUI [Parameter/Question State]

4.1.5 The Message Dialog Box

The Message Dialog Box, shown in Figure 4.8, is used to show high priority messages to the user. Such high priority messages can be error messages or other important information the user must know about.

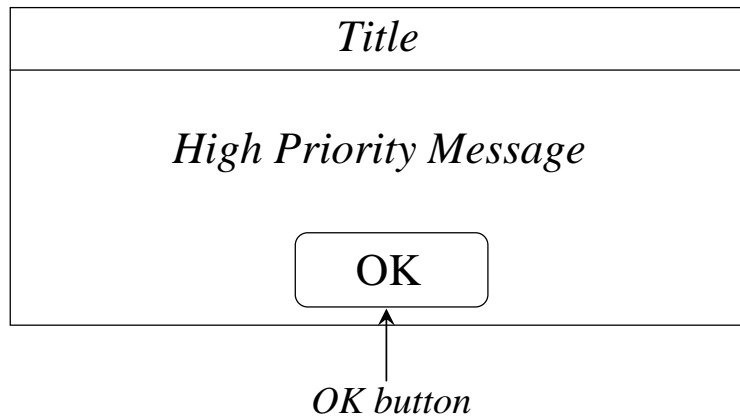


Figure 4.8: The Message Dialog Box of Star GUI

4.2 Design of Star

The Star interface interpreter is composed of many interrelated components or classes. In this section, we describe the key components of Star, including their respective functions and relationships.

4.2.1 Components of Star

The key components or classes of Star and the relationship among them are depicted in Figure 4.9.

Whenever the user launches an application (or app) through the Star interface interpreter, the *Star* class

- (1) reads the IDF written for the app;
- (2) creates the *GemSetting* object that provides access to the app engine;
- (3) uses the *GemSetting* object to validate all app engine methods specified in the IDF;
- (4) uses the *GemSetting* object to invoke the app engine's initialization method for initial computation or setup;

- (5) initializes all tables through the *GemSetting* object; and finally
- (6) loads the Main Panel of the GUI, if the previous tasks are successful.

The *StarWindow* class creates the Main Panel, which is composed of one or more Menus, a Text Display Area, a Status Bar, and a Table Area. Each Menu is composed of one or more menu items (or commands). The Table Area is composed of zero or more Table Widgets, each of which is populated with data through the *BrowsableTableModel* class.

Furthermore, the *CommandController* class receives, interprets, and responds to events generated by the user, as he/she interacts with the widgets. In the Star GUI, events are generated when the user clicks a menu item (or command), clicks a button (in the Command Dialog Box, Question Dialog Box, or Help Box), or selects row(s) in a table. For example, whenever the user clicks a menu item or command, a *CommandMenuItemClicked* event is raised, received, and interpreted by the *CommandController* class. It then responds by loading and displaying a Command Dialog Box for the selected command. Moreover, the *CommandController* class collects parameter values and question responses supplied by the user, validates them against data constraints specified in the IDF, and then sends them to the app engine via the *GemSetting* object.

The *CommandDialog* class creates the Command Dialog Box, which is composed of one or more stages, each of which is composed of one or more parameter widgets. The parameter widgets are used for supplying values for parameters. Depending on the type and the number of repetitions specified for each parameter in the IDF, each parameter widget is composed of zero or more of either Boolean Widget, Choice Widget, File Widget, Date Widget, Number Widget, Text Widget, Time Widget, or Table Entry Widget. As the name suggests, the Table Entry Widget is used for parameters of type *tableEntry*, but the specified source table must be non-browsable⁵. Thus, Table Entry Widgets display contents of non-browsable tables, which are populated with data via the *NonBrowsableTableModel* class.

Finally, the *QuestionDialog* class creates the Question Dialog Box, which is composed of only one parameter widget; the *HelpBox* class creates the Help Box; and the *MessageDialog* class creates the Message Dialog Box.

⁵The user cannot browse data nor select rows from a non-browsable table; hence, it is not visible in the Table Area.

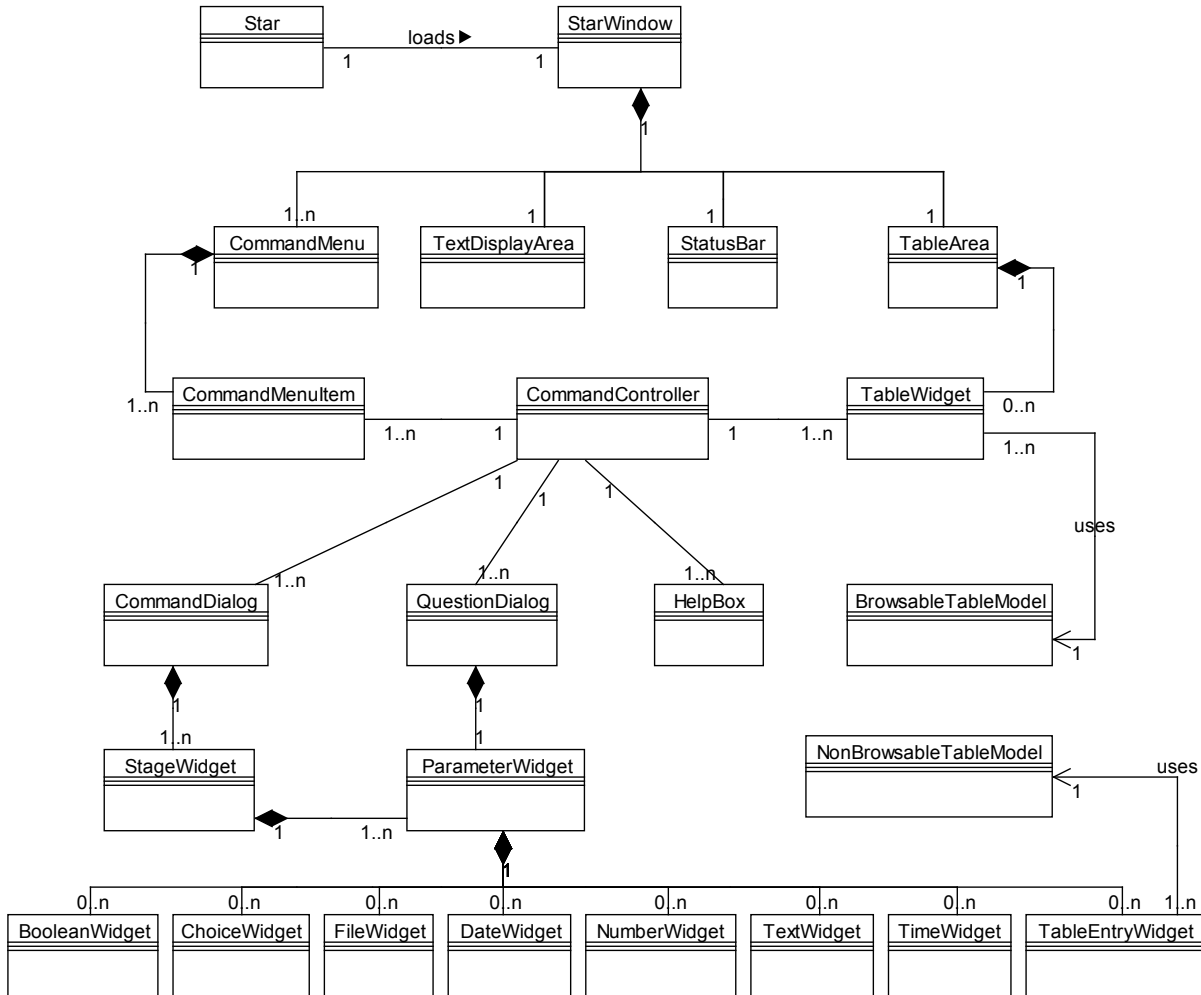


Figure 4.9: The Class Diagram showing the key components of Star and the relationship among them

4.3 Implementation and Demonstration of Star

The Star interface interpreter is implemented in Java. All the graphical interface elements, described in Sections 4.1 and 4.2 above, were created by extending appropriate classes in the Java Swing API [50], such as *JFrame*, *JDialog*, *JMenuBar*, *JMenu*, *JMenuItem*, *JTable*, *JTabbedPane*, *JSplitPane*, *JPanel*, *JScrollPane*, *JTextArea*, *JButton*, *JComboBox*, *JTextField*, *JFormattedTextField*, and several others. For example, the *StarWindow* class extends *JFrame* to create a window-like interface (having a title bar with minimize, maximize, and close buttons), both *CommandDialog* and *QuestionDialog* classes extend *JDialog* to create

application-modal dialog boxes⁶, the `TableWidget` extends `JTable` to create a table, and so on.

Furthermore, all the events generated during user-app interaction were created by implementing appropriate interfaces in the Java AWT Events API [51], such as `WindowListener`, `ActionListener`, `ComponentListener`, and `MouseMotionListener`. For example, the `StarWindow` class implements `ComponentListener` to generate events whenever the user changes the size of the Main Panel, both `CommandDialog` and `QuestionDialog` classes implement `WindowListener` to generate events whenever the user selects any widget in the dialog boxes, and so on. The functions of the Star components were implemented in accordance to the Star Behaviour Specification⁷. The source code of some of the Star components will be found in Appendix G of this thesis.

Having implemented Star, we proceeded to perform a live test by interacting with two apps using Star as the interface interpreter that suits our ability and needs. The two apps are *Temperature Converter* and *Appointment Calendar*.

Interacting with the Temperature Converter App

The Temperature Converter app helps in converting temperatures measured in Fahrenheit to Celsius, and vice-versa. The IDF, and its XML equivalent, for the app are shown in Figure 4.10 and Figure 4.11 respectively.

From the IDF, the app has three commands. The first command, `celsiusToFahrenheit`, converts temperatures in Celsius to Fahrenheit. It has only one stage, which contains a parameter for accepting temperature in Celsius. The second command, `fahrenheitToCelsius`, converts temperatures in Fahrenheit to Celsius. It also has only one stage, which contains a parameter for accepting temperature in Fahrenheit. Lastly, the third command, `exitApp`, is meant for closing the app when the user is done. However, it asks the user to confirm his or her intent to terminate the app by presenting an “are you sure” question. These three commands are members of a command group, called *convert*.

⁶When an application-modal dialog box is displayed, other parts of the user interface are frozen until the user finishes with the dialog box and closes it.

⁷Available in Appendix F of this thesis.

```

Application = TemperatureConverter
ApplicationEngine = TemperatureConverter
IdfVersion = "1.0"
InitializationMethod = initTemperatureConverter

CommandGroup convert = {
  Member = celsiusToFahrenheit
  Member = fahrenheitToCelsius
  Member = exitApp
}

Command celsiusToFahrenheit = {
  BriefHelp = "Converts Celsius to Fahrenheit"
  MultiLineHelp = "Use this command to convert temperatures measured in Celsius to
  temperatures measured in Fahrenheit."
  Stage conversion = {
    Parameter celsius = {
      Type = float
      Label = "Temperature in Celsius"
      OneLineHelp = "Supply your temperature in Celsius (not Fahrenheit)"
    }
  }
  CommandMethod = celsiusToFahrenheit
}

Command fahrenheitToCelsius = {
  BriefHelp = "Converts Fahrenheit to Celsius"
  MultiLineHelp = "Use this command to convert temperatures measured in Fahrenheit
  to temperatures measured in Celsius."
  Parameter fahrenheit = {
    Type = float
    Label = "Temperature in Fahrenheit"
    OneLineHelp = "Supply your temperature in Fahrenheit (not Celsius)"
  }
}

Command exitApp = {
  Label = "Exit"
  OneLineHelp = "Exits the Temperature Converter App"
  Question confirmExit = {
    Type = boolean
    Label = "Are you sure you want to exit?"
    OneLineHelp = "Confirms your intent to exit the app."
    AskIfMethod = confirmAppExit
  }
  QuitAfterIfMethod = appShouldQuit
}

```

Figure 4.10: IDF for the Temperature Converter App

```

<Johar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Application>TemperatureConverter</Application>
  <ApplicationEngine>TemperatureConverter</ApplicationEngine>
  <IdfVersion>1.0</IdfVersion>
  <InitializationMethod>initTemperatureConverter</InitializationMethod>
  <CommandGroup name="convert">
    <Member>celsiusToFahrenheit</Member>
    <Member>fahrenheitToCelsius</Member>
    <Member>exitApp</Member>
  </CommandGroup>
  <Command name="celsiusToFahrenheit">
    <BriefHelp>Converts Celsius to Fahrenheit</BriefHelp>
    <MultiLineHelp>Use this command to convert temperatures measured in Celsius
      to temperatures measured in Fahrenheit.</MultiLineHelp>
    <Stage name="conversion">
      <Parameter name="celsius">
        <Type>float</Type>
        <Label>Temperature in Celsius</Label>
        <OneLineHelp>Supply your temperature in Celsius (not
          Fahrenheit)</OneLineHelp>
      </Parameter>
    </Stage>
    <CommandMethod>celsiusToFahrenheit</CommandMethod>
  </Command>
  <Command name="fahrenheitToCelsius">
    <BriefHelp>Converts Fahrenheit to Celsius</BriefHelp>
    <MultiLineHelp>Use this command to convert temperatures measured in
      Fahrenheit to temperatures measured in Celsius.</MultiLineHelp>
    <Parameter name="fahrenheit">
      <Type>float</Type>
      <Label>Temperature in Fahrenheit</Label>
      <OneLineHelp>Supply your temperature in Fahrenheit (not Celsius)
    </OneLineHelp>
    </Parameter>
  </Command>
  <Command name="exitApp">
    <Label>Exit</Label>
    <OneLineHelp>Exits the Temperature Converter App</OneLineHelp>
    <Question name="confirmExit">
      <Type>boolean</Type>
      <Label>Are you sure you want to exit?</Label>
      <OneLineHelp>Confirms your intent to exit the app.</OneLineHelp>
      <AskIfMethod>confirmAppExit</AskIfMethod>
    </Question>
    <QuitAfterIfMethod>appShouldQuit</QuitAfterIfMethod>
  </Command>
</Johar>

```

Figure 4.11: The XML equivalent of the Temperature Converter App's IDF

The app engine that does the actual computation is the *TemperatureConverter* class⁸, which has six methods that can be invoked by Johar through its *GemFullImplementation* class residing in the *johar.gem* package. The six methods are the *initTemperatureConverter* (which is invoked when the app engine is initialized), *celsiusToFahrenheit* (which computes the Fahrenheit equivalent of the Celsius temperature), *fahrenheitToCelsius* (which computes

⁸Shown in Appendix H of this thesis.

the Celsius equivalent of the Fahrenheit temperature), *exitApp* (which performs some computation prior to terminating the app), *confirmAppExit* (which determines whether to ask the user the “are you sure” question), and *appShouldQuit* (which returns the user’s response to the “are you sure” question).

When the app is launched through Star, the Main Panel in Figure 4.12 is displayed. There are two menus on the Menu Bar: the *Convert* menu and the special *Star* menu. The Convert menu corresponds to the command group “convert” in the IDF. It has three menu items (as shown in Figure 4.13), which correspond to the three member commands in the command group.

The *Star* menu, whose menu item(s) represents service(s) provided by the Star interface interpreter, has the *Help* menu item (as shown in Figure 4.14). Furthermore, the Text Display Area contains a welcome message sent from the app engine when initialized. The Status Bar displays a “Ready” message, which is also sent by the app engine, and signifies the readiness of the app engine to perform computations. The Table Area is empty since there are no browsable tables in the IDF.

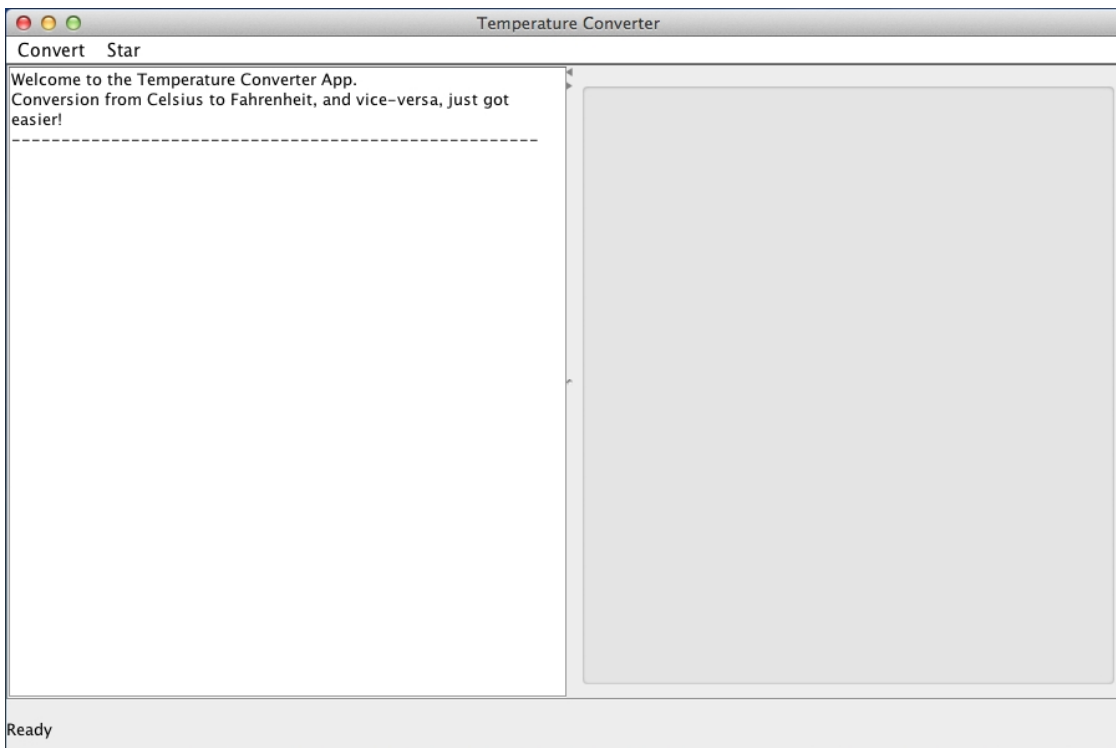


Figure 4.12: Main Panel of the Temperature Converter App



Figure 4.13: Main Panel of the Temperature Converter App [The *Convert* Menu]



Figure 4.14: Main Panel of the Temperature Converter App [The *Star* Menu]

When the “Celsius to Fahrenheit” menu item or command is selected, the Command Dialog Box in Figure 4.15 is displayed. The Command Dialog Box has a parameter section that contains a Number Widget, which is the parameter widget suitable for accepting a floating-point temperature in Celsius. This parameter section corresponds to the “celsius” parameter (in the IDF) whose type is `float` and labelled “Temperature in Celsius”. As shown in Figure 4.15, we entered “13” as the temperature in Celsius, and then clicked the *OK* button to trigger the app engine to begin computation. The result of the computation was sent by the app engine and displayed in the Text Display Area, as shown in Figure 4.16.

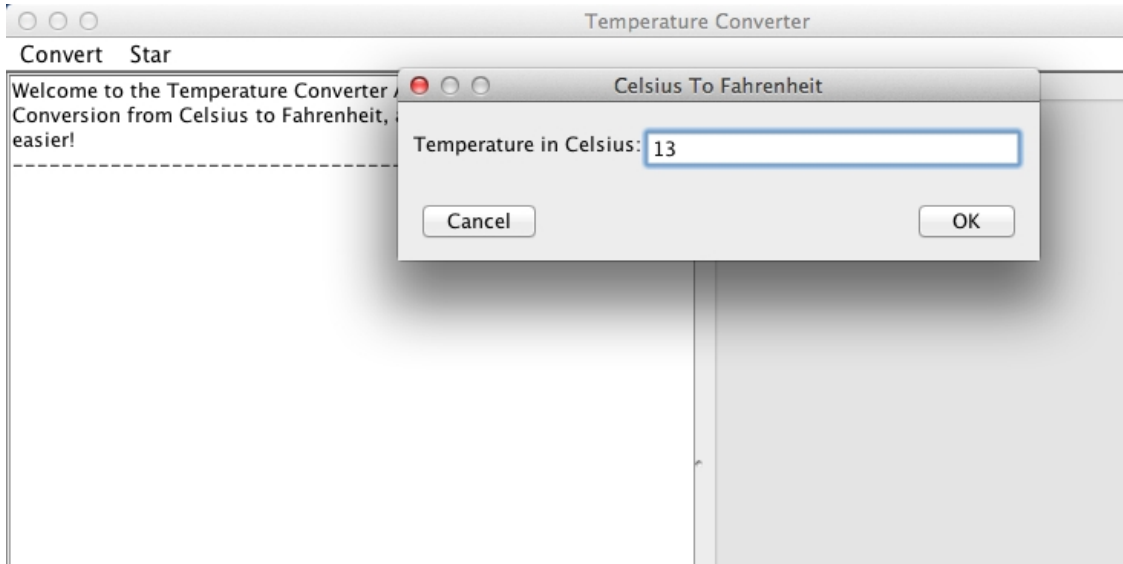


Figure 4.15: The Command Dialog Box for the “Celsius to Fahrenheit” Command

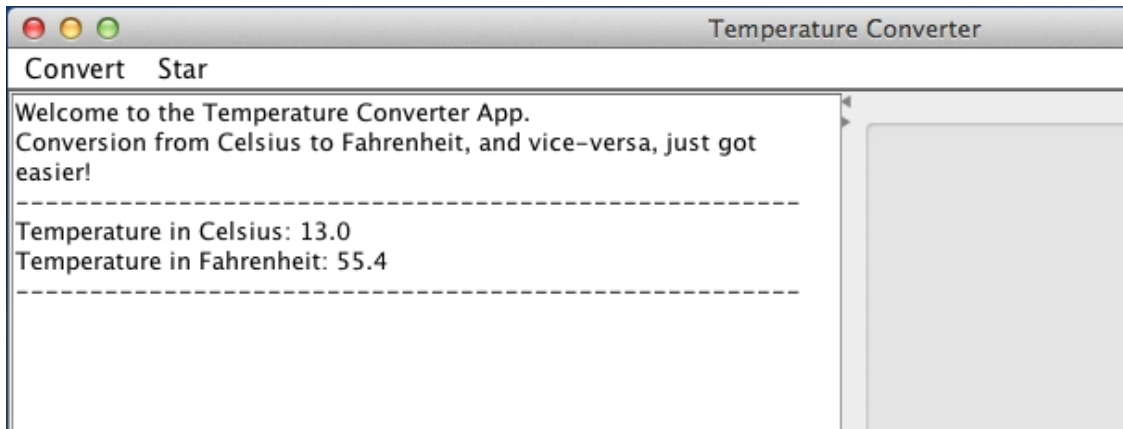


Figure 4.16: Text Display Area shows the conversion result

When *Help* is selected under the *Star* menu, the Top-Level State of the Help Box in Figure 4.17 is displayed. The Command State in Figure 4.18 is displayed when the “Celsius to Fahrenheit” command label is selected from the Top Level State. The Parameter State in Figure 4.19 is displayed when the “Temperature in Celsius” parameter label is selected from the Command State.

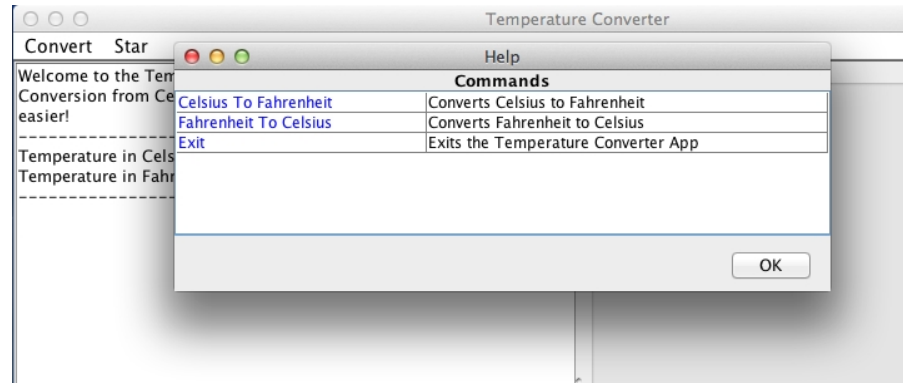


Figure 4.17: Top-Level State of the Temperature Converter App's Help Box

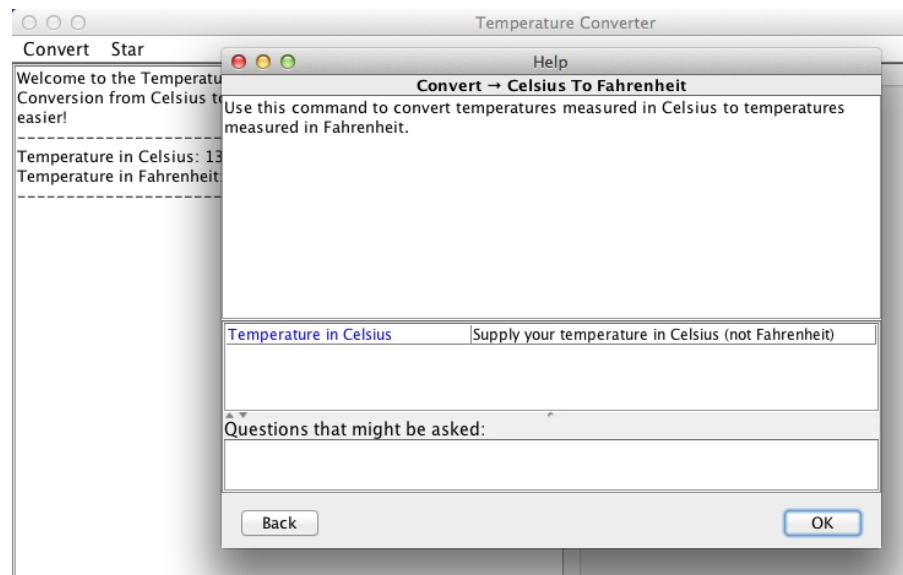


Figure 4.18: Command State of the Temperature Converter App's Help Box



Figure 4.19: Parameter State of the Temperature Converter App's Help Box

Finally, when the “Exit” command is selected under the Convert menu, the “are you sure” question in Figure 4.20 is displayed. The “Exit” command corresponds to the *exitApp* command (in the IDF) which contains the *confirmExit* question of type boolean. To terminate the app, the “Yes” option should be selected; and “No” if the app should stay on. In Figure 4.20, the “Yes” option is selected, and the app terminates when the “OK” button is clicked.

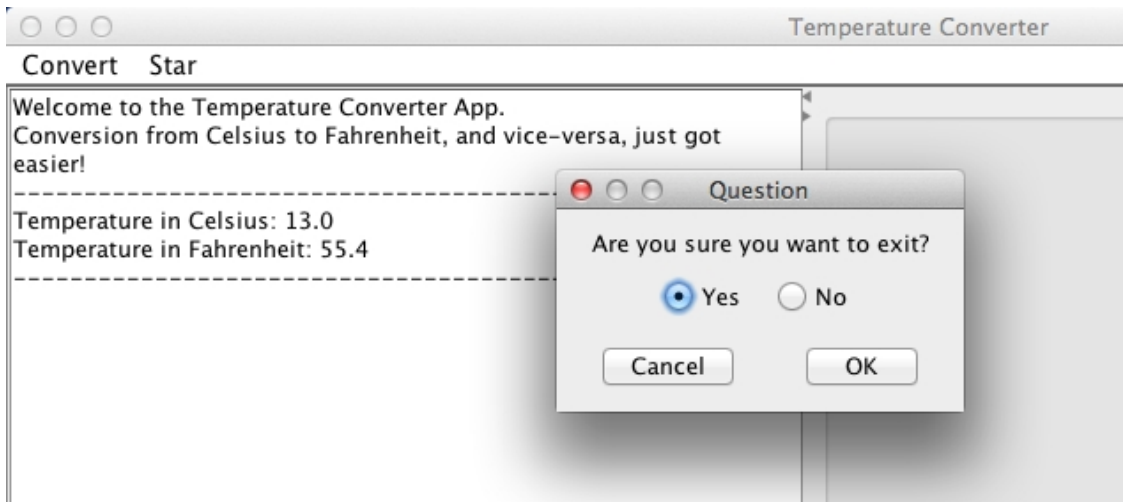


Figure 4.20: The Question Dialog Box confirming user’s intent to exit the App

Interacting with the Appointment Calendar App

The Appointment Calendar app is nicknamed “Ides of Johar”. Its IDF is shown in Appendix I of this thesis. From the IDF, the app has three browsable tables: the “weeks” table whose rows represent weeks in a given month, the “days” table whose rows represent days in a given week and the number of appointments in each day, and the “appointments” table whose rows represent appointments in a given day.

Moreover, the app has four command groups: *appointment* whose members are commands for adding and cancelling appointments, as well as for exiting the app; *previous* whose members are commands for navigating to the previous month, week, and day; *next* whose members are commands for navigating to the next month, week, and day; and *goTo* whose members are commands for moving to a particular week, day, or date.

The app includes commands to *add* and *cancel* appointments, and to navigate to specific dates. The parameters to the “add appointment” command include the time of the appointment, a description of the appointment, and data about how frequently and for how long the appointment should be repeated in the following days, weeks, or months.

When the app is launched via the Star interface interpreter, the Main Panel in Figure 4.21 is displayed. The Table Area shows the three browsable tables, with the “weeks” table initially populated with weeks in the current month. The Text Display Area shows a welcome message from the app engine. The first four menus in the Menu Bar correspond to the four command groups in the IDF, while the last menu is the usual *Star* menu.

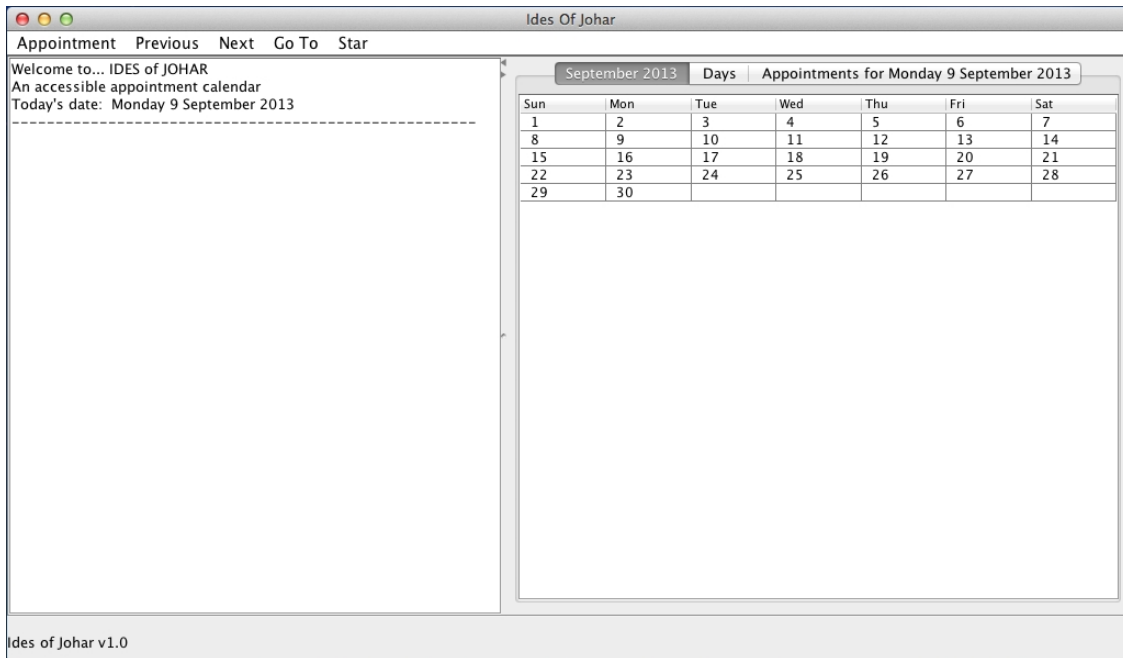


Figure 4.21: Main Panel of the Appointment Calendar App

To add an appointment to the current day, the “Add Appointment” command under the Appointment menu is selected, as shown in Figure 4.22. Afterwards, the Command Dialog Box for adding an appointment is displayed. As shown in Figure 4.23, the appointment time is set to 9:30am of the current day, and should be repeated once in each of the two following weeks.

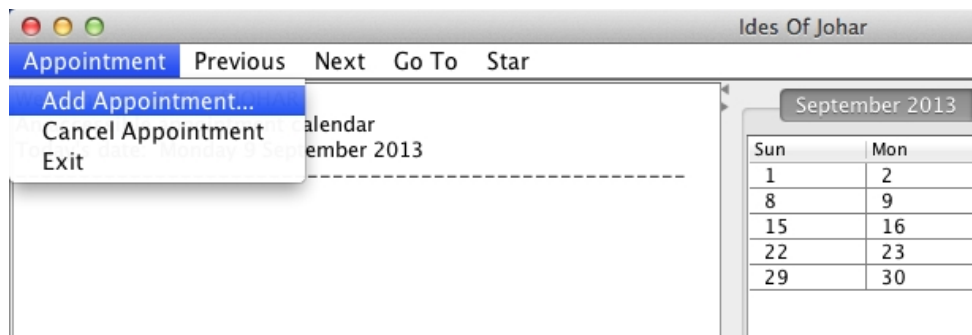


Figure 4.22: Main Panel of the Appointment Calendar App [The *Appointment* Menu]

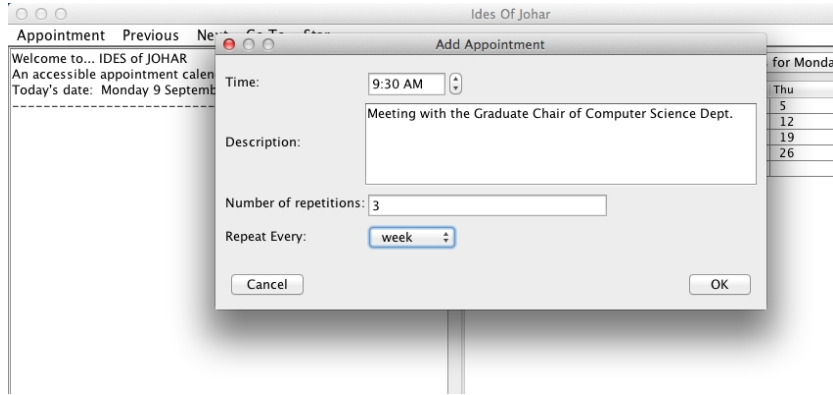


Figure 4.23: Command Dialog Box for the “Add Appointment” Command

After adding the appointment, a notification is displayed in the Text Display Area (as shown in Figure 4.24) and the weeks table indicates the existence of three appointments (as shown in Figure 4.25): the current day’s appointment, a repeated version in the following week, and another repeated version in two weeks’ time.

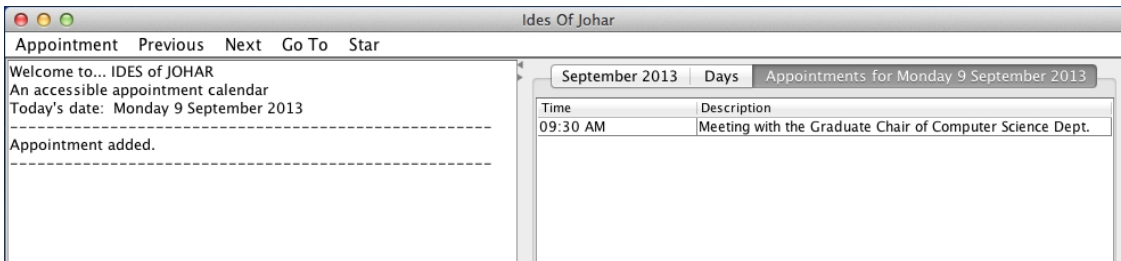


Figure 4.24: A notification and a new appointment are shown in the Text Display Area and Appointment table respectively

Ides Of Johar						
September 2013		Days	Appointments for Monday 9 September 2013			
Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	2	3	4	5	6	7
8	*9*	10	11	12	13	14
15	*16*	17	18	19	20	21
22	*23*	24	25	26	27	28
29	30					

Figure 4.25: Weeks table indicates the existence of three appointments

Moreover, to add or view appointments for a specific date, the *Previous*, *Next*, and *Go To* menus can be used to set the date. For example, to set the appointment date to “January 12,

2014”, the *Go To Date* command under the *Go To* menu can be used. This is demonstrated in Figures 4.26 and 4.27.

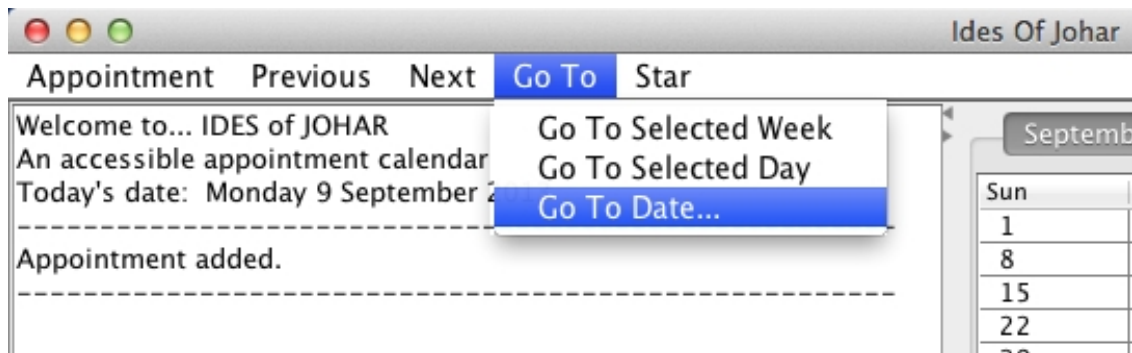


Figure 4.26: Selecting the “Go To Date” Command

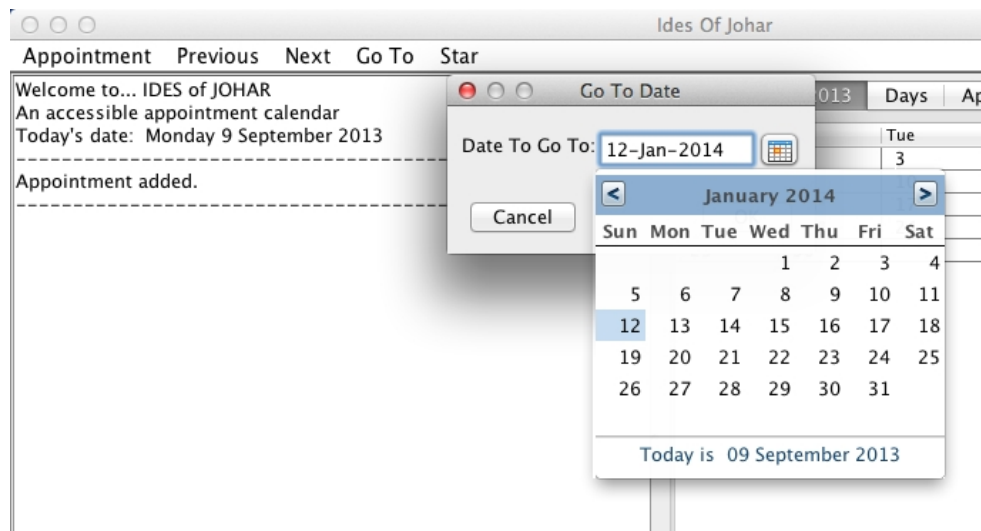


Figure 4.27: The Command Dialog Box for the “Go To Date” Command

Furthermore, to cancel an appointment, the appointment must first be selected from the Appointment table before clicking the “Cancel Appointment” command under the Appointment menu. This is demonstrated in Figures 4.28, 4.29, and 4.30.

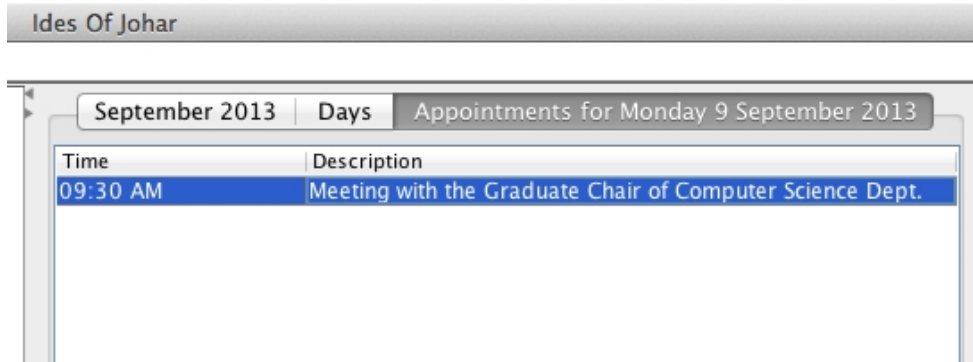


Figure 4.28: Selecting an appointment from the Appointment table

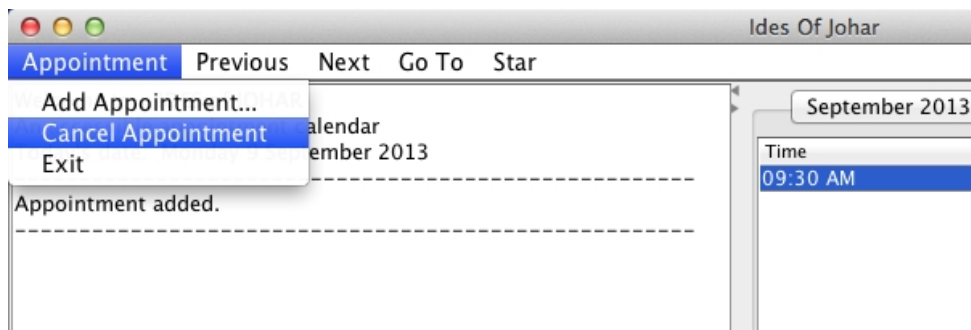


Figure 4.29: Cancelling the selected appointment via the “Cancel Appointment” Command

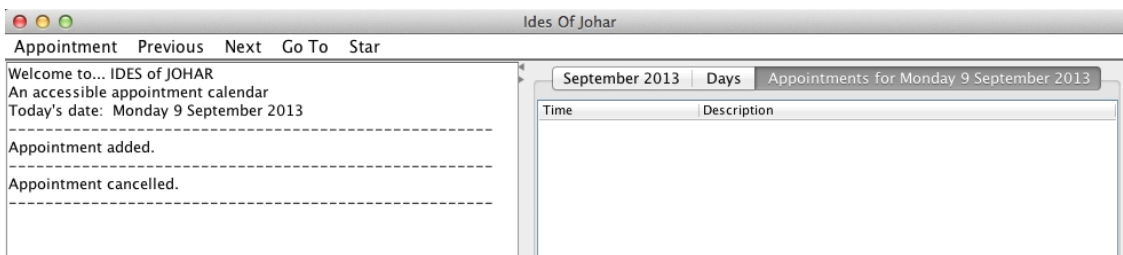


Figure 4.30: Cancellation notification in the Text Display Area and deletion of appointment from the Appointment table

When *Help* is selected under the *Star* menu, the Top-Level State of the Help Box in Figure 4.31 is displayed. To view detailed help information, the label of the desired command should be selected from the help table.

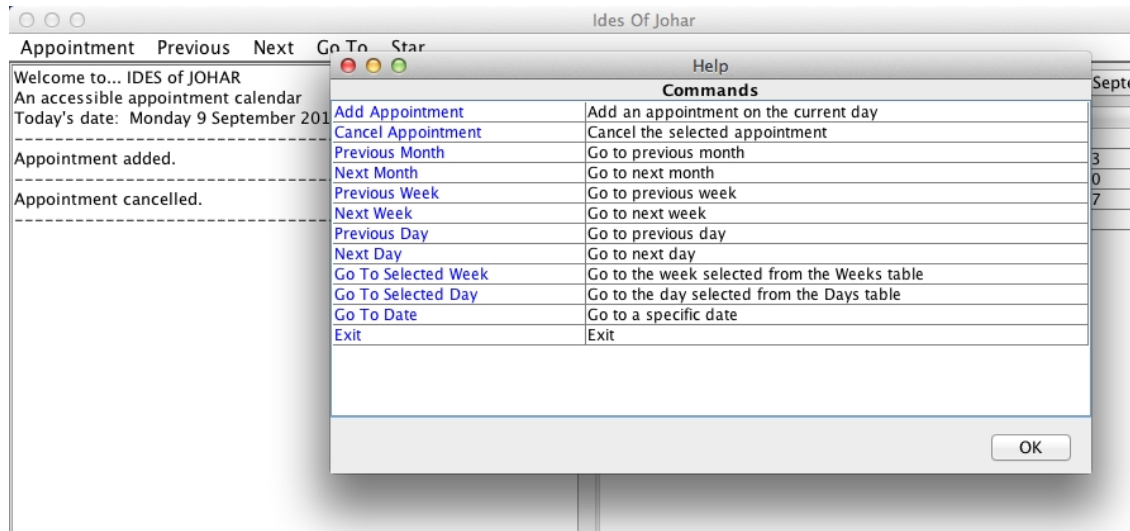


Figure 4.31: Top-Level State of the Appointment Calendar App's Help Box

Finally, to exit the app, the “Exit” command under the Appointment menu is selected, as shown in Figure 4.32. A Message Dialog Box, shown in Figure 4.33, providing a notification about the exit is then displayed and acknowledged (by clicking the *OK* button). Afterwards, the app terminates.

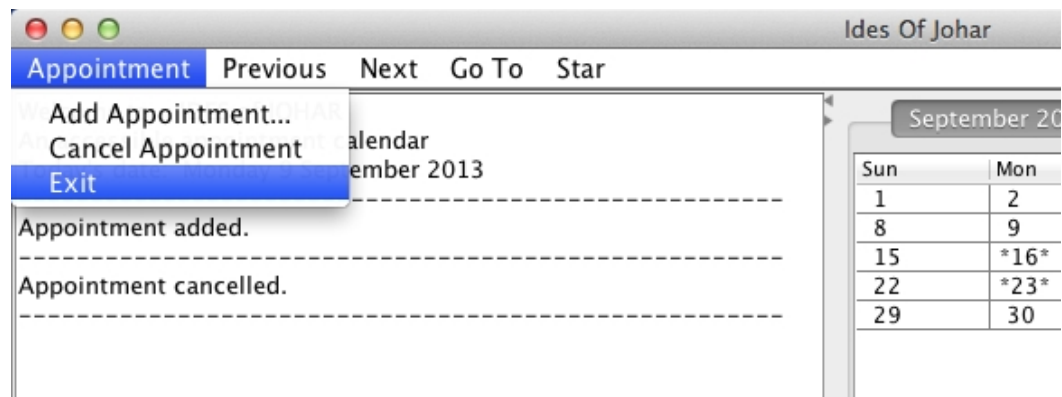


Figure 4.32: Selecting the Exit Command to terminate the Appointment Calendar App

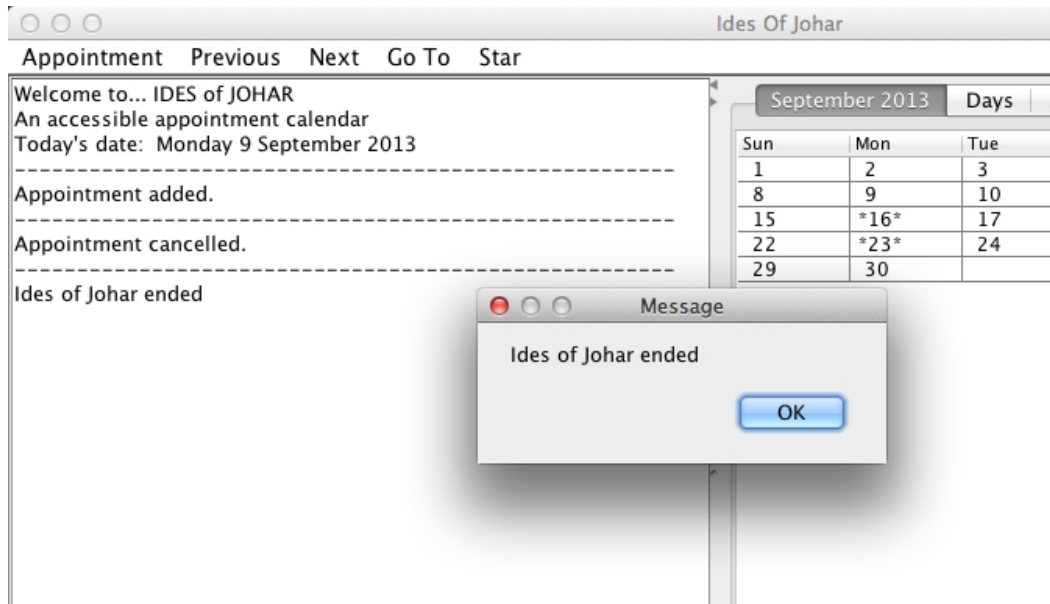


Figure 4.33: A Message Dialog Box notifying the user of a successful termination of the App

4.4 Quality Assurance on Star

Our quality assurance goals are:

- (1) To ensure the behaviour of Star is in agreement with our specification for interface interpreters;
- (2) To ascertain that Star works as expected on apps.

In order to achieve the first goal, we reviewed the Star GUI Specification document and the Star Behaviour Specification document in conjunction with the Interface Interpreter Specification document for the purpose of detecting and correcting inconsistencies, ambiguities, and missing requirements. We took note of each statement in the two Star specification documents that negates some requirement in the Interface Interpreter Specification document. In addition, we recorded each behavioural requirement of Star that is not captured in the Interface Interpreter Specification document, and vice-versa.

The review process described in the previous paragraph was conducted prior to the development of Star. The outcome of this review process revealed some issues, as documented in Table 4.1.

Document with Issue	Section with Issue	Description of Issue
Interface Interpreter Specification	Core Steps (Initialization Phase)	There is no information concerning refreshing or initializing tables in this document. However, this information is available in the Star Behaviour Specification document.
Star GUI Specification	1	Wrong method found in point 6: <code>displayText</code> . As stated in the Interface Interpreter Specification document, the correct method is <code>showText</code> .
Star GUI Specification	2	Clarification required in point 15(b): the definition of an incomplete stage is confusing or ambiguous.
Star GUI Specification	4	Incorrect parameter type found in point 1: <code>string</code> used instead of <code>text</code> .

Table 4.1: Outcome of reviewing Star Specification documents in conjunction with the Interface Interpreter Specification document

We fixed the first issue by including the missing information about the need for interface interpreters to initialize tables via *GemSetting*. We further corrected the wrong method and parameter type in the second and fourth issues respectively. For the third issue, we included additional sentences explaining (in clear terms) what constitutes an incomplete stage of a command. These additional and non-technical sentences help resolve the ambiguity that may have resulted in a faulty implementation if not addressed.

Finally, to achieve the second quality assurance goal, we performed a live test on three apps, two of which were presented in Section 4.3 above. The third is the *Meal Planner* app. With this app, a user can specify the food items to eat for breakfast, lunch, and dinner, as well as the date, start time, and duration of each meal. The user can save the meal information for future retrieval if necessary.

Chapter 5

Other Interface Interpreters

In this chapter, we discuss four previous interface interpreters that were built on top of the 2009 version of Johar, as well as two interface interpreters (besides Star) which ride on the new version of Johar.

5.1 Previous Interface Interpreters

Prior to our quality assurance on Johar that led to its redesign, four interface interpreters - *Speak*, *Press*, *Senatus*, and *Show* - were already in existence. The “Speak” interface interpreter was developed for blind users who rely on keyboard and sound as means of interacting with apps. *Speak* accepts input from users via the keyboard, and displays and speaks the output to users. The “Press” interface interpreter was developed for users with severe motor impairments. It elicits data by cyclically displaying and highlighting choices with a user-controllable timer delay between the choices, and allowing the user to make a selection by pressing a specific key when the intended choice has been highlighted. The “Senatus” interface interpreter presents a Unix-like command-line interface to the user. Users interact with apps through the command-line interface by entering commands that match their intended tasks. Finally, the “Show” interface interpreter presents a graphical user interface (GUI) through which users interact with apps.

However, despite the potential benefits derivable from these four interface interpreters, they are not without shortcomings. For instance, *Speak* does not work well with tables, *Senatus* has issue with parsing dates and times, *Show* does not display tables at launch time and also does not support multi-stage commands, and none of the four interface interpreters work well with questions. Moreover, their smooth operation was hindered by faults in the old version of Johar on which they were built.

Fortunately, our quality assurance on Johar, discussed in Chapter 3, has led to a new and

more reliable version of Johar which in turn facilitates the smooth operation of interface interpreters, as well as ensuring effective collaboration between interface interpreters and apps. We have already designed and implemented a classic WIMP GUI interface interpreter (called *Star*, presented in Chapter 4 of this thesis) which is based on this new version of Johar. We also designed two other interface interpreters - *Grupo* and *StarX* - which are discussed in the subsequent sections. The implementation of these two interface interpreters will be accomplished in the near future.

We believe that the implementation of *StarX* and *Grupo*, together with the quality assurance activities of this thesis, will make the more complex process of designing important interface interpreters for blind, low-vision and motor-impaired users easier.

5.2 The StarX Interface Interpreter

The keyboard has been a major input device for users with restricted hand use, who find it difficult to accurately position a mouse pointer on screen objects [52]. Moreover, frequent mouse usage has been linked to discomfort and pain around the wrist, along the forearm and elbow, and on top of the hand [53]. This is as a result of frequent positioning, scrolling, and clicking with the mouse [53]. Furthermore, a good keyboard implementation has been known to improve user productivity [52].

Thus, in order to make the *Star* interface interpreter (discussed in Chapter 4) accessible to keyboard-bound users, we decided to enhance its GUI to support keyboard shortcuts. This decision led to the design of *StarX*, which is an eXtended version of *Star* that facilitates keyboard-only interactions. Thus, if a user prefers to use only the keyboard (without the mouse) to interact with interface components, *StarX* will fully support the user by activating the keyboard shortcuts defined in the *StarX* GUI Specification document (available in Appendix J of this thesis). *StarX* offers users the flexibility of enabling or disabling the keyboard shortcuts, when necessary. Thus, *StarX* becomes beneficial to all users - both keyboard and mouse users alike.

5.2.1 The StarX GUI

The *StarX* GUI is almost the same as the *Star* GUI described in Chapter 4. However, the *StarX* GUI has been enhanced to respond to keystrokes and to provide visual cues as each part of the interface receives focus.

The Main Panel of *StarX* GUI consists of the Menu Bar, a Text Display Area, a Table Area, and the Status Bar. The Menu Bar consists of application menus and a special menu,

called *StarX*, whose menu items represent services provided by the *StarX* interface interpreter for all Johar apps. The *StarX* menu contains three menu items: *Help*, *Enable Keyboard-only Interaction*, and *Show Hotkeys Pop-up Table*. The *Help* menu item displays the Help Box when selected by the user. The *Enable Keyboard-only Interaction* menu item activates keyboard shortcuts and disables mouse effects (e.g. pointer movement, clicking, scrolling, and dragging), when selected¹. To deactivate keyboard shortcuts and enable mouse effects, the *Disable Keyboard-only Interaction* menu item should be selected by the user.

Furthermore, when keyboard shortcuts have been activated, the user might require the service of a pop-up table which reminds him/her of the shortcuts applicable to the focused part of the interface. This table, called “Hotkeys Pop-up Table”, can be activated by selecting *Show Hotkeys Pop-up Table* under the *StarX* menu² or by pressing the assigned shortcut key. To hide the pop-up table, the user can press the assigned shortcut key or select the *Hide Hotkeys Pop-up Table* menu item under the *StarX* menu. Details about the Hotkeys Pop-up Table will be found in one of the subsections below.

Visual Cue for Focused Interface Widgets

As the user interacts with the user interface using keyboard shortcuts, it is imperative for *StarX* to provide a visual feedback so that the user can know which widget currently has focus. This visual feedback or cue generally helps the user to easily determine where they are in the user interface, even in complex situations.

StarX provides visual feedback by surrounding every focused widget with a partly thick red border. The red border will quickly attract user’s attention to the widget that currently has focus. For example, in Figure 5.1, a red border surrounds the first parameter repetition widget which currently has focus.

<i>Label of parameter followed by colon (:)</i>	<i>1st Repetition Widget</i> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
	<i>2nd Repetition Widget</i> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
	<i>3rd Repetition Widget</i> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
	<i>4th Repetition Widget</i> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
	<input type="checkbox"/>

Figure 5.1: A red border acting as a visual cue for a widget that currently has focus

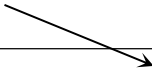
¹The word “Enable” in the *Enable Keyboard-only Interaction* menu item changes to “Disable”, and vice-versa, when selected by the user.

²The word “Show” in the *Show Hotkeys Pop-up Table* menu item changes to “Hide”, and vice-versa, when selected by the user.

The Hotkeys Pop-up Table

This table, shown in Figure 5.2, informs the user about the keyboard shortcuts or hotkeys that are applicable to certain sections or parts of the user interface. Thus, whenever a section receives focus (such as focusing on the Table Area) or a new window is launched (such as launching a Command Dialog Box) in the user interface, the table automatically displays the shortcuts that are applicable to that section or window. For example, if the Table Area of the Main Panel receives focus, the table shows all the shortcuts (and their functions) that are applicable to the Table Area. In addition, the content of the Hotkeys table changes as the user moves from one widget to another within the active section or window. For instance, as the user selects a table widget in the Table Area, the content of the Hotkeys table changes to reflect the keyboard shortcuts applicable to the selected table widget (such as the shortcuts for selecting and deselecting rows). This dynamic pop-up table, therefore, helps the user in effectively navigating different parts of the user interface.

A name identifying the focused part of the user interface is displayed here (e.g. Table Area, Command Dialog, Parameter Repetition Section, Help Box Command State, Text Display Area, etc.)



Active Keyboard Shortcuts for <i><interface section></i>	
<i>Shortcut 1</i>	<i>Function of Shortcut 1</i>
<i>Shortcut 2</i>	<i>Function of Shortcut 2</i>
<i>Shortcut 3</i>	<i>Function of Shortcut 3</i>
.	.
.	.
.	.
<i>Shortcut n</i>	<i>Function of Shortcut n</i>

Figure 5.2: The Hotkeys Pop-up Table

5.2.2 Rationale for the Choice of Keyboard Shortcuts

Our choice of keyboard shortcuts is guided by efficiency and easy memorization of keys. To achieve efficiency, we decided to assign single-key shortcuts to interface elements or widgets

to enable users perform their intended tasks quickly and with less effort. Single-key shortcuts will also be easier to memorize than two or three combinations of keys. We were able to implement this decision or objective to a large extent, as evident from the StarX GUI Specification document in Appendix J; however, we faced two major challenges which are described below.

- Most GUI toolkits, including Java Swing, do not support single-key shortcuts for menus. Instead, they require the *Alt* key to precede any single-key (or mnemonic) assigned to a menu. For example, *Alt+F* selects the File menu, *Alt+E* selects the Edit menu, etc. on many Windows applications (e.g. Text Editors).
- Due to the presence of data entry widgets (such as widgets for entering/selecting textual, numeric, date, and time values) in some interface containers (such as Command Dialog Box, Help Box, and Question Box), using single-key shortcuts for certain actions (such as pressing a button, or transferring focus among data entry widgets) becomes infeasible. For instance, when a text widget (e.g. text box) receives focus, pressing keys like H, Y, S, L causes the characters to appear inside the widget instead of clicking a button or transferring focus to another data entry widget.

We resolved the first challenge by using the *Alt* key in conjunction with the *M* key to select the first menu, and then allowing traversal from one menu to another via the *left* and *right arrow* keys or *Z* and *X* keys. We resolved the second challenge by using the *Ctrl* key in conjunction with another key to click buttons, and using Function keys (e.g. F3, F5, etc.) to transfer focus from one data entry widget to another, as well as to display pop-ups (such as Calendar pop-up box, File Chooser Dialog box, and the Hotkeys pop-up table).

5.3 The Grupo Interface Interpreter

Grupo³ is a “batch mode” interface interpreter that accepts a text file, containing various commands, as input. These commands are peculiar to Grupo (just like Unix commands are peculiar to Unix-based operating systems), and they constitute the means by which users perform their intended tasks. Grupo commands covers table selection, browsing contents of tables, selecting rows from a table, deselecting selected table rows, accessing and launching app commands⁴, specifying parameters for app commands, and accessing help contents. Grupo reads each line of command from the input file, executes the command against the app

³Grupo means “batch”, “set”, or “group” in Spanish.

⁴App commands are defined in the IDF written by the app developer.

engine, and then writes output messages to the Standard Output (stdout)⁵. The syntax, semantics, and behaviour of Grupo commands are available in the Grupo Requirements and Behaviour Specification documents (which will be found in Appendix K and Appendix L of this thesis).

Grupo is a powerful tool for exercising the capabilities of apps, for debugging app engines, for performing regression testing on apps, and for detecting missing features or functionality in apps. In addition, since the input and output of Grupo are text-based, it could be used as the basis of a rudimentary Braille-based or speech-based interface interpreter.

In this section, we discuss how Grupo provides access to tables, app commands, parameters, and help contents. A sample input file, in Figure 5.3, containing Grupo commands for interacting with the Appointment Calendar app (described in Chapter 4) is used as an illustration. We wrap up this section by discussing how Grupo can be used in detecting faults in app engines.

5.3.1 Working with Tables

Grupo provides commands for setting the current table, selecting rows from the current table, deselecting rows in the current table, and displaying the content of a table.

Setting the Current Table

To set the current table in Grupo, the `table` command is used. The `table` command accepts the name of the desired table, which must be one of the tables specified in the IDF, as an argument. For example, to set the Appointment table as the current table, the following line was inserted in the sample input file (as shown in Figure 5.3):

```
table appointments
```

where `appointments` is the name assigned to the Appointment table in the IDF.

Selecting Rows from the Current Table

Having set the current table with the `table` command, rows of data can be selected through the `select` command. The `select` command accepts the desired row number as an argument. The row number is a number n , where $n \geq 1$. Thus, row number 1 represents the first row, 2 represents the second row, 3 represents the third row, and so on. Moreover, in order to select m rows from the current table (where $m \geq 1$), m `select` commands must be issued. For example,

⁵Output messages written to the Standard Output are displayed on the Terminal (in Unix operating systems) or Command Window (in Windows operating systems) by default, but can be redirected to a file if required.

```
//Make appointments the current table, select the second and fourth rows
table appointments
select 2
select 4

//Add a new appointment which is repeated once weekly for 3 weeks
command addAppointment
param time "9:30 am"
param description "Meeting with the Graduate Chair of CS Dept."
param numOfReps 3
param repeatEvery week
ok

//Deselect the fourth row of the appointments table
deselect 4

//Cancel the selected appointment (i.e. second appointment)
command cancel
ok

//Display the content of the appointments table
browse appointments

//Display the brief help of the cancel command
help -b cancel

//Terminate the application
command quit
ok
```

Figure 5.3: An input file containing Grupo commands for interacting with the Appointment Calendar App

to select the second and fourth appointments from the Appointment table, the following lines were inserted in the sample input file:

```
select 2
select 4
```

Deselecting Rows in the Current Table

Having selected rows of data from the current table using the `select` command, any selected row can be deselected through the `deselect` command. Similar to the `select` command, the `deselect` command accepts the desired row number as an argument. The row number is a number n , where $n \geq 1$. Moreover, in order to deselect m rows in the current table (where $m \geq 1$), m `deselect` commands must be issued. For example, to deselect the fourth appointment in the Appointment table, the following line was inserted in the sample input file:

```
deselect 4
```

Displaying the Content of a Table

The data in a table can be displayed on screen through the `browse` command provided by Grupo. The `browse` command accepts the name of the desired table as argument. This table must be declared as *browsable* in the IDF. For example, to display all the appointments in the Appointment table, the following line was inserted in the sample input file:

```
browse appointments
```

where `appointments` is the name assigned to the Appointment table in the IDF.

5.3.2 Accessing App Commands

The app commands, defined in the IDF, represent the app's features through which users carry out their intended tasks. The app commands may accept parameters, also defined in the IDF, as input. Each of these parameters requires value(s) of a specific type⁶ to be supplied by the user.

Grupo provides access to app commands through the “command” keyword, followed by the name of the app command (as specified in the IDF). For example, to add a new appointment, the following line was inserted in the sample input file:

```
command addAppointment
```

⁶The valid types are `int`, `float`, `boolean`, `choice`, `text`, `file`, `date`, `timeOfDay`, and `tableEntry`.

where `addAppointment` is the command for adding a new appointment, as specified in the IDF.

Specifying Parameters for App Commands

Having specified an app command in the input file, the parameters to that app command can then be supplied through the `param` command provided by Grupo. The `param` command accepts both the parameter name⁷ and parameter value as arguments. The `param` command must be placed on a new line immediately after the use of the app command in the input file. Parameter values containing blank spaces (such as texts) are enclosed in double quotation marks. For example, to specify the time and description of the new appointment to be added, the following lines were inserted in the sample input file:

```
param time "9:30 am"  
param description "Meeting with the Graduate Chair of CS Dept."
```

Executing App Commands

After specifying an app command and its parameters, a line containing the `ok` command must be placed immediately after the last `param` command (or after the app command if no `param` command is specified). When Grupo sees the `ok` command, it notifies the app engine (through Johar) to begin computation on the app command that immediately precedes the `ok` command. For example, in Figure 5.3, when Grupo reaches the first `ok` command while reading and interpreting the content of the input file line-by-line, it notifies the app engine to begin computation on the `addAppointment` command (which is the app command that immediately precedes the first `ok` command). Thus, the app engine, after computation, adds the specified appointment to the calendar.

5.3.3 Accessing Help Contents

Help contents defined for app commands and parameters in an IDF can be accessed by the user through the `help` command provided by Grupo. The `help` command provides access to the *BriefHelp*, *OneLineHelp*, and *MultiLineHelp* of each app command or parameter in the IDF through the `-b`, `-o`, and `-m` options respectively. Thus, the `help` command accepts an option (`-b`, `-o`, or `-m`) and the name of the app command or parameter as arguments. However, if an option is not specified in the `help` command, then the `-o` option is used by Grupo. For example,

⁷Specified in the IDF for the app command.

to display the *BriefHelp* of the “cancel” command (which is used for canceling appointments), the following line was inserted in the sample input file:

```
help -b cancel
```

5.3.4 Testing Apps Using Grupo

Since Grupo allows all the commands needed to interact with an app to be batched in a file, it becomes easy to generate test cases for the purpose of testing each functionality or feature of any app. Each test case is a file containing Grupo commands, such as the sample file shown in Figure 5.3, for testing one feature or group of features of an app. As shown in Figure 5.4, Grupo accepts a test case as an input, and then runs it against the app engine via Johar.

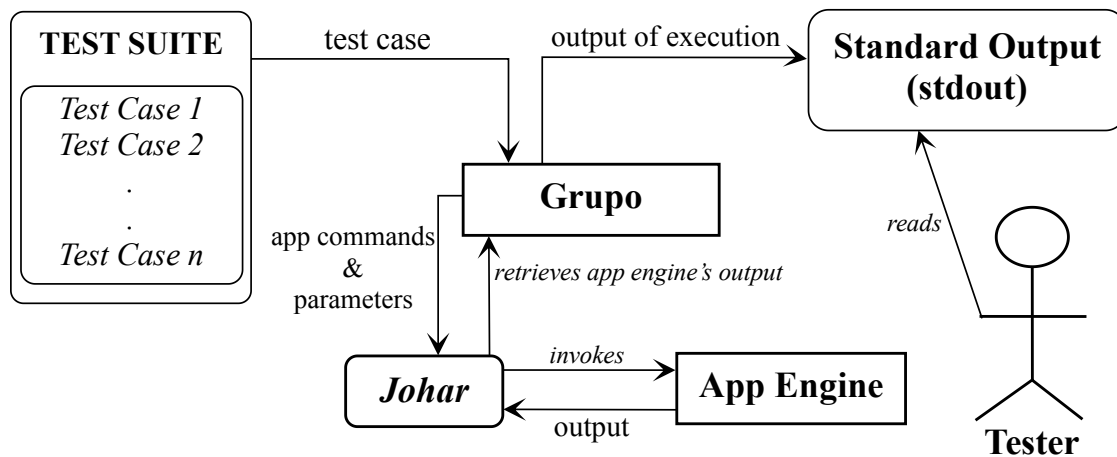


Figure 5.4: Grupo accepts a test case, executes it against the App Engine via Johar, and displays output information on Standard Output for a Tester’s perusal

Grupo displays the output of the app engine’s computation on the Standard Output, including other information that will assist the tester in making accurate decisions. Thus, there is need for the tester to be able to distinguish information displayed on the Standard Output. As a result, we group output information displayed by Grupo into six categories. As shown in Table 5.1, each category of information is identified by a three-letter prefix. For instance, an error message is easily identified via the ERR prefix e.g. *ERR: Integer value expected for parameter “age”*.

Prefix	Category of Information	Purpose
COM	Command Input	Precedes each app command’s input (i.e. parameter) displayed on the Standard Output.

OUT	App Engine's Output	Precedes every output of an app engine's computation displayed on the Standard Output.
TAB	Table Content	Precedes every table whose content is displayed on the Standard Output.
ERR	Error Message	Precedes every error message displayed on the Standard Output.
REM	Remark/Comment	Precedes every comment (starting with "//" in an input file) displayed on the Standard Output.
HLP	Help Content	Precedes every help information displayed on the Standard Output.

Table 5.1: Prefix for each category of information displayed on the Standard Output

The tester can then examine the error messages, app engine output, table contents, and other information displayed on the Standard Output for each test case in order to make critical decisions such as the exact location of bugs or faults in the app, positive/negative effect of an enhancement on the app, and missing or incomplete features in the app.

Chapter 6

Conclusion

Our overall goal is to assure the quality of Johar, which is a framework for developing apps that are accessible to both disabled and non-disabled users. We were able to accomplish this goal, having detected and resolved many inconsistencies, omissions, irrelevancies, and other anomalies that can trigger unexpected or abnormal behaviour in Johar, and/or alter the smooth operation of interface interpreters and apps.

Our approach involved reviewing the two Johar components or packages - `johar.gem` and `johar.idf` - by critically examining the functionality of classes in each component, including how classes interrelate and how functions are allocated or distributed among the classes. We discovered several shortcomings, which were resolved through series of redesign activities. Furthermore, we performed an exhaustive comparative review of four documents (i.e. IDF Format Specification document, XML Schema Document or XSD, the Interface Interpreter Specification document, and the `johar.idf` package) that are vital to the smooth running of all interface interpreters and apps, which in turn led to the detection and resolution of various discrepancies. We concluded the review process by performing automated tests on IDFs. The automated testing tool we developed for this purpose utilized the `johar.idf` package to transform each IDF (used as test cases) to an XML document, to validate the XML document (using the XML Schema Document), and to check for violation against several other constraints not captured in the XSD.

Having redesigned Johar through our quality assurance process, we proceeded to design and implement an interface interpreter (called `Star` which presents WIMP graphical user interfaces to users) in order to demonstrate Johar's ability to guarantee unhindered interaction between interface interpreters and apps. Prior to its implementation, we made sure that the design documents of `Star` are consistent with the specification document which must be satisfied by all interface interpreters (i.e. Interface Interpreter Specification document).

Finally, we designed two other interface interpreters - `Grupo` and `StarX` - whose implementation will be accomplished in the near future.

6.1 Future Work

In the near future, we would like to release our first version of Johar to the open source community. We want the released Johar bundle to initially include some interface interpreters suitable for non-disabled users (such as Star), blind users, motor-impaired users, and low-visioned users, as well as include a series of apps. Finally, we would establish interface interpreter and app *developers community*, as well as *users forum* that will serve as a platform on which Johar developers brainstorm, and on which app users share their views and experiences.

For this future work, the work on this thesis has laid a solid foundation. By assuring the consistency between the Johar design documents, the Star specification, and the implementation of Star, we have shown that Johar is an internally consistent framework that can act as the basis for complex interface interpreters and apps. By our other quality assurance activities, we have also lowered the number of design and implementation bugs, thus making the process of developing further interface interpreters and apps smoother and more trouble-free.

Bibliography

- [1] U.S Department of Justice. Information Technology and People with Disabilities: The Current State of Federal Accessibility. [Online]. Available: <http://www.justice.gov/crt/508/report/software.htm> (Accessed August 2013)
- [2] M. W. Brault. (2012, July) Americans With Disabilities: 2010 (Current Population Reports). [Online]. Available: <http://www.census.gov/prod/2012pubs/p70-131.pdf>
- [3] World Health Organization, “Draft action plan for the prevention of avoidable blindness and visual impairment 2014—2019. Towards universal eye health: a global action plan 2014—2019,” *World Health Organization Sixty-Sixth World Health Assembly*, no. A66/11, pp. 1–18, March 2013. [Online]. Available: http://apps.who.int/gb/ebwha/pdf_files/WHA66/A66_11-en.pdf
- [4] World Health Organization (WHO), *Global Initiative for the Elimination of Avoidable Blindness: action plan 2006–2011*. Geneva 27, Switzerland: WHO Press, 2007.
- [5] Freedom Scientific. JAWS. JAWS Headquarters. [Online]. Available: <http://www.freedomscientific.com/jaws-hq.asp> (Accessed July 2013)
- [6] NV Access. NVDA features. [Online]. Available: <http://www.nvaccess.org/about/nvda-features/> (Accessed July 2013)
- [7] GW Micro, Inc. GW Micro - Window-Eyes. [Online]. Available: <http://www.gwmicro.com/Window-Eyes/> (Accessed July 2013)
- [8] Freedom Scientific. MAGic Screen Magnification Software with Speech. [Online]. Available: <http://www.freedomscientific.com/products/low-vision/MAGic-screen-magnification-software.asp> (Accessed July 2013)
- [9] M. Porta, A. Ravarelli, and G. Spagnoli, “ceCursor, a contextual eye cursor for general pointing in windows environments,” in *Proceedings of the 2010 Symposium on Eye-Tracking Research & Applications*. ACM, 2010, pp. 331–337.
- [10] P. Punyabukkana, S. Chanjaradwichai, and A. Suchato, “Design and evaluation of nonverbal sound-based input for those with motor handicapped,” *Disability and Rehabilitation: Assistive Technology*, vol. 8, no. 2, pp. 108–114, 2013.

- [11] R. Sinclair. (2000, May) Microsoft Active Accessibility: Architecture. Microsoft Corporation. [Online]. Available: <http://msdn.microsoft.com/en-us/library/ms971310.aspx>
- [12] Microsoft Corporation. UI Automation Overview. [Online]. Available: <http://msdn.microsoft.com/en-us/library/windows/desktop/ee684076.aspx> (Accessed July 2013)
- [13] Linux Foundation. (2009, December) IAccessible2: Enhancing Accessibility and Multi-Platform Development. [Online]. Available: <http://www.linuxfoundation.org/collaborate/workgroups/accessibility/iaccessible2/overview>
- [14] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock, "Automatically generating personalized user interfaces with SUPPLE," *Artificial Intelligence*, vol. 174, no. 12, pp. 910–950, 2010.
- [15] K. Z. Gajos, J. O. Wobbrock, and D. S. Weld, "Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 2008, pp. 1257–1266.
- [16] J. Abascal, A. Aizpurua, I. Cearreta, B. Gamecho, N. Garay-Vitoria, and R. Minon, "Automatically generating tailored accessible user interfaces for ubiquitous services," in *Proceedings of the 13th international ACM SIGACCESS conference on Computers and accessibility*. ACM, October 2011, pp. 187–194.
- [17] J. H. Andrews and F. Hussain, "Johar: a framework for developing accessible applications," in *Proceedings of the 11th international ACM SIGACCESS conference on Computers and accessibility*. ACM, 2009, pp. 243–244.
- [18] C. Reynolds, "A critical examination of separable user interface management systems: constructs for individualization," *SIGCHI bulletin*, vol. 29, pp. 41–45, 1997.
- [19] D. Benyon, *Designing interactive systems: A comprehensive guide to HCI and interaction design*. Addison Wesley, 2010.
- [20] Ai Squared. ZoomText Magnifier/Reader. [Online]. Available: <http://www.aisquared.com/zoomtext/> (Accessed July 2013)
- [21] S. Burgstahler. (2012) Working together: People with disabilities and computer technology. University of Washington. [Online]. Available: <http://www.washington.edu/doit/Brochures/PDF/wtcomp.pdf> (Accessed July 2013)
- [22] D. Freitas and G. Kouroupetroglou, "Speech technologies for blind and low vision persons," *Technology and Disability*, vol. 20, no. 2, pp. 135–156, 2008.
- [23] Microsoft Corp. Hear text read aloud with Narrator. [Online]. Available: <http://windows.microsoft.com/en-ca/windows-8/hear-text-read-aloud-with-narrator> (Accessed July 2013)

- [24] Apple Inc. VoiceOver for OS X. a feature that speaks for itself. [Online]. Available: <http://www.apple.com/accessibility/osx/voiceover/> (Accessed July 2013)
- [25] The GNOME Project. (2010) What is LSR? [Online]. Available: <https://wiki.gnome.org/LSR> (Accessed July 2013)
- [26] GNOME Project. (2011, November) About Orca. GNOME Foundation. [Online]. Available: <https://wiki.gnome.org/Orca>
- [27] Web Accessibility In Mind (WebAIM). (2012, May) Screen Reader User Survey 4 Results. [Online]. Available: <http://webaim.org/projects/screenreadersurvey4/>
- [28] S. Chanjaradwichai, P. Punyabukkana, and A. Suchato, "Design and evaluation of a non-verbal voice-controlled cursor for point-and-click tasks," in *Proceedings of the 4th International Convention on Rehabilitation Engineering & Assistive Technology*. Singapore Therapeutic, Assistive & Rehabilitative Technologies (START) Centre, 2010, p. 48.
- [29] P. Biswas and P. Robinson, "The cluster scanning system," *Universal Access in the Information Society*, pp. 1–9, 2012.
- [30] P. Biswas and P. Langdon, "A new input system for disabled users involving eye gaze tracker and scanning interface," *Journal of Assistive Technologies*, vol. 5, no. 2, pp. 58–66, 2011.
- [31] M. Basheer. (2013, March) Blind User Computing. [Online]. Available: <http://lbsitbytes2010.wordpress.com/2013/03/20/blind-user-computing-done-by-mufeeda-basheer/>
- [32] Gunnar Schmidt. GNOME Accessibility Architecture (ATK and AT-SPI). GNOME Foundation. [Online]. Available: <http://accessibility.kde.org/developer/atk.php> (Accessed July 2013)
- [33] Apple Inc. (2012, July) Accessibility Overview for OS X. [Online]. Available: <https://developer.apple.com/library/mac/documentation/Accessibility/Conceptual/AccessibilityMacOSX/AccessibilityMacOSX.pdf>
- [34] Oracle Corp. and Affiliates. Package javax.accessibility. [Online]. Available: <http://docs.oracle.com/javase/7/docs/api/javax/accessibility/package-summary.html> (Accessed August 2013)
- [35] H. R. Hartson and D. Hix, "Human-computer interface development: concepts and systems for its management," *ACM Computing Surveys (CSUR)*, vol. 21, no. 1, pp. 5–92, 1989.
- [36] M. Green, "Report on dialogue specification tools," in *Computer Graphics Forum*, vol. 3, no. 4. Wiley Online Library, 1984, pp. 305–313.

- [37] L. Bass, R. Faneuf, R. Little, N. Mayer, B. Pellegrino, S. Reed, R. Seacord, S. Sheppard, and M. R. Szczur, "A metamodel for the runtime architecture of an interactive system," *SIGCHI Bulletin*, vol. 24, no. 1, pp. 32–37, 1992.
- [38] J. Deacon. (2013, September) Model-View-Controller (MVC) architecture. [Online]. Available: <http://www.jdl.co.uk/briefings/MVC.pdf>
- [39] U.ENZLER. (2009, April) Passive View Command (PVC) Pattern. [Online]. Available: <http://www.planetgeek.ch/2009/04/08/passive-view-command-pvc-pattern/>
- [40] F. Alonso, J. L. Fuertes, Á. L. González, and L. Martínez, "User-interface modelling for blind users," in *Computers Helping People with Special Needs*. Springer, 2008, pp. 789–796.
- [41] O. Shaer, R. J. Jacob, M. Green, and K. Luyten, "User interface description languages for next generation user interfaces," in *CHI'08 extended abstracts on Human factors in computing systems*. ACM, 2008, pp. 3949–3952.
- [42] J. Helms, R. Schaefer, K. Luyten, J. Vanderdonckt, J. Vermeulen, and M. Abrams, "User Interface Markup Language (UIML) Specification version 4.0," Organization for the Advancement of Structured Information Standards, Tech. Rep., May 2009.
- [43] A. Puerta and J. Eisenstein, "XIML: A universal language for user interfaces," *White paper*, 2001. [Online]. Available: <http://pascal.joyeux.free.fr/PROBATOIRE/SystemesAutonomes/XimlWhitePaper.pdf>
- [44] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, M. Florins, and D. Trevisan, "USIXML: A user interface description language for context-sensitive user interfaces," in *Proceedings of the ACM AVI'2004 Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages* (Gallipoli 2004), 2004, pp. 55–62.
- [45] D. Ryzko, D. Ryżko, P. Gawrysiak, H. Rybinski, and M. Kryszkiewicz, *Emerging Intelligent Technologies in Industry*, ser. Studies in Computational Intelligence. Springer, August 2011. [Online]. Available: <http://books.google.ca/books?id=VxC4ITrB44YC>
- [46] N. Souchon and J. Vanderdonckt, "A review of XML-compliant user interface description languages," in *Interactive Systems. Design, Specification, and Verification*. Springer, 2003, pp. 377–391.
- [47] L. Wang and A. Sajeev, "Abstract interface specification languages for device-independent interface design: classification, analysis and challenges," in *Pervasive Computing and Applications, 2006 1st International Symposium on*. IEEE, 2006, pp. 241–246.
- [48] K. Z. Gajos, J. J. Long, and D. S. Weld, "Automatically Generating Custom User Interfaces for Users With Physical Disabilities," in *Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility*. ACM, October 2006, pp. 243–244.

- [49] K. Z. Gajos, J. O. Wobbrock, and D. Weld, "Automatically generating user interfaces adapted to users' motor and vision capabilities," in *Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM, October 2007, pp. 231–240.
- [50] Oracle Corporation and Affiliates. Package javax.swing. [Online]. Available: <http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html> (Accessed August 2013)
- [51] Oracle Corporation & Affiliates. Package java.awt.event. [Online]. Available: <http://docs.oracle.com/javase/7/docs/api/java/awt/event/package-summary.html> (Accessed August 2013)
- [52] IBM Corporation. (2013, January) Keyboard equivalents for actions. [Online]. Available: <http://www-03.ibm.com/able/guidelines/software/swkbdequiv.html>
- [53] Canadian Centre for Occupational Health & Safety. (2011, April) Computer Mouse - Common Problems from Use. [Online]. Available: http://www.ccohs.ca/oshanswers/ergonomics/office/mouse/mouse_problems.html

Appendix A

Johar Interface Description File (IDF): Format Specification

This document describes the specification of the Johar Interface Description File (IDF), version 1.0.

A.1 Syntax

A.1.1 Syntax of Attribute Declarations

An IDF consists of *attribute declarations*, possibly separated by whitespace. Whitespace has no particular meaning except inside string or longtext values (see below). Each attribute declaration is of one of the following forms:

Attribute

Attribute name

Attribute = value

Attribute name = value

Each *Attribute* is an upper-case identifier, i.e. a sequence of letters, numbers and underscores starting with an upper-case letter. Attributes must be those named in the specification below. Each *name* is a lower-case identifier, i.e. a sequence of letters, numbers and underscores starting with a lower-case letter. Generally, each *name* is chosen by the writer of the IDF.

Values are of four different forms: identifiers, strings, longtext, and structures. An *identifier* is a sequence of letters, numbers and underscores. A *string* is a sequence of characters in-between paired double quote characters. Inside a string, the backslash is the escape character: the sequence `\` indicates a literal double quote, the sequence `\\` indicates a literal backslash, and the backslash followed by a newline indicates a literal newline.

```

Table clientData
Command open = {
  BriefHelp = "Open or create client data file."
  MultiLineHelp = {{
    Open the file containing data about clients.
    If the file does not exist, it is created.
  }}
  Parameter preferredExtension = {
    Type = text
  }
}

```

Figure A.1: Example of IDF syntax.

Although multi-line strings can be constructed by using the backslash followed by a newline, this is not a very convenient way of entering them. A *longtext* is a value format designed for multi-line text. A *longtext* value consists of a sequence of two open curly braces (“{””) at the end of a line, followed by any lines of arbitrary text, followed by a sequence of two close curly braces (“}””) on a separate line.

A *structure* consists of a single open curly brace, followed by any number of attribute declarations, followed by a single close curly brace. Each attribute declaration has the format described at the top of this section. If an attribute declaration contains a structure, then the attributes in the structure are referred to as *child* attributes, and the enclosing attribute as the *parent* attribute; we also say that the child attributes are *sub-attributes* of the parent.

A.1.2 Example

As an example of all of the above, consider the attribute declarations shown in Figure A.1. In that example, there happen to be no repetitions of attribute names, although in some cases attributes can be repeated. The `Table` attribute has a name but no value; the `Command` and `Parameter` attributes each have both a name and a value (both values happen to be structures); and all other attributes have a value but no name. `Type` has an identifier value, `BriefHelp` has a string value, and `MultiLineHelp` has a *longtext* value. `BriefHelp`, `MultiLineHelp`, and `Parameter` are sub-attributes of `Command`, and `Type` is a sub-attribute of `Parameter`.

A.1.3 Processing of Identifiers as Values

Wherever a double-quoted string literal can appear as a value for an attribute, an upper- or lower-case identifier can also appear if such an identifier would be in the right format. In this

case the value is treated as a string consisting of just the characters in the identifier. For instance, the string "tableEntry" and the lower-case identifier tableEntry are treated exactly the same when they appear as the value of a parameter.

A.2 Allowed Attributes

In what follows below, if no mention is made of the *name* corresponding to an attribute, then the attribute must have no associated name. Each attribute is followed by a description of the *multiplicity* of the attribute, i.e. the number of times that the attribute can appear at the top level and/or as a sub-attribute of another attribute. If an attribute is optional (has multiplicity "0 or 1"), it may have a *default value*, which is also described.

A.2.1 Top-level Attributes

This section describes the attributes that can appear at the top level of the IDF.

- Application: The unique identifier of the application. (Required)
 - *value*: An upper-case identifier (the name of the application).

Multiplicity: Exactly 1.

- ApplicationEngine: The class which contains the application-specific logic of the application.
 - *value*: An upper-case identifier (the name of the application engine class).

Multiplicity: 0 or 1.

Default value: The value of the `Application` attribute.

- Command: A structure representing the basic top-level unit of user-application interaction.
 - *name*: A lower-case identifier (the name of the command).
 - *value*: A structure. See Sections A.2.2, Sub-attributes of Command, and A.2.3, Sub-attributes of Stage and Single-Stage Commands.

Multiplicity: 1 or more.

- CommandGroup: A structure representing a group of related commands.

- *name*: A lower-case identifier (the name of the command group).
- *value*: A structure. See Section A.2.6, Sub-attributes of *CommandGroup*.

Multiplicity: 0 or more.

A *CommandGroup* is a group of conceptually related commands. The commands themselves are defined in *Command* attributes; *CommandGroups* group them by name, and contain no additional information about the *Commands*.

An interface interpreter may use command group information in order to structure the interface. For instance, a classic GUI might present each command group in a separate dropdown menu on a menu bar, and any interface interpreter might use command group information in order to structure help information.

A command cannot appear as a member of more than one command group. All commands that are not explicitly put into a command group in an IDF are put into an implicitly-defined command group, whose name is *commands*. In particular, if no *CommandGroups* are defined, all commands become members of *commands*.

- **IdfVersion**: The version of the IDF format used in the creation of the file. (Required)
 - *value*: A double-quoted string literal, containing a valid major and minor version number.

Multiplicity: Exactly 1.

The version of the first public release of Johar will be 1.0. Therefore, initially all Johar IDFs should contain the declaration `IdfVersion = "1.0"`.

- **InitializationMethod**: The name of the application engine method used to initialize the engine.
 - *value*: A lower-case identifier (the name of the application engine method).

Multiplicity: 0 or 1.

Default value: `applicationEngineInitialize`.

- **Table**: The structure representing a list of similar entities presented to the user by the application engine.
 - *name*: A lower-case identifier (the name of the table).
 - *value*: A structure. See Section A.2.7, Sub-attributes of *Table*.

Multiplicity: 0 or more.

A.2.2 Sub-attributes of Command

This section describes the attributes that are acceptable sub-attributes of any Command attribute. In addition, single-stage Commands (commands that have no Stage attribute) can contain any of the attributes described below in Section A.2.3, Sub-attributes of Stage and Single-Stage Commands.

- **ActiveIfMethod**: The name of an application engine method that can be called to determine if the user should be able to access the command.
 - *value*: A lower-case identifier. This is the method that will be called to determine if the command is active, i.e. if the user should be allowed to issue the command.

Multiplicity: 0 or 1.

If no **ActiveIfMethod** attribute is given, then the command is always active.

It is recommended that the **ActiveIfMethod** is as efficient as possible (e.g. returning only the value of a field or data member), because it may be called frequently by the interface interpreter.

- **BriefHelp**: A brief help message describing the purpose of the command.
 - *value*: A string of 30 characters or fewer, containing no carriage return.

Multiplicity: 0 or 1.

Default value: The **Label** of the command, truncated to 30 characters, if needed.

This attribute, if given, gives a very brief help message describing the purpose of the command, suitable for such things as ToolTips.

- **CommandMethod**: The application engine method that is called to actually carry out the command.
 - *value*: A lower-case identifier. This is the method that will be called to carry out the actual command.

Multiplicity: 0 or 1.

Default value: The command name.

- **Label**: The text describing the Command in the interface.
 - *value*: Any string.

Multiplicity: 0 or 1.

Default value: Derived from the name of the command using standard camel-case translation (see Section A.4).

The `Label`, if given, is the text that will appear (possibly with an appended ellipsis, "...") on the menu item, button, or other interface element corresponding to the command.

- `MultiLineHelp`: A thorough help message describing the purpose of the command.
 - *value*: A string or multi-line text of any length.

Multiplicity: 0 or 1.

Default value: The value of the `OneLineHelp` attribute.

This attribute, if given, gives a thorough help message. IDF writers wanting to give thorough help for each *parameter* of a command should use the `MultiLineHelp` attributes of the *parameters*, as this will promote encapsulation and may (depending on the interface interpreter) give the user more control of the volume of help given.

- `OneLineHelp`: A one-line help message describing the purpose of the command.
 - *value*: A string of 80 characters or fewer, containing no carriage return.

Multiplicity: 0 or 1.

Default value: The value of the `BriefHelp` attribute.

This attribute, if given, gives a help message describing the purpose of the command, somewhat longer than the `BriefHelp`. The `OneLineHelp` messages may be useful for the interface interpreter to display in a list.

- `Prominence`: An integer describing how prominently the Command should be shown to the user.
 - *value*: An integer greater than or equal to 0. Value ranges:
 - * 3000 or more: High prominence. For instance, in a classic GUI, commands with prominence 3000 or more might be placed on the screen as buttons so that they are as quickly accessible as possible.
 - * 2000-2999: Normal prominence.
 - * 1000-1999: Reduced prominence.
 - * 0-999: Low prominence. For instance, in a classic GUI, commands with prominence 0-999 might be placed on a "More commands" popup.

Multiplicity: 0 or 1.

Default value: 2000.

- **Question:** A question which may be asked of the user, given the values of the `Parameters` and previous `Questions` in the command.
 - *name*: A lower-case identifier (the name of the question).
 - *value*: A structure. See Section A.2.5, Sub-attributes of `Question`.

Multiplicity: 0 or more.

- **QuitAfter:** An indication of whether the interface interpreter running the application should always terminate after the command terminates.
 - *value*: A Johar boolean (e.g., `yes` or `no`). See Section A.3.

Multiplicity: 0 or 1.

Default value: `no`.

A value of `yes` means that the interface interpreter should always quit after execution of the `CommandMethod`. A value of `no` means that it should expect more commands, unless there is a `QuitAfterIfMethod` which returns `true` (see below).

- **QuitAfterIfMethod:** An application engine method that is called to determine whether the interface interpreter running the application engine should terminate after the command terminates.
 - *value*: A lower-case identifier (the method to be called to determine quit status). The method should return a boolean (`true` if the application should terminate, `false` otherwise).

Multiplicity: 0 or 1.

If a `QuitAfterIfMethod` attribute is present, the indicated method is called after the `CommandMethod` is called. If the `QuitAfterIfMethod` returns `true`, then the interface interpreter takes this to be an indication that the application should terminate.

- **Stage:** One stage in the processing of the `Command`. Each stage may take separate `Parameters`.
 - *name*: A lower-case identifier (the name of the `Stage`).
 - *value*: A structure. See Section A.2.3, Sub-attributes of `Stage` and `Single-Stage Commands`.

Multiplicity: 0 or more.

A `Command` can be broken up into several `Stages`. There can be zero or more explicit `Stage` sub-attributes. If there are zero explicit `Stage` attributes, then one stage is implicitly defined. All the attributes mentioned in Section A.2.3 which appear as sub-attributes of the `Command` are then placed into the one implicitly-defined stage. The names of the `Parameters` in all of the `Stages` in a command must be disjoint.

Each stage in a multi-stage command may be handled at a separate time by an interface interpreter. A classic GUI interface interpreter, for instance, may present a multi-stage command using a “wizard”-style dialog box, in which the user can move forward or backward through the stages by clicking a “next” button. This may facilitate the elicitation of parameters for infrequently-given commands or commands that have many parameters.

If a `Command` has more than one `Stage`, then the interface interpreter may call the `ParameterCheckMethods` of each stage separately, and the `DefaultValueMethods` of each parameter separately (see below for more thorough information). This gives a mechanism by which the user can supply values for parameters in one stage, which are then used to compute the default values of other parameters. Thus, if the application programmer needs to collect user-input values of parameter A (e.g. input file name) before computing the default value of parameter B (e.g. output file name), parameter A can be in an earlier stage and parameter B in a later stage.

A.2.3 Sub-attributes of Stage and Single-Stage Commands

These sub-attributes can appear inside a `Stage` in a `Command`. If the command has no explicit `Stages`, they can also appear directly inside the `Command`, in which case they define the sub-attributes of the one implicit stage of the command.

- **Parameter**: A piece of data which comes from the user and is relevant to the `Command`, such as an integer, floating-point number or string.
 - *name*: A lower-case identifier (the name of the parameter).
 - *value*: A structure. See Section A.2.4, Sub-attributes of `Parameter`.

Multiplicity: 0 or more.

- **ParameterCheckMethod**: A method in the application engine that can be called to check the validity of the stage’s parameters.

- *value*: A lower-case identifier. This is the method that will be called to check the validity of the parameter values. It should return a string value (null or the empty string if all the parameters are valid, an error message if one or more parameters are invalid).

Multiplicity: 0 or 1.

If a `ParameterCheckMethod` is given, then if the user has input invalid values, the interface interpreter can display the error message and allow the user to edit the erroneous values they originally gave. The `ParameterCheckMethod` should not do any processing related to the main function of the command, since the user may later decide to change the parameters, or even to cancel the command.

If no `ParameterCheckMethod` is given, then the parameter values will be checked by any implicit rules given for the parameter (e.g. the `MinValue` and `MaxValue` attributes for an `int` parameter).

A.2.4 Sub-attributes of Parameter

This section describes the attributes that are acceptable sub-attributes of any `Parameter` attribute.

- **BriefHelp**: A brief help message describing the meaning of the parameter.
 - *value*: A string of 30 characters or fewer, containing no carriage return.

Multiplicity: 0 or 1.

Default value: The `Label` of the parameter, truncated to 30 characters, if needed.

This attribute, if given, gives a very brief help message describing the purpose of the parameter. For instance, a classic GUI could use this as the text of a `ToolTip`.

- **Choices**: A string representing the possible choices of values of a parameter of type `choice` (see below, attribute `Type`).
 - *value*: A double-quoted string literal.

Multiplicity: For parameters of type `choice`, exactly 1. For other parameters, 0.

The string contains the possible choices of values, separated by bar (“|”) characters; e.g. “`portrait|landscape`”, “`clubs|diamonds|hearts|spades`”. To include a literal bar character in a choice, the bar should be preceded by a backslash (“\”) character. To include a literal backslash character in a choice, use two backslashes (“\\”).

- **DefaultValue**: The default value for the parameter.
 - *value*: A boolean, integer, real number or double-quoted string literal, as appropriate.

Multiplicity: 0 or 1.

If the user gives no explicit value for a parameter, then the interface interpreter will act as if they have given the `DefaultValue` as the value. For a parameter of type `choice`, the value must be one of the choices in the `Choices` attribute string.

- **DefaultValueMethod**: A method in the application engine to be called to give the default value for the parameter.
 - *value*: A lower-case identifier (the name of the method to be called to return the default value).

Multiplicity: 0 or 1. A parameter cannot have both a `DefaultValue` attribute and a `DefaultValueMethod` attribute.

In a Java application engine, the `DefaultValueMethod` must return a value of type `boolean`, `long`, `double` or `String`, as appropriate. For a parameter of type `choice`, the value returned must be one of the choices in the `Choices` attribute string.

- **FileConstraint**: A constraint on the status of a parameter of type `file` (see below, attribute `Type`).
 - *value*: A lower-case identifier. Accepted values:
 - * `mustExist`: The file must exist at the time the command is issued.
 - * `mustBeReadable`: The file must exist and be readable by the application at the time the command is issued.
 - * `mustNotExistYet`: The file must not exist at the time the command is issued.
 - * `none`: No constraint.

Multiplicity: For parameters of type `file`, 0 or 1. For other parameters, 0.

Default value: `none`.

- **Label**: The string that will be used to describe the parameter if and when the user is asked to enter a value for the parameter.
 - *value*: A double-quoted string literal.

Multiplicity: 0 or 1.

Default value: Derived from the name of the command using standard camel-case translation (see Section A.4).

The `Label`, if given, is the text that will appear to the user to indicate what parameter they are to enter. For instance, in a classic GUI, this would be text displayed beside the user-controllable widget used to set the value of the parameter.

- `MaxNumberOfChars`: The maximum number of characters that can be entered by the user as the value of a `text` parameter (see below, attribute `Type`).
 - *value*: An integer literal greater than or equal to 1; or the lower-case identifier `unlim`.

Multiplicity: For parameters of type `text`, 0 or 1. For other parameters, 0.

Default value: `unlim`.

- `MaxNumberOfLines`: The maximum number of lines that can be entered by the user as the value of a `text` parameter (see below, attribute `Type`).
 - *value*: An integer literal greater than or equal to 1; or the lower-case identifier `unlim`.

Multiplicity: For parameters of type `text`, 0 or 1. For other parameters, 0.

Default value: 1.

- `MaxNumberOfReps`: The maximum number of repetitions of the parameter that the user can give.
 - *value*: An integer literal greater than or equal to 1, giving the maximum number of repetitions allowed for this parameter; or the lower-case identifier `unlim`.

Multiplicity: 0 or 1.

Default value: 1.

The user is not allowed to give more than `MaxNumberOfReps` repetitions of the parameter. See `MinNumberOfReps` for more detail.

- `MaxValue`: The maximum possible value of a parameter of type `int` or `float` (see below, attribute `Type`).
 - *value*: An integer or real number literal.

Multiplicity: For parameters of type `int` or `float`, 0 or 1. For other parameters, 0.

Default value: The maximum value representable in a signed 64-bit integer (resp. 64-bit floating-point) number.

- MinNumberOfReps: The minimum number of repetitions of the parameter that the user can give (see below).
 - *value*: An integer literal greater than or equal to 0, giving the minimum number of repetitions allowed for this parameter.

Multiplicity: 0 or 1.

Default value: 1.

`MinNumberOfReps` must be less than or equal to `MaxNumberOfReps`.

The minimum and maximum number of repetitions indicate how many times the parameter can be repeated. If a `Parameter` has a `DefaultValue` or `DefaultValueMethod`, then any repetitions up to the `MinNumberOfReps` that are not explicitly changed by the end user are filled in by that value. If the `Parameter` has neither a `DefaultValue` nor a `DefaultValueMethod`, and `MinNumberOfReps` is greater than 0, then the end user is required to fill in at least `MinNumberOfReps` repetitions.

For example, a command which takes a person's name as a parameter might have a `Parameter` of type `text` with no default value and a minimum and maximum number of repetitions equal to 1, obliging the user to enter a name. As another example, a command to show the differences between two files might take exactly two file parameters. To enforce this restriction, the programmer might create a `Parameter` with type `file` and with `MinNumberOfReps` and `MaxNumberOfReps` both equal to 2.

- MinValue: The minimum possible value of a parameter of type `int` or `float` (see below, attribute `Type`).
 - *value*: An integer or real number literal.

Multiplicity: For parameters of type `int` or `float`, 0 or 1. For other parameters, 0.

Default value: The minimum value representable in a signed 64-bit integer (resp. 64-bit floating-point) number.

`MinValue` must be less than or equal to `MaxValue`.

- MultiLineHelp: A thorough help message describing the meaning of the parameter.

- *value*: A string or multi-line text of any length.

Multiplicity: 0 or 1.

Default value: The `OneLineHelp` of the parameter.

This attribute, if given, gives a thorough help message regarding the parameter.

- `OneLineHelp`: A one-line help message describing the meaning of the parameter.

- *value*: A string of 80 characters or fewer, containing no carriage return.

Multiplicity: 0 or 1.

Default value: The `BriefHelp` of the parameter.

This attribute, if given, gives a help message describing the purpose of the parameter, somewhat longer than the `BriefHelp`. The `OneLineHelp` messages may be useful for the interface interpreter to display in a list.

- `ParentParameter`: The name of another parameter that controls the existence of the current parameter.

- *value*: A lower-case identifier (the name of the parent parameter).

Multiplicity: 0 or 1.

This attribute and `ParentValue` are used for parameters that only make sense when some other parameter has a certain value. In this case the other parameter is referred to as the parent parameter.

As an example, when printing a document, the user may choose to “print to a file”, in which case the user must enter a file name parameter. However, if the user does not choose to print to a file, then there is no reason to expect the user to enter a file name. This situation can be set up by giving the `print` command two parameters: `printToFile`, a boolean parameter, and `outputFileName`, a file parameter whose parent parameter is `printToFile` and whose `ParentValue` is `true`. If the parent parameter does not have the indicated `ParentValue`, then the user is not required to enter any value for the parameter, regardless of any information about the minimum number of repetitions required.

Circular `ParentParameter` references are not allowed. That is, a chain of `ParentParameter` references must end in a parameter with no `ParentParameter`.

- `ParentValue`: The value of the parent parameter that triggers the existence of this parameter.

- *value*: An integer, floating-point or string literal, as appropriate.

Multiplicity: For parameters with a `ParentParameter` value, exactly 1. For other parameters, 0.

See `ParentParameter` for more detail.

- Prominence: An integer describing how prominently the `Parameter` should be shown to the user.

- *value*: An integer greater than or equal to 0. Value ranges:
 - * 3000 or more: High prominence. For instance, in a classic GUI, parameters with prominence 3000 or more may be placed in the most prominent location in the dialog box.
 - * 2000-2999: Normal prominence.
 - * 1000-1999: Reduced prominence.
 - * 0-999: Low prominence. For instance, in a classic GUI, parameters with prominence 0-999 might be accessed only through an “Advanced” button in the parameter dialog box.

Multiplicity: 0 or 1.

Default value: 2000.

- RepsModel: A description of the intended model for the repetitions of the parameter.

- *value*: A lower-case identifier. Accepted values:
 - * **set**: The repetitions of the parameter are considered to be a set of disjoint values. The interface interpreter does not have to keep track of the order in which the user has input the values, and does not have to load them in the Gem in the order that the user has given them. The interface interpreter does not have to provide a way for the user to input multiple repetitions that have the same value.
 - * **multiset**: The repetitions of the parameter are considered to be a set of values, with one repetition possibly having the same value as another. The interface interpreter does not have to keep track of the order in which the user has input the values, and does not have to load them in the Gem in the order that the user has given them. However, it does have to provide the user with the ability to give multiple repetitions with the same value.

- * **sequence**: The repetitions of the parameter are considered to be a sequence of values, with one repetition possibly having the same value as another. The interface interpreter must load them in the Gem in the order that the user has given them.

Multiplicity: 0 or 1.

Default value: `set`.

The `RepsModel` may be used by an interface interpreter in order to structure the interface. For instance, in a classic GUI, a `choice` parameter with `MaxNumberOfReps = unlim` and `RepsModel = set` may be presented as a set of radio buttons, any of which can be turned on. In contrast, a `file` parameter with `MaxNumberOfReps = unlim` and `RepsModel = sequence` must be presented so that the user can specify a sequence of files, for instance so that the application engine method is guaranteed to process them in that order.

- **SourceTable**: The `Table` associated with a `tableEntry` parameter (see below, attribute `Type`).
 - *value*: A lower-case identifier (the name of the table from which the user selects values for the parameter).

Multiplicity: For a parameter of type `TableEntry`, exactly 1. For other parameters, 0.

- **Type**: One of a few values describing what kind of parameter it is.
 - *value*: A lower-case identifier, signifying the type of the parameter. Possible values are:
 - * **boolean**: Either true or false.
 - * **choice**: One of a fixed number of choices. Every parameter with a `Type` of `choice` must have a `Choices` attribute.
 - * **date**: A calendar date.
 - * **file**: A file name.
 - * **float**: A floating-point number.
 - * **int**: An integer.
 - * **text**: A text string.
 - * **tableEntry**: The parameter is an entry from one of the tables declared in the IDF. Every parameter with a `Type` of `tableEntry` must have a `SourceTable` attribute.

* `timeOfDay`: A time of day.

Multiplicity: exactly 1.

A.2.5 Sub-attributes of Question

This section describes the attributes that are acceptable sub-attributes of any `Question` attribute.

`Questions` are similar to `Parameters`. However, the intention is that the system expects a value for a `Question` only when an application engine method judges that a value is required, based on the values of the `Parameters`. An interface interpreter may also provide a “cancel” option when asking a question, to allow the user to cancel the command in response to the question.

- **AskIfMethod**: The application engine method to call in order to see if the question should be asked. (Required)
 - *value*: A lower-case identifier, the name of the method to call to determine whether to require a value for the `Question`.

Multiplicity: exactly 1.

The application engine programmer can assume that the values for the `Parameters` of the command, and all previous `Questions`, are accessible from the `Gem` when the `ask-if` method is called.

For example, say that the programmer of an editor application wants the question “File has been modified. Save changes?” to be asked at appropriate points (for instance, for the `close`, `new` and `quit` commands). They could do this by adding the following `Question` to all appropriate commands:

```
Question saveIfModified = {
  Type = boolean
  Label = "File has been modified. Save changes?"
  AskIfMethod = fileModified
}
```

- The following attributes of `Parameter` are also acceptable sub-attributes of any `Question`:
 - `BriefHelp`

- Choices
 - DefaultValue
 - DefaultValueMethod
 - FileConstraint
 - Label
 - MaxNumberOfChars
 - MaxNumberOfLines
 - MaxValue
 - MinValue
 - MultiLineHelp
 - OneLineHelp
 - Prominence
 - SourceTable
 - Type
- The following attributes of Parameter are *not* acceptable sub-attributes of any Question:
 - MaxNumberOfReps
 - MinNumberOfReps
 - ParentParameter
 - ParentValue
 - RepsModel

In addition, the SourceTable of any question of type tableEntry must refer to a non-browsable table.

A.2.6 Sub-attributes of CommandGroup

This section describes the attributes that are acceptable sub-attributes of any CommandGroup attribute.

- Label: The text describing the CommandGroup in the interface.
 - *value*: Any string.

Multiplicity: 0 or 1.

Default value: Derived from the name of the command group using standard camel-case translation (see Section A.4).

The `Label`, if given, is the text that will appear on the menu heading, button, or other interface element corresponding to the command group.

- **Member**: One of the commands which is a member of this `CommandGroup`.
 - *value*: a lower-case identifier specifying one command that is a member of this command group.

Multiplicity: 1 or more.

A.2.7 Sub-attributes of Table

This section describes the attributes that are acceptable sub-attributes of any `Table` attribute.

- **Browsable**: A value indicating whether the user should be able to browse the table.
 - *value*: A Johar boolean (e.g., yes or no). See Section A.3.

Default value: yes.

Some tables are intended to be presented to the user for them to browse and select rows in, separately from the processing of a given command. Others are simply intended to be tables of candidate values for parameters. The former kind of table is referred to as “browsable”, the latter “not browsable”. The `Browsable` attribute controls this behaviour.

- **DefaultHeading**: The default heading of the table.
 - *value*: A double-quoted string or long text.

Default value: The `Label` of the table.

This attribute gives the heading that will be associated with the table if no heading is set by the application engine. If no `DefaultHeading` attribute is given, then the heading will be derived from the name of the table using standard camel-case translation (see Section A.4).

- **Label**: The text describing the `Table` in the interface.
 - *value*: Any string.

Multiplicity: 0 or 1.

Default value: Derived from the name of the table using standard camel-case translation (see Section A.4).

The `Label`, if given, is the text that will appear on the menu item, button, tab, or other interface element corresponding to the table.

A.2.8 Generated Attribute Values

When an IDF is read, it is processed into an internal form that can be used by an interface interpreter. For convenience, default values are generated automatically for some sub-attributes if they are not given explicitly in the IDF. The descriptions of these default values are given above, in the sections pertinent to the individual parameters.

The label and help attributes, if not given, are generated in a specific sequence. The sequence, along with the default values generated, are as follows.

- `Label`: The de-camel-cased version of the name of the command, command group, parameter or question.
- `BriefHelp`: The `Label` for the command, parameter, or question, truncated to 30 characters if necessary.
- `OneLineHelp`: The value of `BriefHelp`.
- `MultiLineHelp`: The value of `OneLineHelp`.

A.3 Johar Booleans

In some places in Johar IDFs, it is possible to give a boolean value. In all of these places, the following values are acceptable:

- `yes`, `Yes`, `YES`, `true`, `True`, or `TRUE`, all of which mean the same thing.
- `no`, `No`, `NO`, `false`, `False`, or `FALSE`, all of which mean the same thing.

This flexibility is intended to mirror the flexibility which interface interpreters are encouraged to have in accepting boolean values from end users of the applications.

Camel-Case Identifier	Translation
add	Add
saveAs	Save as
findInThisPage	Find in this page
inputTextFile	Input text file
inputXMLFile	Input x m l file
inputXmlFile	Input xml file

Figure A.2: Examples of camel-case translation.

A.4 Camel Case Translation

To simplify interface description files, some values of attributes of commands and parameters are translated into strings shown to the user by following an algorithm for translating camel-case identifiers. This section describes this algorithm.

“Camel case” is the phrase used to refer to the practice of writing identifiers with mixed upper- and lower-case letters but no underscores. Often, when camel case is used, each upper-case letter is intended to start a word. The algorithm used for camel-case translation is therefore as follows.

- (1) Separate the words in the identifier by single spaces, assuming that each upper-case letter starts a new word.
- (2) Capitalize the first letter of the resulting string if it is not capitalized.
- (3) Translate all the rest of the letters in the resulting string to lower-case.

Figure A.2 shows some examples of the effect of the camel-case translation algorithm. The first four examples are likely to be what the developer wants; the last two are unlikely to be what the developer wants. If the effect of the translation algorithm is not what the developer wants, then they can use the `Label` attribute to achieve what they want – for instance, by using a parameter name `inputXMLFile` with the attribute `Label = "Input XML file"`.

Appendix B

Interface Interpreters (IntIs): Requirements Specification

This document describes the requirements that must be met by all Johar interface interpreters (IntIs). It is assuming that the IntI is written in Java, but similar requirements apply to IntIs written in any language.

In this document, the words *must* and *must not* are in italics and indicate normative statements (strict requirements for the IntI). The words *may* and *does not have to* are also in italics, but indicate behaviour that is permitted or not forbidden; they are used for clarity. The words *should* and *should not* are also in italics, and indicate behaviour that is recommended but not necessary.

The requirements in this document are numbered 1, 2, 3 and so on. They will be referred to in other documents as IntI-R1, IntI-R2, IntI-R3, and so on.

B.1 Core Steps

This section describes the sequence of steps that must be taken by an IntI in interacting with the IDF, the GemSetting, and the application engine. We refer to these as the “core steps”.

- (1) A given IntI *may* take many steps other than the core steps, and *may* interact with the user in many ways outside of these core steps (for instance, in providing help or facilitating table browsing).
- (2) However, whenever it interacts with the IDF, the GemSetting and the application engine, the method calls *must* follow the pattern of the core steps.

Core steps:

- (3) Initialization phase:

- (i) The `IntI` *must* first get a `johar.idf.Idf` object by using the method `johar.idf.Idf.idfFromFile(fname)`, where the file name argument is the name of an XML or IDF-format file.
 - (ii) If `johar.idf.Idf.idfFromFile(fname)` throws an `IdfFormatException`, then the `IntI` *must* show the error message and exit.
 - (iii) If the IDF's `IdfVersion` is greater than that supported by the `IntI`, then the `IntI` *must* show an error message and exit.
 - (iv) The `IntI` *must* then call `GemFactory.newGemSetting()`, using the `Idf` object returned by `idfFromFile` as the first parameter and a valid `ShowTextHandler` as the second parameter.
 - (v) The `IntI` *must* then call `GemSetting.validate()` on the `GemSetting` returned by `GemFactory`.
 - (vi) If `GemSetting.validate()` throws an `IdfFormatException`, the `IntI` *must* show the error message and exit.
 - (vii) The `IntI` *must* then call `GemSetting.initializeAppEngine()`. (This will set up the `Gem` tables and call the `InitializationMethod` for the app engine.)
- (4) After the initialization phase, the `IntI` *may* continue with the processing described below.
 - (5) The `IntI` *may* then perform the Command Loop (see below) as many times as needed, until the `IntI` terminates.
 - (6) At any time while executing the Command Loop, the `IntI` *may* exit its current iteration and begin a new iteration. [Rationale: the user may cancel the command processing at any time.]

The Command Loop consists of the following steps:

- (7) The `IntI` *may* call the `ActiveIfMethods` of any commands that have them.
- (8) It *must* then call `GemSetting.selectCurrentCommand(cmdName)`. If `cmdName` corresponds to a command with an `ActiveIfMethod`, then that method *must* be one that was called since the beginning of this iteration of the Command Loop. [Rationale: the active status of a command may have been changed by the effect of the previous command.]
- (9) It *must* then call `GemSetting.selectCurrentStage(0)`. [Rationale: the parameter values for one stage must be loaded and checked before the default values of the

parameters in the next stage are obtained, to ensure correct communication with the application engine methods of multi-stage commands.]

- (10) The `IntI` *may* then perform the Stage Loop (see below) as many times as needed, at least until `MinNumberOfReps` repetitions of every parameter of every stage of the command has been loaded into the `GemSetting`.
- (11) The `IntI` *may* continue to perform the Stage Loop after a value for every parameter of every stage of the command has been loaded.
- (12) The `IntI` *must* continue to perform the Stage Loop until every `ParameterCheckMethod`, in every stage of the command that has one, has returned `null` or the empty string.
- (13) The `IntI` *must* perform the following steps for every question in the current command, from the first question specified to the last question specified.
 - (i) The `IntI` *must* call the `AskIfMethod` of the question.
 - (ii) If the `AskIfMethod` returns `true`, and the question has a `DefaultValueMethod`, the `IntI` *must* call the `DefaultValueMethod` of the question.
 - (iii) If the `AskIfMethod` returns `true`, the `IntI` *must* load a value for the question into the `GemSetting`.
- (14) The `IntI` *must* call the `CommandMethod` for the command.
- (15) The `IntI` *must* ensure that the `showText` text from the command has been effectively communicated to the user. [Rationale: the `showText` from the last command executed in a run of the application may be important, and must not be rendered inaccessible by the termination of the application.]
- (16) If the command's `QuitAfter` attribute is `false`, but the command has a `QuitAfterIfMethod`, then the `IntI` *must* call that method.
- (17) If the command's `QuitAfter` attribute is `true`, or the command has a `QuitAfterIfMethod` which has returned `true`, then the `IntI` *must* terminate.

See Table B.1 for the Java type that a Java `IntI` *must* load as the value of a parameter, depending on what Type the parameter is.

Parameter Type	Java type	Comment
boolean	boolean	
choice	java.lang.String	One of the choices
date	java.util.Calendar	
file	java.io.File	
float	double	The floating-point type with the maximum range and precision in Java
int	long	The integer type with the maximum range and precision in Java
text	java.lang.String	
tableEntry	int	The row number (starting with 0) of one row that the user has selected
timeOfDay	java.util.Calendar	

Table B.1: Bindings of Johar parameter types to Java types.

The Stage Loop consists of the following steps:

- (18) At any time, the IntI *may* call the `DefaultValueMethod` of any parameter in the current stage.
- (19) At any time, the IntI *may* load values for repetitions of any parameter in the current stage.
- (20) If the IntI loads a value for a repetition of a parameter with a `DefaultValue` or `DefaultValueMethod`, and the value was not selected directly by the user, then the value *must* come from the `DefaultValue` or from a call to the `DefaultValueMethod` made since the beginning of this iteration of the Stage Loop. [Rationale: the previous stage's `ParameterCheckMethod` may have caused a change to the value returned by the parameter's `DefaultValueMethod`.]
- (21) The IntI *must* load at least `MinNumberOfReps` repetitions of each parameter in the current stage before calling the current stage's `ParameterCheckMethod`, unless the parameter has a `ParentParameter` whose value is not the parameter's `ParentValue`.
- (22) At any time after that, the IntI *may* call the current stage's `ParameterCheckMethod`, if it has one.
- (23) If the current stage has a `ParameterCheckMethod`, then the IntI *must* call it at least once.
- (24) If the current stage has a `ParameterCheckMethod`, then the IntI *must not* load any parameter values between the time it last calls the `ParameterCheckMethod` and the

time it next calls `GemSetting.selectCurrentStage()`.

- (25) Finally, the IntI *may* call `GemSetting.selectCurrentStage()` with a parameter that is either one greater than the index number of the current stage, or less than the index number of the current stage.

B.2 Other Requirements

Requirements concerning values of parameters and questions:

- (26) The IntI *must* provide a way for the user to select values for parameters and questions for any command.
- (27) For a parameter or question of type `choice`, any value loaded by the IntI *must* be a choice from the parameter's or question's `Choices` string.
- (28) For a parameter or question of type `file`, any value loaded by the IntI *must* respect the parameter's or question's `FileConstraint`.
- (29) For a parameter or question of type `text`, any value loaded by the IntI *must* respect the parameter's or question's `MaxNumberOfChars` and `MaxNumberOfLines` attribute values.
- (30) For a parameter or question of type `int` or `float`, any value loaded by the IntI *must* respect the parameter's or question's `MaxValue` and `MinValue` attribute values.

Requirements concerning repetitions of parameters:

- (31) The IntI *must* provide a way for the user to select multiple values for repetitions of every parameter of a command, up to the parameter's `MaxNumberOfReps`.
- (32) For every parameter, the IntI *must* load at least `MinNumberOfReps` repetitions for the parameter before calling the `CommandMethod` of the command, unless the parameter has a `ParentParameter` whose value is not the `ParentValue` of the parameter.
- (33) For every parameter, the IntI *must not* load a value for the parameter if the parameter has a `ParentParameter` whose value is not the `ParentValue` of the parameter.
- (34) For every parameter, the IntI *must* load at most `MaxNumberOfReps` repetitions for the parameter before calling the `CommandMethod` of the command.

- (35) If the `RepsModel` of a parameter is `set`, then the `IntI` *may* load only one repetition of each value selected by the user.
- (36) If the `RepsModel` of a parameter is `multiset` or `sequence`, then the `IntI` *must* load the number of repetitions of each value selected by the user.
- (37) If the `RepsModel` of a parameter is `set` or `multiset`, then the `IntI` *may* load values in any order.
- (38) If the `RepsModel` of a parameter is `sequence`, then the `IntI` *must* load values in the order specified by the user.

Requirements concerning user interface structure:

- (39) The `IntI` *may* use the value of `Application` as a unique identifier of the application currently being run.
- (40) The `IntI` *may* use the application's command groups in order to structure the interface.
- (41) The `IntI` *may* use the `Prominence` of commands, etc. in order to structure the interface.
- (42) The `IntI` *should* use the `Label` of a command, parameter, or question as a description, where needed in the interface.
- (43) The `IntI` *may* use the `RepsModel` of any parameter in order to structure the interface.
- (44) The `IntI` *should* provide a convenient way for the user to select valid values for parameters or questions of type `date`, `file`, and `timeOfDay`.

Other `IntI` requirements:

- (45) The `IntI` *must* provide access to all commands in the application.
- (46) For a command with any stage with any parameter of type `tableEntry`, where the `SourceTable` is browsable, the `IntI` *must not* select the command as the current command unless the user has selected at least `MinNumberOfReps` entries in the parameter's `SourceTable`.
- (47) The `IntI` *must* allow the user to access, browse and select rows in all browsable tables that are not currently hidden.
- (48) The `IntI` *must not* allow the user to access, browse or select rows in non-browsable tables.

- (49) The IntI *must not* allow the user to access, browse or select rows in tables that are currently hidden.
- (50) The IntI *must* provide access to all the help messages provided in the IDF.
- (51) The IntI *may* use any of the Label, BriefHelp, OneLineHelp, and MultiLineHelp messages provided in the IDF wherever they are needed.

Appendix C

Johar XML Schema Document

The Johar XSD (*johar.xsd*) is used for validating the XML equivalent of an IDF.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!--
    Design:
    1. Every sub-attribute can be given in any order, so every
       sub-attribute is declared as an "xs:choice".
    2. Each "xs:choice" is given an unbounded number of occurrences.
    3. The Java code checks to make sure that there are enough and
       not too many of each sub-attribute.
    This design seems to be needed in order to avoid either a long
    XSD file or imposing an order on the sub-attributes.
  -->

  <!-- The overall format of a Johar Interface Declaration File (IDF) -->
  <xs:element name="Johar">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">

        <xs:element name="Application" type="upperIdentifier"/>
        <xs:element name="ApplicationEngine" type="xs:string"/>
        <xs:element name="Command" type="CommandType"/>
        <xs:element ref="CommandGroup"/>
        <xs:element name="IdfVersion" type="xs:string"/>
        <xs:element name="InitializationMethod" type="lowerIdentifier"/>
        <xs:element ref="Table"/>

      </xs:choice>
    </xs:complexType>
  </xs:element>
```

```

<!-- ===== BEGIN declarations relevant to Commands ===== -->

<!-- What can go inside a Command declaration -->
<xs:complexType name="CommandType">
  <xs:choice minOccurs="0" maxOccurs="unbounded">

    <xs:element name="ActiveIfMethod" type="lowerIdentifier"/>
    <xs:element ref="BriefHelp"/>
    <xs:element name="CommandMethod" type="lowerIdentifier"/>
    <xs:element name="Label" type="xs:string"/>
    <xs:element name="MultiLineHelp" type="xs:string"/>
    <xs:element ref="OneLineHelp"/>
    <xs:element name="Parameter" type="paramType"/>
    <xs:element name="ParameterCheckMethod" type="lowerIdentifier"/>
    <xs:element name="Prominence" type="nonNegativeInt"/>
    <xs:element name="Question" type="questionType"/>
    <xs:element name="QuitAfter" type="joharBoolean"/>
    <xs:element name="QuitAfterIfMethod" type="lowerIdentifier"/>
    <xs:element name="Stage" type="StageContents"/>

  </xs:choice>
  <xs:attribute name="name" type="xs:string"/>
</xs:complexType>

<!-- What can go inside a Stage declaration -->
<xs:complexType name="StageContents">
  <xs:choice minOccurs="0" maxOccurs="unbounded">

    <xs:element name="Parameter" type="paramType"/>
    <xs:element name="ParameterCheckMethod" type="lowerIdentifier"/>

  </xs:choice>

  <xs:attribute name="name" type="xs:string"/>
</xs:complexType>

<!-- ===== END declarations relevant to Commands ===== -->

<!-- BEGIN declarations relevant to Parameters and Questions -->

<!-- What can go inside a Parameter declaration -->
<xs:complexType name="paramType">

```

```

<xs:choice minOccurs="0" maxOccurs="unbounded">

  <xs:element ref="BriefHelp"/>
  <xs:element name="Choices" type="xs:string"/>
  <xs:element name="DefaultValue" type="xs:string"/>
  <xs:element name="DefaultValueMethod" type="lowerIdentifier"/>

  <xs:element name="FileConstraint">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="mustBeReadable"/>
        <xs:enumeration value="mustExist"/>
        <xs:enumeration value="mustNotExistYet"/>
        <xs:enumeration value="none"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>

  <xs:element name="Label" type="xs:string"/>
  <xs:element name="MaxNumberOfChars" type="positiveInt"/>
  <xs:element name="MaxNumberOfLines" type="positiveInt"/>
  <xs:element name="MaxNumberOfReps" type="positiveInt"/>
  <xs:element name="MaxValue" type="xs:decimal"/>
  <xs:element name="MinNumberOfChars" type="nonNegativeInt"/>
  <xs:element name="MinNumberOfReps" type="nonNegativeInt"/>
  <xs:element name="MinValue" type="xs:decimal"/>
  <xs:element name="MultiLineHelp" type="xs:string"/>
  <xs:element ref="OneLineHelp"/>
  <xs:element name="ParentParameter" type="lowerIdentifier"/>
  <xs:element name="ParentValue"/>
  <xs:element name="Prominence" type="nonNegativeInt"/>

  <xs:element name="RepsModel">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="set"/>
        <xs:enumeration value="multiset"/>
        <xs:enumeration value="sequence"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>

  <xs:element name="SourceTable" type="lowerIdentifier"/>

```

```

<xs:element name="Type">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="boolean"/>
      <xs:enumeration value="choice"/>
      <xs:enumeration value="date"/>
      <xs:enumeration value="file"/>
      <xs:enumeration value="float"/>
      <xs:enumeration value="int"/>
      <xs:enumeration value="text"/>
      <xs:enumeration value="tableEntry"/>
      <xs:enumeration value="timeOfDay"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

</xs:choice>

  <xs:attribute name="name" type="xs:string"/>
</xs:complexType>

<!-- What can go inside a Question declaration -->
<xs:complexType name="questionType">

  <xs:choice minOccurs="0" maxOccurs="unbounded">

    <xs:element name="AskIfMethod" type="lowerIdentifier"/>

    <xs:element ref="BriefHelp"/>
    <xs:element name="Choices" type="xs:string"/>
    <xs:element name="DefaultValue" type="xs:string"/>
    <xs:element name="DefaultValueMethod" type="lowerIdentifier"/>

    <xs:element name="FileConstraint">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="mustBeReadable"/>
          <xs:enumeration value="mustExist"/>
          <xs:enumeration value="mustNotExistYet"/>
          <xs:enumeration value="none"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>

    <xs:element name="Label" type="xs:string"/>

```



```

<xs:element name="MaxNumberOfChars" type="positiveInt"/>
<xs:element name="MaxNumberOfLines" type="positiveInt"/>
<xs:element name="MaxValue" type="xs:decimal"/>
<xs:element name="MinNumberOfChars" type="nonNegativeInt"/>
<xs:element name="MinValue" type="xs:decimal"/>
<xs:element name="MultiLineHelp" type="xs:string"/>
<xs:element ref="OneLineHelp"/>
<xs:element name="Prominence" type="nonNegativeInt"/>
<xs:element name="SourceTable" type="lowerIdentifier"/>

<xs:element name="Type">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="boolean"/>
      <xs:enumeration value="choice"/>
      <xs:enumeration value="date"/>
      <xs:enumeration value="file"/>
      <xs:enumeration value="float"/>
      <xs:enumeration value="int"/>
      <xs:enumeration value="text"/>
      <xs:enumeration value="tableEntry"/>
      <xs:enumeration value="timeOfDay"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

</xs:choice>

  <xs:attribute name="name" type="xs:string"/>
</xs:complexType>

<!-- END declarations relevant to Parameters and Questions -->

<!-- What can go inside a CommandGroup declaration -->
<xs:element name="CommandGroup">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">

      <xs:element name="Label" type="xs:string"/>
      <xs:element name="Member" type="xs:string"/>

    </xs:choice>
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>

```

```

</xs:element>

<!-- What can go inside a Table declaration -->
<xs:element name="Table">
  <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">

      <xs:element name="Browsable" type="joharBoolean"/>
      <xs:element name="DefaultColumnNames" type="xs:string"/>
      <xs:element name="DefaultHeading" type="xs:string"/>
      <xs:element name="Label" type="xs:string"/>

    </xs:choice>
    <xs:attribute name="name" type="xs:string"/>
  </xs:complexType>
</xs:element>

<!-- Declarations common to many of the above -->

<xs:element name="BriefHelp" type="xs:string"/>

<xs:element name="OneLineHelp" type="xs:string"/>

<xs:simpleType name="joharBoolean">
  <xs:restriction base="xs:normalizedString">
    <xs:enumeration value="yes"/>
    <xs:enumeration value="no"/>
    <xs:enumeration value="true"/>
    <xs:enumeration value="false"/>
    <xs:enumeration value="Yes"/>
    <xs:enumeration value="No"/>
    <xs:enumeration value="True"/>
    <xs:enumeration value="False"/>
    <xs:enumeration value="YES"/>
    <xs:enumeration value="NO"/>
    <xs:enumeration value="TRUE"/>
    <xs:enumeration value="FALSE"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="positiveInt">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1"/>
  </xs:restriction>
</xs:simpleType>

```

```
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="nonNegativeInt">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="lowerIdentifier">
    <xs:restriction base="xs:string">
      <xs:pattern value="\s*[a-z]([a-zA-Z0-9])*\s*" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="upperIdentifier">
    <xs:restriction base="xs:string">
      <xs:pattern value="\s*[A-Z]([a-zA-Z0-9])*\s*" />
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

Appendix D

Report On The Review Of Johar-Related Documents

The outcome of our review is presented below. For clarity, classes in the *johar.idf* package and the XML Schema Document (*johar.xsd*) are italicized in the table, while attributes are printed in Typewriter font. Also, “IDF Format Specification” is abbreviated as “IDF Format Spec.”, while “Interface Interpreter Specification” is abbreviated as “IntI Spec.”.

Section	Attribute	Sentence	Violation
2.1	CommandGroup	A Command cannot appear as a member of more than one CommandGroup.	There is no statement in <i>IdfCommandGroup</i> that checks for this condition. Members of a CommandGroup are only added to a CommandGroup vector without first checking if a member already exists in other vectors.
2.1	InitializationMethod	The name of the application engine method used to initialize the engine.	This attribute is not available in current <i>johar.xsd</i> version, and not captured in <i>Idf</i> .

2.2	CommandMethod	<p>IDF Format Spec.: Single-stage Commands (commands that have no Stage attribute) can contain any of the attributes described below in Section 2.3, Sub-attributes of Stage and Single-Stage Commands.</p> <p>Intl Spec.: The Intl must call the CommandMethod for the command.</p>	<p>These sentences imply that the CommandMethod should appear at most once within a Command attribute (or a single-stage command), and not within the Stage attribute (if any) in the Command. In the Johar source code, the value of a CommandMethod is extracted in the <i>IdfStage</i> class. I think the current <i>IdfStage</i> was written with the mindset of single-staged commands. Each instance of <i>IdfStage</i> represents each stage in the Command attribute. Thus, an issue will arise if an application developer puts a CommandMethod in one or more Stages, which will be valid according to the present code. Also, the <i>johar.xsd</i> supports each Stage to have a CommandMethod.</p>
2.2	Label	Same as the sentence in the IDF Format Spec. as stated in the 3rd row above	Same violation as the third row above, except that I refer to the Label attribute instead of CommandMethod.
2.2	Prominence	Default value: 2000	Default value specified in <i>IdfCommand</i> is 1000.

2.2	Question	Same as the sentence in the IDF Format Spec. as stated in the 3rd row above	Can the Question attribute appear in each Stage ? If no, then <i>IdfStage</i> and <i>johar.xsd</i> implemented this concept wrongly. Otherwise, they are correct.
2.2	Stage	The names of the Parameters in all of the Stages in a Command must be disjoint.	The current implementation of <i>IdfStage</i> only extracts and stores the names of the parameters for each stage. This condition is not explicitly tested for.
2.3	ParameterCheckMethod		In the current <i>IdfStage</i> , there is a mistake in this attribute's name while extracting its value. The name is specified as "parameterCheckMethod", instead of ParameterCheckMethod . Since XML is case-sensitive, the value in the ParameterCheckMethod element will not be read.
2.3	Parameter ParameterCheckMethod	These sub-attributes can appear inside a Stage in a Command	In the <i>johar.xsd</i> , the attributes of a Stage are: CommandMethod , Label , Parameter , ParameterCheckMethod , and Question . This is a violation.
2.2 and 2.4	BriefHelp OneLineHelp		There is no conditional expression testing for the maximum length of BriefHelp and OneLineHelp in both <i>IdfCommand</i> and <i>IdfParameter</i> .

2.4	FileConstraint	Multiplicity: For parameters of type file, 0 or 1.	In <i>IdfParameter</i> , the multiplicity specified is 1.
2.4	FileConstraint	Accepted values: mustExist: The file must exist at the time the command is issued. mustBeReadable: The file must exist and be readable by the application at the time the command is issued. mustNotExistYet: The file must not exist at the time the command is issued. none: No constraint.	Value none is not part of accepted values specified in <i>johar.xsd</i> .
2.4	Label	Default value: Derived from the name of the command using standard camel-case translation.	The default value is derived from the name of the parameter, as specified in <i>IdfParameter</i> .
2.4	MaxNumberOfChars MaxNumberOfReps	Value: An integer literal greater than or equal to 1; or the lower-case identifier unlim	There is no statement in the <i>IdfElement</i> class to check for unlim in order to replace it with, for instance, "Integer.MAX_VALUE".
2.4	MaxNumberOfLines	The maximum number of lines that can be entered by the user as the value of a text parameter.	There is no statement that extracts the value of MaxNumberOfLines (and that specifies the default value) in <i>IdfParameter</i> .
2.4	MinNumberOfReps	MinNumberOfReps must be less than or equal to MaxNumberOfReps.	This condition is not checked for in the <i>Idf*</i> classes.

2.4	MinValue	MinValue must be less than or equal to MaxValue.	This condition is not checked for in the Idf* classes.
2.4	Prominence	Default value: 2000	Default value specified in <i>IdfParameter</i> is 1000.
2.4	RepsModel	Default value: set	Default value specified in <i>IdfParameter</i> is the empty string.
2.4	AskIfMethod	Multiplicity: exactly 1	The maximum number of occurrence specified in <i>johar.xsd</i> is “unbounded”.
2.5	Question	The following attributes of <i>Parameter</i> are not acceptable sub-attributes of any <i>Question</i> : <ul style="list-style-type: none"> – MaxNumberOfReps – MinNumberOfReps – ParentParameter – ParentValue – RepsModel 	Both <i>johar.xsd</i> and <i>IdfQuestion</i> violate this rule. In <i>johar.xsd</i> , “questionType” simply extended “paramType” without constraints, while <i>IdfQuestion</i> extended <i>IdfParameter</i> without any restriction or constraints.
2.6	Label	Label: The text describing the <i>CommandGroup</i> in the interface.	Label is not part of the attributes captured by both <i>IdfCommandGroup</i> and <i>johar.xsd</i> .
2.7	Label	Label: The text describing the <i>Table</i> in the interface.	Label is not part of the attributes captured by both <i>IdfTable</i> and <i>johar.xsd</i> .

2.7	DefaultHeading	<p>Default value: The Label of the table.</p> <p>This attribute gives the heading that will be associated with the table if no heading is set by the application engine. If no DefaultHeading attribute is given, then the heading will be derived from the name of the table using standard camel-case translation.</p>	<p>An empty string is specified as the default value in <i>IdfTable</i>.</p>
-----	----------------	--	--

Other Observations:

- `MinNumberOfChars` is not specified in the IDF Format Specification document, but captured in *IdfParameter*.
- `Mnemonic` is not specified in the IDF Format Specification document, but captured in both *johar.xsd* and *IdfCommandGroup*.
- The maximum number of occurrence specified for the `DefaultHeading` attribute of `Table` in *johar.xsd* is unbounded. Thus, there would not be any error if an application developer specifies more than one `DefaultHeading` attribute in the IDF.
- *IdfParameter* is used capture the sub-attributes of both `Parameter` and `Question` (except `AskIfMethod`). Thus, some messages are sent to the user which may be ambiguous (depending on the user's cognitive skills). An example of such messages is "Parameter/Question has no ParentParameter attribute". Since a `Question` cannot have a `ParentParameter` attribute, there is no need for it to appear in the message.

Appendix E

“Star” Interface Interpreter: Requirements Specification of the Star GUI

The diagram(s) illustrating each section of this document will be found in Chapter 4 of this thesis.

E.1 The Main Panel

The Main Panel is displayed when the application is started, and remains visible throughout the lifetime of the application.

- (1) At the top is the *Menu Bar*. This shows one menu for each command group in the application, and one menu named “Star” (for services provided by the Star interpreter for all Johar applications). We refer to the non-Star menus as “application menus”.
- (2) Each application menu is labelled in the Menu Bar by the name of the command group.
- (3) The menu items in each application menu are the commands that are in that command group.
- (4) An application menu item is clickable (not greyed out) if the corresponding command is *active*.
- (5) A command is *active* at a given point if either it has no `ActiveIfMethod`, or it has an `ActiveIfMethod` and that method returns `true`.
- (6) On the left is the *Text Display Area*. This shows all text messages sent by the application engine using `showText` that are not displayed through some other means.

- (7) On the right is the *Table Area*. This is a tabbed pane.
- (8) In the Table Area, there is one tab corresponding to every revealed Table currently in the application.
- (9) The application decides which Tables to conceal and reveal.
- (10) If, during the processing of a command, the application engine sets the top table, then immediately after the command has finished processing, the topmost table in the Table Area is the last table to have been set as the top table.
- (11) If, during the processing of a command, the application engine does not set the top table, then immediately after the command has finished processing, the topmost table in the Table Area remains unchanged from the time that the command was issued.
- (12) Notwithstanding the table set as the topmost table by the app engine, the user can (between issuing one command and issuing the next command) click on a table tab in order to move that table to the top.
- (13) At the bottom is the *Status Bar*. This shows low-prominence messages sent by the application engine using `showText`.
- (14) On the Star menu, there is one menu item, “Help”.
- (15) When the “Help” menu item is selected, the Help Box is displayed (see below).

E.2 The Command Dialog Box

- (1) At the top is the Label attribute of the command.
- (2) Under that is one section for every *queryable parameter* (see below) in the current stage.
- (3) Many commands have only one stage. However, if there is more than one stage, only the queryable parameters in the current stage will appear in the parameter section.
- (4) There is one section of the box for each queryable parameter in the current stage.
- (5) At the bottom are two to four buttons: `Cancel`, `Previous`, `Next`, and `OK`. `Cancel` and `OK` always appear; `Previous` and `Next` do not always appear.
- (6) The *Cancel button* always appears and is always enabled.

- (7) The Cancel button is on the left-hand side of the dialog box.
- (8) The *Previous button*:
 - (i) Appears if there is more than one *queryable stage* (see below).
 - (ii) Is enabled if the current stage is not the first queryable stage.
- (9) The *Next button*:
 - (i) Appears if there is more than one queryable stage.
 - (ii) Is enabled if the current stage is not the last queryable stage.
- (10) The *OK button*:
 - (i) Always appears.
 - (ii) Is enabled if there are no *incomplete stages* (see below).
- (11) The OK button is on the right-hand side of the dialog box.
- (12) A parameter is *inactive* if both (a) it has a `ParentParameter`, and (b) the current value of the `ParentParameter` is not the `ParentValue` for this parameter.
- (13) A *queryable parameter* is any parameter that is not a `tableEntry` parameter with a browsable `SourceTable`.
- (14) A *queryable stage* is a stage that contains some queryable parameter.
- (15) An *incomplete stage* is a stage that contains some parameter such that: (a) the parameter is not inactive, (b) there is no `DefaultValue` or `DefaultValueMethod` for the parameter, and (c) the current number of repetitions for which the user has selected a value is less than the `MinNumberOfReps` for the parameter.
 (That is, any repetitions that the user has not filled in and that do not have a default value should be interpreted as repetitions that the user does not want to give to the application engine. If the number of the remaining repetitions is less than the `MinNumberOfReps`, then the user still needs to fill in more values.)

E.3 Parameter Section of the Command Dialog Box

Every queryable parameter in the current stage is represented by a section of the command dialog box.

- (1) On the left is the `Label` attribute of the parameter, at the top of the left-hand side, followed by a colon.
- (2) On the right is one subsection for each repetition of the parameter, and sometimes a small *Add Another* button.
- (3) Initially (before any user interaction), the number of repetition sections will be equal to $\text{maximum}(1, m)$, where m is the value of `MinNumberOfReps` for the parameter.
- (4) The *Add Another* button:
 - (i) Has the `ToolTip` “Add another”.
 - (ii) Is a small button with a plus sign (“+”) in it.
 - (iii) Appears only if `MinNumberOfReps` is not equal to `MaxNumberOfReps` for this parameter.
 - (iv) Is enabled only if the current number of repetitions of the parameter is less than `MaxNumberOfReps` for this parameter.
 - (v) Is on the left-hand side of the right-hand half of the parameter section.
- (5) The entire section corresponding to the parameter is greyed out if the parameter is inactive.

E.4 Repetition Section of the Parameter Section

Every repetition of a queryable parameter in the current stage is represented by a section of the command dialog box.

- (1) On the left is a widget which the user can use to select the value of the repetition. This widget will be different for different types of parameters. For instance, for a `boolean` parameter, it may be a widget consisting of two radio buttons labelled “Yes” and “No”, whereas for a `text` parameter it may be a text area.
- (2) On the right of the section is zero to three small buttons with icons in them.
- (3) The *Move Up* button:
 - (i) Is a small button with an up-arrow in it.
 - (ii) Has the `ToolTip` “Move up”.

- (iii) Appears only if `MaxNumberOfReps` is not equal to 1 for this parameter, and `RepsModel` for this parameter is `sequence`.
 - (iv) Is enabled only if this is not the first repetition (the topmost repetition).
- (4) The `Move Down` button:
- (i) Is a small button with a down-arrow in it.
 - (ii) Has the `ToolTip` “Move down”.
 - (iii) Appears only if `MaxNumberOfReps` is not equal to 1 for this parameter, and `RepsModel` for this parameter is `sequence`.
 - (iv) Is enabled only if this is not the last repetition (the bottommost repetition).
- (5) The `Delete` button:
- (i) Is a small button with an X (“x”) in it.
 - (ii) Has the `ToolTip` “Delete”.
 - (iii) Appears only if `MinNumberOfReps` is not equal to `MaxNumberOfReps` for this parameter.
 - (iv) Is enabled only if the current number of repetitions of the parameter is more than $\text{maximum}(1, m)$, where m is the value of `MinNumberOfReps` for this parameter.

E.5 Question Dialog Box

The Question Dialog Box is used to ask `Questions`.

- (1) On the top is the `Label` attribute of the question.
- (2) Below that is a parameter occurrence selection widget, such as would appear in the `Command Dialog Box` for one repetition of a parameter.
- (3) On the lower left is a `Cancel` button.
- (4) On the lower right is an `OK` button.

E.6 Help Box

The Help Box has three states: the Top-Level state, the Command state, and the Parameter/Question state. It may be wise to put the contents of the window into a `JScrollPane`, since the size of the contents will sometimes depend on the size of the `MultiLineHelp` attributes of given commands and parameters.

E.6.1 Top-Level State

The Help Box begins in the Top-Level state.

- (1) At the top is the word `Commands`.
- (2) Below that is a table (perhaps a `JTable`) with one row for every command in the application. The table has two columns.
- (3) Each row of the table contains the `Label` attribute of the command in the first column. This should be displayed in a way that makes it clear that it is “clickable”.
- (4) Each row of the table contains, in the second column, the `OneLineHelp` attribute of the command.
- (5) At the bottom right is a button labelled `OK`. If the user clicks this, the box should be disposed.

E.6.2 Command State

If a user selects a command label when the help box is in the Top-Level state, it moves to the Command state.

- (1) At the top is the `Label` attribute of the `CommandGroup` that the command is in, an arrow, and the `Label` attribute of the command. (This gives an indication of where to find the command on the menus.)
- (2) Below that is a section containing the `MultiLineHelp` attribute of the command.
- (3) Below that is a table (perhaps a `JTable`) with one row for each parameter in the command. This parameter table has two columns.
- (4) Each row of the parameter table contains the `Label` attribute of the parameter in the first column. This should be displayed in a way that makes it clear that it is “clickable”.

- (5) Each row of the parameter table contains, in the second column, the `OneLineHelp` attribute of the parameter.
- (6) If there are any questions in the command, then below the parameter table, there is a `JLabel` reading “Questions that might be asked:”, and a table with one row for each question in the command. This question table also has two columns.
- (7) Each row of the question table contains the `Label` attribute of the question in the first column. This should be displayed in a way that makes it clear that it is “clickable”.
- (8) Each row of the question table contains, in the second column, the `OneLineHelp` attribute of the question.
- (9) At the bottom left is a button labelled Back. If the user clicks this, the box should go back to the Top-Level state.
- (10) At the bottom right is a button labelled OK. If the user clicks this, the box should be disposed.

E.6.3 Parameter/Question State

If a user selects a parameter or question label when the help box is in the Command state, it moves to the Parameter/Question state.

- (1) At the top is the `Label` attribute of the `CommandGroup` that the command is in, an arrow, and the `Label` attribute of the command. (This provides continuity with the Command state.)
- (2) Below that is a section containing the text “Parameter:”, followed by the `Label` attribute of the parameter.
- (3) Below that is a section containing the `MultiLineHelp` attribute of the parameter.
- (4) At the bottom left is a button labelled Back. If the user clicks this, the box should go back to the Command state with the selected command.
- (5) At the bottom right is a button labelled OK. If the user clicks this, the box should be disposed.
- (6) For a Question, the Help Box is identical, except that the text “Parameter:” reads “Question:” instead.

Appendix F

“Star” Interface Interpreter: Requirements Specification (Behaviour)

F.1 Top-Level Behaviour

- (1) Read the IDF.
- (2) Create the Main Panel.
 - (i) Every `CommandGroup` should correspond to a menu on the menu bar.
 - (ii) There should also be the menu “Star”, as the rightmost menu on the menu bar.
 - (iii) The menu item for a command should be the `Label` of the command, followed by “. . .” if selecting the command will result in a `Command Dialog` being created (see below).
- (3) Initialize the `Text Area` to empty.
- (4) Create the `GemSetting` and validate it.
- (5) Call the application engine’s initialization method.
- (6) Refresh the tables (see below).
- (7) Append a horizontal line (this could be just a line of minus characters) to the bottom of the `Text Area`. (This will separate any initial message from the application engine from the messages that are responses to the commands.)
- (8) Call the `ActiveIfMethod` method of every command, and make each menu item of each menu active or inactive (greyed out) according to the result of the `ActiveIfMethod` method of the command.

F.2 Selecting a Command from a Menu

When the user selects a command from a menu:

- (1) Set a String field `lastDisplayedText` to the empty string. (Rationale: this will be for displaying the last text that the program shows the user. See below.)
- (2) If any table selection is *incomplete* (see below) for the command, then pop up a dialog box with an “OK” button telling the user which tables they have to select rows in in order to issue the command, and a description of the bounds of the number of rows to select (e.g., “at least 2”, “between 1 and 5”).
- (3) Otherwise, if no stage in the command has a queryable parameter (i.e., there are no parameters to any of the stages of the command, or all parameters in all stages of the command are `tableEntry` parameters that are not queryable):
 - (i) Call `gemSetting.selectCurrentCommand(cmdName)`.
 - (ii) Call `gemSetting.selectCurrentStage(0)`.
 - (iii) For each stage *i* in the command, perform the Stage Loop from the Intf Specification document.
 - (iv) Perform the Question-and-Wrapup Procedure with parameters 0 and `true`. (This indicates that the Question-and-Wrapup Procedure should start with question number 0, and should wrap up the command if any questions are cancelled.)
- (4) Otherwise (i.e., if no table selection is incomplete for the command, but there are queryable parameters):
 - (i) Call `gemSetting.selectCurrentCommand(cmdName)`.
 - (ii) Create the Command Dialog for the command. (The Command Dialog will take over the rest of the processing of the command.)

A table selection is *incomplete for a given command* if it contains at least one stage with a parameter with the following characteristics:

- (1) The parameter is of type `tableEntry`;
- (2) The `SourceTable` of the parameter is `Browsable`;
- (3) The parameter does not have a `ParentParameter`; and
- (4) The `MinNumberOfReps` for the parameter is greater than the number of rows that are currently selected in the `SourceTable` of the parameter.

F.3 Question-and-Wrapup Procedure

The Question-and-Wrapup Procedure takes two parameters: the number of the question (`int questionNumber`), and an indication of whether the command should be wrapped up if a question is cancelled (`boolean wrapUpIfCancelled`).

- (1) If `questionNumber` is greater than or equal to the number of questions in the command, then this indicates that everything is OK for the command to be actually executed.
 - (i) Call the command's `CommandMethod`.
 - (ii) Execute the Command Wrapup Procedure (see below) with parameter `false` (indicating command was not cancelled).
 - (iii) Return.

Otherwise, continue with the below steps.

- (2) Call the `AskIfMethod` for the question.
- (3) If the `AskIfMethod` returns `false`, then:
 - (i) Execute the Question-and-Wrapup Procedure (recursively), using `questionNumber + 1` and `wrapUpIfCancelled` as parameters.
 - (ii) Return.

Otherwise, continue with the below steps.

- (4) If the question has a `DefaultValue`, set the value of (the response to) the question to the default value.
- (5) Otherwise, if the question has a `DefaultValueMethod`, then call it and set the value of (the response to) the question to the default value.
- (6) Create a question dialog box for the question.

F.3.1 Question Dialog Cancel Button Action

If the user presses the Cancel button on the Question Dialog Box:

- Dispose of the question dialog box.
- If `wrapUpIfCancelled`, then execute the Command Wrapup Procedure with parameter `true`, indicating the command was cancelled.

F.3.2 Question Dialog OK Button Action

If the user presses the OK button on the Question Dialog Box:

- (1) Validate the current value of (the response to) the question as if it is a parameter, as indicated in requirements 26-51 of the IntI specification. (A question with no default value, for which the user has not selected a value, should be interpreted as an invalid response.)
- (2) If the value of (the response to) the question does not pass validation:
 - (i) Present an error message to the user in a dialog box. The error dialog box should have just an “OK” button.
 - (ii) When the user presses OK, dispose of the error dialog box. (Rationale: This will have the effect of returning control to the question dialog box, so that the user can select another value and press OK again or Cancel.)
- (3) Otherwise:
 - (i) Load the current value of (the response to) the question into the Gem.
 - (ii) Dispose of the question dialog box.
 - (iii) Execute the Question-and-Wrapup Procedure (recursively), using `questionNumber + 1` and `wrapUpIfCancelled` as parameters.

F.4 Command Wrapup Procedure

The Command Wrapup Procedure is called after the user selects a command from the menu and some processing has been done. It takes one boolean parameter, called `commandWasCancelled`. `commandWasCancelled` should be `false` only if the application engine method was called before the command wrapup procedure was called. If it is `true`, this means that the command was cancelled in some way by the user or by Star itself.

- (1) If there is a Command Dialog, dispose of it.
- (2) If `commandWasCancelled` is `false`:
 - (i) Determine whether the application should quit, by checking the `QuitAfter` attribute and/or calling the `QuitAfterIfMethod` of the command.
 - (ii) If the application should quit:

- (a) If `lastDisplayedText` is non-null and not the empty string, then create a dialog box containing `lastDisplayedText` and an “OK” button, and display it. Wait for the user to click “OK”, and then delete the dialog box. (Rationale: if the command executed a `showText` method and then `Star` exits, it may exit before the user has had time to read the text.)
 - (b) Exit the application, e.g. using `System.exit(0)`.
- (iii) Otherwise:
- (a) Append a horizontal line to the Text Display Area, in order to separate the last command’s output from the output of any future commands.
 - (b) Refresh the tables (see below).
- (3) Call the `ActiveIfMethod` of every command, and make each menu item of each menu active or inactive (greyed out) according to the result of the `ActiveIfMethod` of the command.

F.5 Refreshing the Tables

To refresh the tables:

- (1) For each browsable table currently being displayed in the Table Area that is now hidden, delete the corresponding tab.
- (2) For each browsable table not currently being displayed in the Table Area that is now non-hidden, create a tab in the Table Area. (After this happens, it should be the case that the only tables with tabs in the Table Area are non-hidden tables.)
- (3) For each non-hidden table which has been updated since the last command execution terminated, update the data on the tab to reflect the contents of the table.
- (4) If there is a top table now, then place that table’s tab on top in the tab pane.

F.6 The ShowTextHandler

The `ShowTextHandler` for `Star` handles text according to the prominence of the text.

- (1) 0-1999: For each line of text in the message:

- (i) Truncate the line of text, if necessary, to fit in the Status Bar. (Rationale: it should fit all right, but just in case it doesn't, we can show part of it. Because it is low-prominence, it seems OK to just show part of it.)
- (ii) Set the Status Bar to the text.

[Note: This will cause the last line of the most recent message to overwrite anything that was in the Status Bar before. This is OK because these messages are “low prominence”.]

- (2) 2000-2999: Append the text to the text being displayed in the Text Display Area. Append the text also to `lastDisplayedText`.
- (3) 3000 and higher: Create a dialog box containing the text and an “OK” button, and display it. When the user clicks the “OK” button, the box should be deleted.

F.7 The Command Dialog Box

F.7.1 Creating the Command Dialog Box

- (1) Call `gemSetting.selectCurrentStage(0)`.
- (2) Perform an Initialize Stage procedure.
- (3) While the current stage has no queryable parameters, perform a Next Stage procedure (see below).
- (4) Update the dialog box to reflect the current stage, as described in the Star GUI Specification document.

F.7.2 Initialize Stage Procedure

- (1) If the current stage is a stage that has never been initialized so far, then for each parameter in the current stage:
 - (i) If the parameter has a `DefaultValue`, set the value of the parameter to the default value.
 - (ii) Otherwise, if the parameter has a `DefaultValueMethod`, then call it and set the value of the parameter to the default value.

F.7.3 Next Stage Procedure

- (1) Perform a Wrap Up Stage procedure.
- (2) If the Wrap Up Stage procedure returns `true`, then (assuming that the current stage is in the variable `currentStage`):
 - (i) Set the current stage to `currentStage+1`.
 - (ii) Call `gemSetting.selectCurrentStage(currentStage)`.
 - (iii) Perform an Initialize Stage procedure (see above).

F.7.4 Previous Stage Procedure

- (1) Perform a Wrap Up Stage procedure.
- (2) If the Wrap Up Stage procedure returns `true`, then (assuming that the current stage is in the variable `currentStage`):
 - (i) Set the current stage to `currentStage-1`.
 - (ii) Call `gemSetting.selectCurrentStage(currentStage)`.
 - (iii) Perform an Initialize Stage procedure (see above).

F.7.5 Wrap Up Stage Procedure

This procedure takes no parameters. It returns `true` if all the tasks that the user had to do in the current stage have been completed, and it returns `false` otherwise.

- (1) Validate the current values of the current repetitions of the parameters, as indicated in requirements 26-51 of the IntI specification. (A parameter with no default value, for which the user has not selected a value, should be interpreted as not adding a valid repetition of the parameter.)
- (2) If any parameter or parameter repetition does not pass validation, then:
 - (i) Collect information in a string about anything that does not pass validation.
 - (ii) Present that information to the user in a dialog box. The dialog box should have just an “OK” button.
 - (iii) When the user clicks OK:

- (a) If the current stage has no queryable parameters, then perform the Command Wrapup procedure, with the parameter `true`. (Rationale: This might happen in some obscure situations, such as when a `tableEntry` parameter has a parent parameter which gets set to the parent value. In these situations, there is nothing we can do but cancel the command. The `true` parameter to the Command Wrapup procedure indicates it has been cancelled.)
- (iv) Return `false`. (Rationale: the user has not completed everything they have to do for this stage.)
- (3) (Otherwise:) Load the current values of the current repetitions of the parameters for the current stage into the Gem.
- (4) Call the `ParameterCheckMethod` of the current stage, if it has one.
- (5) If the `ParameterCheckMethod` exists and returns a non-null, non-empty string:
 - (i) Show the user the string in a popup with an “OK” button.
 - (ii) When the user clicks OK:
 - (a) If the current stage has no queryable parameters, then perform the Command Wrapup procedure, with the parameter `true`.
 - (iii) Return `false`.
- (6) Otherwise, return `true`.

F.7.6 Next Button Action

When the user presses the Next button (if it is not greyed out):

- (1) Perform a Next Stage procedure (see above).
- (2) While the current stage has no queryable parameters, perform a Next Stage procedure (see above).
- (3) Update the dialog box to reflect the current stage, as described in the Star GUI Specification document.

F.7.7 Previous Button Action

When the user presses the Previous button (if it exists and is not greyed out):

- (1) Perform a Previous Stage procedure (see above).
- (2) While the current stage has no queryable parameters, perform a Previous Stage procedure (see above).
- (3) Update the dialog box to reflect the current stage, as described in the Star GUI Specification document.

F.7.8 OK Button Action

When the user presses the OK button (if it exists and is not greyed out):

- (1) Perform the Wrap Up Stage procedure.
- (2) If the Wrap Up Stage procedure returns `false`, then return.
- (3) For each stage `i` in the current command, from stage 0 to the last stage:
 - (i) Call `gemSetting.selectCurrentStage(i)`.
 - (ii) Perform the Initialize Stage procedure.
 - (iii) Perform the Wrap Up Stage procedure.
 - (iv) If the Wrap Up Stage procedure returns `false`, then return.

(Rationale: there may have been some previous stages which have become invalid as a result of the current stage; also, there may be stages after the current stage with parameters whose values have not been loaded yet.)

- (4) Perform the Question-and-Wrapup Procedure with parameters 0 and `false`.

F.7.9 Cancel Button Action

When the user presses the Cancel button:

- (1) Perform the Command Wrapup procedure, with the parameter `true`.

F.8 The Parameter Section

F.8.1 Add Another Button Action

When the user clicks the Add Another button:

- Add another repetition section to the parameter section for the parameter, at the bottom of the list. You may need to add a repetition to the model as well. (Rationale: This is safe because if the Add Another button exists and is not greyed out, then we are not at the maximum number of repetitions for the parameter yet.)

F.8.2 Move Up Button Action

When the user clicks the Move Up button for repetition k :

- Exchange the value in repetition k with the value in repetition $k - 1$. This should be done in the model so that the changes will be automatically reflected in the GUI. (Rationale: If the Move Up button exists and is not greyed out, then there is another repetition above the k th repetition.)

F.8.3 Move Down Button Action

When the user clicks the Move Down button for repetition k :

- Exchange the value in repetition k with the value in repetition $k + 1$. This should be done in the model so that the changes will be automatically reflected in the GUI. (Rationale: If the Move Down button exists and is not greyed out, then there is another repetition below the k th repetition.)

F.8.4 Delete Button Action

When the user clicks the Delete button for repetition k :

- Delete repetition k . This should be done in the model as well. (Rationale: If the Delete button exists and is not greyed out, then we are not at the minimum number of repetitions for the parameter yet.)

Appendix G

Some Source Code of “Star” Implementation

G.1 The *Star* class

This is the class that launches the Star interface interpreter.

```
package johar.interfaceinterpreter.star;

import johar.idf.*;
import johar.utilities.TextInputValidator;
import johar.gem.*;

/**
 * The Star Interface Interpreter's main class.
 */

public class Star implements ShowTextHandler {
    private Idf idf;
    private GemSetting gem;
    private StarWindow starFrame;
    private String starTitle;
    private CommandMenu cmdMenu;
    private CommandMenuItem cmdMenuItem;
    private StatusBar statusBar;
    private TextDisplayArea textArea;
    private ScrollingWidget scrollWidget;
    private CommandController cc;
    private TableArea tableArea;

    /**
```

```

* Star Constructor: implements the top-level behaviour
* specification.
* @param idfName
* Name of the IDF
*/
public Star(String idfName) {
    try {
        idf = Idf.idfFromFile(idfName); //Read the IDF

        //Check the IDF version
        if (!idf.getIdfVersion().equals("1.0")){
            showText("The specified IDF Version is not " +
                "supported. The only IDF version allowed "
                + "is 1.0.", 3000);
            System.exit(1);
        }

        //Create the GemSetting and validate it
        gem = GemFactory.newGemSetting(idf, this);
        gem.validate();

        //Create the Command Controller
        cc = new CommandController(gem, idf, this);
        createStarGUI(); //Create the Main Panel and
            //the Text Display Area

        //Call the application engine's initialization
        //method and refresh the tables
        gem.initializeAppEngine();

        createTableArea(); //Creates the Table Area
            //and refreshes the tables.

        //Append a horizontal line to the bottom of
        //the Text Area.
        starFrame.appendHorizontalLineToTextArea();
    } catch (IdfFormatException ex) {
        //Show message to user
        showText(ex.getMessage(), 3000);
    } catch (Exception e) {
        MessageDialog.showError("An error occurred " +
            "while launching" + " the application." +
            " [Error Details: " + e.getMessage()
            + "]");
    }
}

```

```

}

/*
 * Create the Menus and Call the ActiveIfMethod method
 * of every command, and make each menu item of each
 * menu active or inactive (greyed out) according to
 * the result of the ActiveIfMethodmethod of the
 * command.
 */
private void createMenus() {

    /* The IdfAnalyzer provides access to various
     * information about the specified IDF e.g.
     * list of queryable/non-queryable params
     * in a command/stage, list of questions in a
     * command, list of tables, list of browsable
     * tables, etc. */
    IdfAnalyzer idfAnalyzer = new IdfAnalyzer(idf);

    IdfCommandGroup cmdGroup;
    IdfCommand cmd;
    int numOfCmdGrps = idf.getNumCommandGroups();
    int numOfCmds = idf.getNumCommands();
    int numOfMembers = 0;
    boolean isActive;

    for (int i = 0; i < numOfCmdGrps; i++) {
        cmdGroup = idf.getCommandGroupNumber(i);
        cmdMenu = new CommandMenu(cmdGroup.getCommandGroupName(),
            cmdGroup.getLabel());
        starFrame.addMenu(cmdMenu);
        numOfMembers = cmdGroup.getNumMembers();
        for (int j = 0; j < numOfMembers; j++) {
            for (int k = 0; k < numOfCmds; k++) {
                cmd = idf.getCommandNumber(k);
                isActive = gem.methodIsActive(cmd.getCommandName());
                if (cmdGroup.getMemberNumber(j)
                    .equals(cmd.getCommandName())) {
                    cmdMenuItem = new CommandMenuItem(cmd.getCommandName(),
                        cc, false);
                    idfAnalyzer.setCurrentCommand(cmd.getCommandName());

                    if (idfAnalyzer.hasQueryableParams())
                        cmdMenuItem.setText(cmd.getLabel() + "...");
                    else

```

```
        cmdMenuItem.setText(cmd.getLabel());

        cmdMenuItem.setEnabled(isActive);
        starFrame.addMenuItem(cmdMenuItem,
            cmdGroup.getCommandGroupName());
        break;
    }
}
}

// Add the Star menu
cmdMenu = new CommandMenu("star", "Star");
starFrame.addMenu(cmdMenu);

// Add the Help menu item to Star
cmdMenuItem = new CommandMenuItem("help", cc, true);
cmdMenuItem.setText("Help");
cmdMenuItem.setEnabled(true);
starFrame.addMenuItem(cmdMenuItem, "star");
}

// Create the Status bar
private void createStatusBar() {
    statusBar = new StatusBar();
    starFrame.setStatusBar(statusBar);
}

// Create the Text Display Area
private void createTextDisplayArea() {
    textArea = new TextDisplayArea();
    scrollWidget = new ScrollingWidget(textArea);
    scrollWidget.setName("textDisplayArea");
    starFrame.setTextArea(scrollWidget);
}

// Create the Table Area
private void createTableArea() {
    tableArea = new TableArea(idf, gem, cc);
    tableArea.setName("tableDisplayArea");
    starFrame.setTableArea(tableArea);
}

// Set the title of the Star GUI
private void setStarTitle() {
```

```
        starTitle = idf.getApplication();
        starFrame.setTitle(TextInputValidator
            .titleCaseTranslation(starTitle));
    }

    // Create the Star GUI
    private void createStarGUI() {
        starFrame = new StarWindow();
        createMenus();
        createTextDisplayArea();
        createStatusBar();
        setStarTitle();
    }

    /**
     * Show the Star GUI
     */
    public void show() {
        if (starFrame != null)
            starFrame.setVisible(true);
    }

    /**
     * Get a Star Window (or Star Frame) instance
     *
     * @return Star Window instance
     */
    public StarWindow getStarFrame() {
        return starFrame;
    }

    /**
     * Implements the showText method of the ShowTextHandler interface
     */
    public void showText(String text, int priorityLevel) {
        if (priorityLevel >= HIGHPRIO_LEVEL) {
            MessageDialog.show(text);
        } else if (priorityLevel >= RESULT_LEVEL) {
            textArea.setContent(text);
            cc.lastDisplayedText += text;
        } else if (priorityLevel >= STATUS_LEVEL) {
            starFrame.setStatusMessage(text);
        } else if (priorityLevel >= DEBUG_LEVEL) {
            System.out.println(text);
        } else {
```

```
        System.out.println("Priority Level in
            Show Text Handler is not valid.");
    }
}

/**
 * The main method.
 *
 * @param args
 * Command-Line arguments
 */
public static void main(String args[]) {
    Star star = new Star(args[0]);
    star.show();
}
}
```

G.2 The *CommandDialog* class

This class creates the Command Dialog Box.

```
package johar.interfaceinterpreter.star;

import java.awt.Component;
import java.awt.Container;
import java.awt.Toolkit;
import java.util.TreeMap;
import java.awt.Dialog;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import javax.swing.JDialog;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JComponent;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;

import johar.gem.GemSetting;
import johar.idf.Idf;
import johar.idf.IdfCommand;
import johar.idf.IdfParameter;
```



```

/**
 * The Command Dialog Box's creator.
 *
 */
public class CommandDialog extends JDialog implements WindowListener {
    private Idf _idf;
    private GemSetting _gem;
    private CommandController _cc;
    private IdfCommand _currentCommand;
    private Container container;
    private TreeMap<Integer, StageWidget> stageWidgetMap;
    private StageWidget stageWidget;
    private JPanel buttonsPanel;
    private int _currentStage;
    private IdfAnalyzer idfAnalyzer;
    private int numOfQueryableStages;

    /**
     * The CommandDialog constructor
     *
     * @param cc
     * the Command Controller
     * @param gem
     * the GemSetting
     * @param idf
     * the IDF
     * @param currentCommand
     * the current IdfCommand
     */
    public CommandDialog(CommandController cc, GemSetting gem, Idf idf,
        IdfCommand currentCommand) {
        _idf = idf;
        _cc = cc;
        _currentCommand = currentCommand;
        _gem = gem;
        currentCommand.getNumStages();
        stageWidgetMap = new TreeMap<Integer, StageWidget>();
        idfAnalyzer = new IdfAnalyzer(_idf);
        numOfQueryableStages = idfAnalyzer
            .getNumOfQueryableStages(currentCommand);
        initialize();
    }

    // Initializes the Command Dialog

```

```

private void initialize() {
    container = getContentPane();
    container.setLayout(new BorderLayout(container, BorderLayout.Y_AXIS));

    setModalityType(Dialog.ModalityType.APPLICATION_MODAL);
    setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
    setLocation(300, 30);
    setResizable(false);
    setTitle(_currentCommand.getLabel());
    setName(_currentCommand.getCommandName());

    addWindowListener(this);
}

/**
 * Initializes the specified stage's widget
 *
 * @param stageNumber
 * the stage number
 */
public void initializeStageWidget(int stageNumber) {
    try {
        _currentStage = stageNumber;

        // Perform if the current stage has never been initialized so far
        if (!stageWidgetMap.containsKey(stageNumber)) {
            stageWidget = new StageWidget(_cc, _gem, _idf, _currentCommand,
                stageNumber);
            stageWidget.setBorder(new EmptyBorder(10, 10, 10, 10));

            container.add(stageWidget);
            stageWidgetMap.put(stageNumber, stageWidget);
        }
    } catch (Exception e) {
        MessageDialog
            .showError("An error occurred while performing the requested
                operation. [Error Details: "
                    + e.getMessage() + "]);
    }
}

/**
 * Updates the Command Dialog to reflect changes
 */
public void revalidate() {

```

```
if (stageWidgetMap.size() > 0) {
    for (int stageNumber : stageWidgetMap.keySet()) {
        if (stageNumber == _currentStage) {

            // Make the current stage visible
            stageWidgetMap.get(stageNumber).setVisible(true);
        } else
            stageWidgetMap.get(stageNumber).setVisible(false);

        /*
         * Disable all inactive parameters in current stage and enable
         * the active ones
         */
        deactivateParams(stageNumber);
    }
}

validateWindow();
repaint();
pack();

if (getX() != 300 && getY() != 30)
    setLocation(300, 30);
}

private void validateWindow() {
    addButtons(); // Add the Cancel, Previous, Next, and OK buttons

    // Disables/enables buttons based on the number of queryable
    //stages
    if (idfAnalyzer.getNumOfQueryableStagesBefore(_currentCommand,
        _currentStage) > 0
        && idfAnalyzer.getNumOfQueryableStagesAfter(_currentCommand,
            _currentStage) > 0) {
        setPreviousButtonEnabled(true);
        setNextButtonEnabled(true);
    } else {
        if (idfAnalyzer.getNumOfQueryableStagesBefore(_currentCommand,
            _currentStage) == 0
            && idfAnalyzer.getNumOfQueryableStagesAfter(
                _currentCommand, _currentStage) == 0) {
            setPreviousButtonEnabled(false);
            setNextButtonEnabled(false);
        } else if (idfAnalyzer.getNumOfQueryableStagesBefore(
            _currentCommand, _currentStage) == 0) {
```

```

        setPreviousButtonEnabled(false);
        setNextButtonEnabled(true);
    } else if (idfAnalyzer.getNumOfQueryableStagesAfter(
        _currentCommand, _currentStage) == 0) {
        setPreviousButtonEnabled(true);
        setNextButtonEnabled(false);
    }
}
}

/**
 * Get the GUI of the specified stage
 *
 * @param stageNumber
 *   stage number
 * @return the stage GUI
 */
public StageWidget getStageWidget(int stageNumber) {
    if (stageWidgetMap.containsKey(stageNumber)) {
        return stageWidgetMap.get(stageNumber);
    } else
        return null;
}

/**
 * Get the number of Stages that have been initialized so far
 *
 * @return the number of initialized stages
 */
public int getStageCount() {
    return stageWidgetMap.size();
}

// Makes parameter widgets active or inactive
private boolean deactivateParams(int stageNumber) {
    boolean isDeactivated = false;

    if (stageWidgetMap.containsKey(stageNumber)) {
        JComponent stageComp = getStageWidget(stageNumber);

        for (Component c : stageComp.getComponents()) {
            if (c instanceof ParameterWidget) {
                if (isParamActive(c.getName(), stageNumber))
                    setEnabledComponents(c, true);
            } else {

```

```

        setEnabledComponents(c, false);
        isDeactivated = true;
    }
}
}
}

return isDeactivated;
}

// Enable or disable components
private void setEnabledComponents(Component comp, boolean enable) {
    JComponent paramWidget = (JComponent) comp;
    ParameterWidget pWidget = (ParameterWidget) paramWidget;
    IdfParameter paramObj = pWidget.getParamObject();
    int repNumber = -1;

    for (Component c : paramWidget.getComponents()) {
        try {
            if (!(c instanceof MoveUpButton || c instanceof MoveDownButton
                || c instanceof DeleteButton))
                c.setEnabled(enable);
            else {
                if (!enable)
                    c.setEnabled(enable);
                else {
                    repNumber = -1;
                    if (c instanceof MoveUpButton) {
                        repNumber = ((MoveUpButton) c).getRepNumber();
                        if (repNumber == 0)
                            c.setEnabled(false);
                        else
                            c.setEnabled(true);
                    } else if (c instanceof MoveDownButton) {
                        repNumber = ((MoveDownButton) c).getRepNumber();
                        if (repNumber == pWidget.getFieldsCount() - 1)
                            c.setEnabled(false);
                        else
                            c.setEnabled(true);
                    } else if (c instanceof DeleteButton) {
                        if (paramObj.getMinNumberOfReps() == pWidget
                            .getFieldsCount())
                            c.setEnabled(false);
                        else
                            c.setEnabled(true);
                    }
                }
            }
        }
    }
}

```

```

        }
    }
} catch (Exception e) {
}
}
}

// Adds buttons to the Command Dialog
private void addButtonPanel() {
    deleteButtonPanel();

    buttonsPanel = new JPanel();
    buttonsPanel
        .setLayout(new BorderLayout(buttonsPanel, BorderLayout.LINE_AXIS));
    buttonsPanel.setBorder(new EmptyBorder(10, 10, 10, 10));

    buttonsPanel.setName("buttonsPanel");

    CommandCancelButton cancelButton = new CommandCancelButton(_cc);
    buttonsPanel.add(cancelButton);

    buttonsPanel.add(Box.createHorizontalGlue());

    if (numOfQueryableStages > 1) {
        PreviousButton prevButton = new PreviousButton(_cc);
        prevButton.setText("Previous");
        prevButton.setName("PreviousButton");

        if (idfAnalyzer.getNumOfQueryableStagesBefore(_currentCommand,
            _currentStage) == 0)
            prevButton.setEnabled(false);

        buttonsPanel.add(prevButton);

        NextButton nextButton = new NextButton(_cc);
        nextButton.setText("Next");
        nextButton.setName("NextButton");

        if (idfAnalyzer.getNumOfQueryableStagesAfter(_currentCommand,
            _currentStage) == 0)
            nextButton.setEnabled(false);

        buttonsPanel.add(nextButton);
    }
}

```

```
    }

    CommandOKButton okButton = new CommandOKButton(_cc);
    buttonsPanel.add(okButton);

    container.add(buttonsPanel);
}

// Get the index of a button in the buttons panel
private int getButtonIndex(String name) {
    return WidgetAnalyzer.getWidgetIndexInDialog(this, name);
}

// Delete the buttons panel
private void deleteButtonPanel() {
    int index = WidgetAnalyzer.getWidgetIndexInDialog(this,
        "buttonsPanel");

    if (index > -1)
        container.remove(index);
}

// Enables/Disables the previous button
private void setPreviousButtonEnabled(boolean enable) {
    int index = getButtonIndex("PreviousButton");

    if (index > -1)
        ((PreviousButton) container.getComponent(index))
            .setEnabled(enable);
}

// Enables/Disables the next button
private void setNextButtonEnabled(boolean enable) {
    int index = getButtonIndex("NextButton");

    if (index > -1)
        ((NextButton) container.getComponent(index))
            .setEnabled(enable);
}

/**
 * Checks if the specified parameter is active.
 *
 * @param paramName
 * name of parameter
 */
```

```

* @param stageNumber
* stage number
* @return true or false
*/
public boolean isParamActive(String paramName, int stageNumber) {
    idfAnalyzer.setCurrentCommand(_currentCommand.getCommandName());
    IdfParameter param = idfAnalyzer
        .getIdfParameter(paramName, stageNumber);

    // Check if the parameter has a ParentParameter
    boolean isActive = false;
    if (param.getParentParameter() == null
        || param.getParentParameter().equals("")) {
        isActive = true;
    } else {
        /*
        * Since the parameter has a ParentParameter, verify if the
        * parameter's ParentValue is same as the parent parameter's value
        */
        IdfParameter parentParam = idfAnalyzer.getIdfParameter(param
            .getParentParameter());
        String expectedParentValue = param.getParentValue(); // Expected
            // value of
            // the
            // Parent
            // Parameter
        String actualParentValue = ""; // Actual value of the Parent
            // Parameter

        try {
            for (int t = 0; t < parentParam.getMaxNumberOfReps(); t++) {
                actualParentValue = _gem.getParameter(
                    parentParam.getParameterName(), t).toString();

                if (actualParentValue.equals(expectedParentValue)) {
                    isActive = true;
                    break;
                }
            }
        } catch (Exception e) {
            isActive = false;
        }

        // If parent parameter has no value in Gem, then get its default
        // value (if any)

```



```
    if (actualParentValue == null || actualParentValue.equals("")) {
        if (parentParam.getDefaultValueMethod() != null
            && !parentParam.getDefaultValueMethod().equals(""))
            actualParentValue = _gem.callDefaultValueMethod(
                parentParam.getParameterName()).toString();
        else
            actualParentValue = parentParam.getDefaultValue();

        if (actualParentValue.equals(expectedParentValue))
            isActive = true;
    }
}

return isActive;
}

public void windowClosed(WindowEvent e) {
    dispose();
}

public void windowActivated(WindowEvent e) {
}

public void windowClosing(WindowEvent e) {
}

public void windowDeactivated(WindowEvent e) {
}

public void windowDeiconified(WindowEvent e) {
}

public void windowIconified(WindowEvent e) {
}

public void windowOpened(WindowEvent e) {
}
}
```

G.3 The *QuestionDialog* class

This class creates the Question Dialog Box.

```
package johar.interfaceinterpreter.star;

import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.util.ArrayList;
import java.util.List;
import java.awt.Dialog;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JPanel;

import johar.idf.Idf;
import johar.idf.IdfQuestion;

/**
 * The Question Dialog Box for each question attribute in the IDF.
 *
 */
public class QuestionDialog extends JDialog implements WindowListener {
    private IdfQuestion _questionObj;
    private CommandController _cc;
    private JPanel mainPanel;
    private Container container;
    private GridBagConstraints constraints;
    private ParameterWidget paramWidget;
    private Object _defaultValue;
    private List<ParameterWidget> paramWidgetList;
    private IdfAnalyzer idfAnalyzer;

    /**
     *
     * @param defaultValue
     * default value of the Question attribute
     * @param cc
     * the Command Controller object
     * @param questionObj
     * IdfQuestion object
     */
    public QuestionDialog(Object defaultValue, CommandController cc,
        Idf idf, IdfQuestion questionObj) {
```

```

    _questionObj = questionObj;
    _cc = cc;
    _defaultValue = defaultValue;
    paramWidgetList = new ArrayList<ParameterWidget>();
    idfAnalyzer = new IdfAnalyzer(idf); /*
        * Provides access to various
        * information about the specified
        * IDF e.g. list of
        * queryable/non-queryable params in
        * a command/stage, list of
        * questions in a command, list of
        * tables, list of browsable tables,
        * etc.
        */
}

// Initialize the GUI components
private void initialize() {
    mainPanel = new JPanel();
    container = getContentPane();
    container.setLayout(new FlowLayout(FlowLayout.CENTER, 20, 10));
    mainPanel.setLayout(new GridBagLayout());
    constraints = new GridBagConstraints();

    setModalityType(Dialog.ModalityType.APPLICATION_MODAL);
    setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
    setLocation(300, 90);
    setResizable(false);

    container.add(mainPanel);
    addWindowListener(this);
}

/**
 * Creates and displays the dialog box.
 */
public void showDialog() {
    try {
        initialize();
        _questionObj.getLabel();

        createWidgetQuestion();
        addOKCancelButtons();
        setTitle("Question");
        pack();
    }
}

```

```

        setVisible(true);
    } catch (Exception e) {
        MessageDialog
            .showError("An error occurred while performing the requested
                operation. [Error Details: "
                    + e.getMessage() + "]");
    }
}

// Creates the widget for the question
private void createWidgetQuestion() {
    constraints.gridx = 0;
    constraints.fill = GridBagConstraints.HORIZONTAL;
    constraints.insets = new Insets(1, 0, 0, 0);
    constraints.gridy = 0;

    // Creates a Label for the Question Dialog
    JLabel questionLabel = new JLabel();

    if (_questionObj.getLabel().length() > 90)
        questionLabel.setText("<html><body><p style='width: 400px'>"
            + _questionObj.getLabel());
    else
        questionLabel.setText(_questionObj.getLabel());

    mainPanel.add(questionLabel, constraints);

    // Creates and stores the widgets
    String type = _questionObj.getType();
    if (type.equals("text"))
        addTextField();
    else if (type.equals("int") || type.equals("float"))
        addNumberField();
    else if (type.equals("boolean"))
        addBooleanField();
    else if (type.equals("choice"))
        addChoiceField();
    else if (type.equals("timeOfDay"))
        addTimeField();
    else if (type.equals("date"))
        addDateField();
    else if (type.equals("file"))
        addFileField();
    else if (type.equals("tableEntry"))
        && !idfAnalyzer.getTable(_questionObj.getSourceTable())

```

```
        .getBrowsable())
        addTableEntryField();

    }

    // The position of the question widget
    private void initPosition() {
        constraints.fill = GridBagConstraints.NONE;
        constraints.anchor = GridBagConstraints.CENTER;
        constraints.gridy = 1;
        constraints.insets = new Insets(5, 0, 0, 0);
    }

    // Creates widget for question of type "text"
    private void addTextField() {
        constraints.gridy = 1;
        constraints.insets = new Insets(5, 0, 0, 0);

        paramWidget = new TextWidget(_defaultValue, _questionObj, _cc);
        paramWidgetList.add(paramWidget); // Add to Map
        mainPanel.add(paramWidget, constraints);
    }

    // Creates widget for question of type "int" or "float"
    private void addNumberField() {
        initPosition();

        paramWidget = new NumberWidget(_defaultValue, _questionObj, _cc);
        paramWidgetList.add(paramWidget); // Add to Map
        mainPanel.add(paramWidget, constraints);
    }

    // Creates widget for question of type "boolean"
    private void addBooleanField() {
        initPosition();

        paramWidget = new BooleanWidget(_defaultValue, _questionObj, _cc);
        paramWidgetList.add(paramWidget); // Add to Map
        mainPanel.add(paramWidget, constraints);
    }

    // Creates widget for question of type "choice"
    private void addChoiceField() {
        initPosition();
```

```
    paramWidget = new ChoiceWidget(_defaultValue, _questionObj, _cc);
    paramWidgetList.add(paramWidget); // Add to Map
    mainPanel.add(paramWidget, constraints);
}

// Creates widget for question of type "timeOfDay"
private void addTimeField() {
    initPosition();

    paramWidget = new TimeWidget(_defaultValue, _questionObj, _cc);
    paramWidgetList.add(paramWidget); // Add to Map
    mainPanel.add(paramWidget, constraints);
}

// Creates widget for question of type "date"
private void addDateField() {
    initPosition();

    paramWidget = new DateWidget(_defaultValue, _questionObj, _cc);
    paramWidgetList.add(paramWidget); // Add to Map
    mainPanel.add(paramWidget, constraints);
}

// Creates widget for question of type "file"
private void addFileField() {
    initPosition();

    paramWidget = new FileWidget(_defaultValue, _questionObj, _cc);
    paramWidgetList.add(paramWidget); // Add to Map
    mainPanel.add(paramWidget, constraints);
}

// Creates widget for question of type "tableEntry",
//whose SourceTable is
// non-browsable
private void addTableEntryField() {
    initPosition();

    paramWidget = new TableEntryWidget(_defaultValue, _questionObj, _cc);
    paramWidgetList.add(paramWidget); // Add to Map
    mainPanel.add(paramWidget, constraints);
}

// Creates the buttons
private void addOKCancelButtons() {
```

```
constraints.fill = GridBagConstraints.NONE;
constraints.insets = new Insets(15, 0, 0, 0);
constraints.gridy = 2;
constraints.gridwidth = GridBagConstraints.REMAINDER;
constraints.anchor = GridBagConstraints.WEST;

QuestionCancelButton cancelButton = new QuestionCancelButton(_cc,
    _questionObj.getParameterName());
mainPanel.add(cancelButton, constraints);

constraints.anchor = GridBagConstraints.EAST;
constraints.gridy = 2;

QuestionOKButton okButton = new QuestionOKButton(_cc,
    _questionObj.getParameterName());
mainPanel.add(okButton, constraints);
}

/**
 * Gets the Question Widget
 *
 * @return the map containing parameter widget
 */
public List<ParameterWidget> getQuestionWidget() {
    return paramWidgetList;
}

public void windowClosed(WindowEvent e) {
    dispose();
}

public void windowActivated(WindowEvent e) {
}

public void windowClosing(WindowEvent e) {
}

public void windowDeactivated(WindowEvent e) {
}

public void windowDeiconified(WindowEvent e) {
}

public void windowIconified(WindowEvent e) {
}
```

```

    public void windowOpened(WindowEvent e) {
    }
}

```

G.4 The *HelpBox* class

This class creates the Help Box.

```

package johar.interfaceinterpreter.star;

import javax.swing.JSplitPane;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.Font;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.SwingConstants;
import javax.swing.border.EmptyBorder;

import johar.idf.Idf;
import johar.idf.IdfCommand;
import johar.idf.IdfParameter;
import johar.idf.IdfQuestion;

/**
 * This class creates a Help Box with three states:
 * the Top-Level state, the Command state, and the
 * Parameter/Question state.
 */
public class HelpBox extends JFrame {
    private Container container;
    private Idf _idf;
    private CommandController _cc;
    private IdfAnalyzer idfAnalyzer;
}

```



```

private HelpContents contents;
private JTextArea multiLineBox;
private ScrollingWidget multiLineScroll;
private JPanel titlePanel;
private JSplitPane splitter;

/**
 * The class' constructor
 *
 * @param idf
 * the IDF object
 * @param cc
 * the Command Controller object
 */
public HelpBox(Idf idf, CommandController cc) {
    _idf = idf;
    _cc = cc;
    idfAnalyzer = new IdfAnalyzer(idf); /*
        * Provides access to various
        * information about the specified
        * IDF e.g. list of
        * queryable/non-queryable params in
        * a command/stage, list of
        * questions in a command, list of
        * tables, list of browsable tables,
        * etc.
        */
    contents = new HelpContents(_idf, _cc); // Get the tables

    try {
        initDialog();
    } catch (Exception e) {
        MessageDialog
            .showError("An error occurred while performing the " +
                "requested operation. [Error Details: "
                + e.getMessage() + "]);
    }
}

// Creates and shows the Help Box
private void initDialog() {
    container = getContentPane();
    container.setLayout(new BorderLayout());

    setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);

```

```

        setLocation(300, 30);
        setTitle("Help");
        setName("help");
        setSize(new Dimension(700, 600));
        showTopLevelState();
        setVisible(true);
    }

    // Sets the title of all the three states
    private void setTitle(int index, String title) {
        int compIndex = getCompIndex("Title" + index);
        if (compIndex >= 0) {
            JPanel titlePan = (JPanel) container.getComponent(compIndex);
            ((JLabel) titlePan.getComponent(0)).setText(title);
        } else {
            titlePanel = new JPanel();
            titlePanel.setLayout(new BorderLayout());
            JLabel titleLabel = new JLabel(title);
            titlePanel.setName("Title" + index);
            titlePanel.setBackground(new Color(235, 235, 235));
            titlePanel.add(titleLabel, BorderLayout.NORTH);
            titleLabel.setHorizontalAlignment(SwingConstants.CENTER);
            titleLabel.setFont(new Font(null, Font.BOLD, 13));
            container.add(titlePanel, BorderLayout.NORTH);
        }
    }

    // Creates the scrollable text area, and sets its content to the
    // MultiLineHelp
    private void setMultiLineText(String name, String text,
        String textHeader) {
        int compIndex = getCompIndex(name);
        if (compIndex >= 0) {
            JPanel titlePan = (JPanel) container.getComponent(compIndex);
            if (WidgetAnalyzer.getWidgetIndex(titlePan, "multiLineBox") == -1)
            {
                multiLineBox = new JTextArea();
                multiLineBox.setText(text);
                multiLineBox.setEditable(false);
                multiLineBox.setWrapStyleWord(true);
                multiLineBox.setLineWrap(true);
                multiLineScroll = new ScrollingWidget(multiLineBox);
                multiLineScroll.setName("multiLineBox");

                if (name.equals("Title1")) {

```

```

        titlePan.setPreferredSize(new Dimension(700, 200));
        titlePan.add(multiLineScroll);
    } else {
        JLabel titleLabel = new JLabel(textHeader);
        multiLineScroll.setPreferredSize(new Dimension(700,
            getPreferredSize().height - 460));
        titlePan.add(titleLabel, BorderLayout.CENTER);
        titleLabel.setHorizontalAlignment(SwingConstants.CENTER);
        titleLabel.setFont(new Font(null, Font.ITALIC, 13));
        titlePan.add(multiLineScroll, BorderLayout.SOUTH);
    }
} else
    ((ScrollingWidget) titlePan.getComponent(1))
        .getTextAreaInstance().setText(text);
}
}

/**
 * Shows the Top Level State
 */
public void showTopLevelState() {
    setTitle(0, "Commands");

    if (getCompIndex("topLevelHelp") == -1)
        container.add(contents.getTopLevelStateTable());

    if (!(getCompIndex("buttonsPanel") >= 0))
        addButtons();

    setCompVisible("topLevelHelp", true);
    deleteComponent("paramsTables");
    setTitleVisible("Title0", true);
    deleteComponent("Title1");
    deleteComponent("Title2");
    setButtonVisible("back", false);

    validate();
    repaint();
}

/**
 * Shows the Command State for the selected command
 *
 * @param command
 * command selected

```

```

*/
public void showCommandState(IdfCommand command) {
    String commandGrp = "";
    try {
        commandGrp = idfAnalyzer.getCommandGroup(command.getCommandName())
            .getLabel();
    } catch (Exception e) {
    }

    setTitle(1, commandGrp + " \u2192 " + command.getLabel());
    setMultiLineText("Title1", command.getMultiLineHelp(), "");

    JPanel questionPanel = new JPanel();
    questionPanel.setLayout(new BorderLayout());
    JLabel titleLabel = new JLabel("Questions that might be asked:");
    titleLabel.setFont(new Font(null, Font.PLAIN, 14));
    questionPanel.setBackground(new Color(240, 240, 240));
    questionPanel.setName("questionsHelp");
    questionPanel.add(titleLabel, BorderLayout.NORTH);
    questionPanel.add(contents.getCommandStateTableQues(command));

    if (getCompIndex("paramsTables") == -1) {
        splitter = new JSplitPane();
        splitter.setDividerSize(5);
        splitter.setOneTouchExpandable(true);
        splitter.setOrientation(JSplitPane.VERTICAL_SPLIT);

        splitter.setDividerLocation(250);
        splitter.setName("paramsTables");
        splitter.setLeftComponent(contents
            .getCommandStateTableParams(command));
        splitter.setRightComponent(questionPanel);
        container.add(splitter);
    }

    deleteComponent("topLevelHelp");
    setCompVisible("paramsTables", true);
    deleteComponent("Title0");
    setTitleVisible("Title1", true);
    deleteComponent("Title2");
    setButtonVisible("back", true);

    validate();
    repaint();
}

```

```

/**
 * Shows the Parameter State for the selected parameter
 *
 * @param command
 * current command
 * @param param
 * selected parameter
 * @param isQuestion
 * true, if param is a Question; false, if param is a Parameter
 */
public void showParameterState(IdfCommand command,
    IdfParameter param, boolean isQuestion) {
    String commandGrp = "";

    commandGrp = idfAnalyzer.getCommandGroup(command.getCommandName())
        .getLabel();

    setTitle(2, commandGrp + " \u2192 " + command.getLabel());

    if (!isQuestion)
        setMultiLineText("Title2", param.getMultiLineHelp(),
            "Parameter: " + param.getLabel());
    else
        setMultiLineText("Title2", param.getMultiLineHelp(),
            "Question: " + param.getLabel());

    deleteComponent("topLevelHelp");
    deleteComponent("paramsTables");
    deleteComponent("Title0");
    deleteComponent("Title1");
    setTitleVisible("Title2", true);
    setButtonVisible("back", true);

    validate();
    repaint();
}

/**
 * Gets the table of commands displayed in the Top Level State
 *
 * @return the table object
 */
public TableWidget getTopLevelTable() {
    int compIndex = getCompIndex("topLevelHelp");

```

```

    if (compIndex >= 0) {
        return ((ScrollingWidget) container.getComponent(compIndex))
            .getTableWidgetInstance();
    }
    return null;
}

/**
 * Gets the table of parameters displayed in the Command State
 *
 * @return the table object
 */
public TableWidget getParamsTable() {
    int compIndex = getCompIndex(splitter, "paramsHelp");
    if (compIndex >= 0) {
        return ((ScrollingWidget) splitter.getComponent(compIndex))
            .getTableWidgetInstance();
    }
    return null;
}

/**
 * Gets the table of questions displayed in the Command State
 *
 * @return the table object
 */
public TableWidget getQuestionsTable() {
    int compIndex = getCompIndex(splitter, "questionsHelp");
    if (compIndex >= 0) {
        JPanel quesPanel = (JPanel) splitter.getComponent(compIndex);
        return ((ScrollingWidget) quesPanel.getComponent(1))
            .getTableWidgetInstance();
    }
    return null;
}

/**
 * Gets the selected row index from the commands table in the
 * Top Level State
 *
 * @return the index of the selected row
 */
public int getTopLevelTableSelectedRow() {
    TableWidget table = getTopLevelTable();
    if (table != null)

```

```

        return table.getSelectedRow();
    else
        return -1;
    }

/**
 * Gets the selected row index from the parameters table
 * in the Command State
 *
 * @return the index of the selected row
 */
public int getParamsTableSelectedRow() {
    TableWidget table = getParamsTable();
    if (table != null)
        return table.getSelectedRow();
    else
        return -1;
}

/**
 * Gets the selected row index from the questions table
 * in the Command State
 *
 * @return the index of the selected row
 */
public int getQuestionsTableSelectedRow() {
    TableWidget table = getQuestionsTable();
    if (table != null)
        return table.getSelectedRow();
    else
        return -1;
}

// Deletes the specified component from the main container
private void deleteComponent(String name) {
    int compIndex = getCompIndex(name);
    if (compIndex >= 0) {
        container.remove(compIndex);
    }
}

// Makes the specified component to be visible or not visible
private void setCompVisible(String name, boolean isVisible) {
    int compIndex = getCompIndex(name);
    if (compIndex >= 0) {

```

```

        container.getComponent(compIndex).setVisible(isVisible);
    }
}

// Makes the specified button to be visible or not visible
private void setButtonVisible(String name, boolean isVisible) {
    int compIndex = getCompIndex("buttonsPanel");
    if (compIndex >= 0) {
        ((JPanel) container.getComponent(compIndex)).getComponent(0)
            .setVisible(isVisible);
    }
}

// Makes the title visible or not visible
private void setTitleVisible(String name, boolean isVisible) {
    int compIndex = getCompIndex(name);
    if (compIndex >= 0) {
        ((JPanel) container.getComponent(compIndex))
            .setVisible(isVisible);
    }
}

// Get the index of the specified component in the main container
private int getCompIndex(String name) {
    return WidgetAnalyzer.getWidgetIndex(
        (JComponent) container, name);
}

// Get the index of the specified component in a particular
//container
private int getCompIndex(JComponent comp, String name) {
    return WidgetAnalyzer.getWidgetIndex(comp, name);
}

// Adds buttons to the main container
private void addButton() {
    JPanel buttonsPanel = new JPanel();

    buttonsPanel.setBorder(new EmptyBorder(10, 10, 5, 10));
    buttonsPanel
        .setLayout(new BoxLayout(buttonsPanel, BoxLayout.LINE_AXIS));
    buttonsPanel.setName("buttonsPanel");

    HelpBackButton backButton = new HelpBackButton(_cc);
    backButton.setVisible(false);
}

```



```
        buttonsPanel.add(backButton);

        HelpOKButton okButton = new HelpOKButton(_cc);

        buttonsPanel.add(Box.createHorizontalGlue());
        buttonsPanel.add(okButton);
        container.add(buttonsPanel, BorderLayout.SOUTH);
    }
}
```

Appendix H

App Engine of the Temperature Converter App

```
/*
 * Application Engine of the Temperature Converter App
 *
 * TemperatureConverter.java
 */

import java.text.DecimalFormat;
import johar.gem.Gem;

public class TemperatureConverter {

    //Initialization Method displays a Welcome message to the user
    public void initTemperatureConverter(Gem gem){
        String message = "Welcome to the Temperature Converter App."
            + "\nConversion from Celsius to Fahrenheit, and "
            + "vice-versa, just got easier!";
        gem.showText(message, 2000);
        gem.showText("Ready", 1000);
    }

    //Method to convert Celsius to Fahrenheit
    public void celsiusToFahrenheit(Gem gem){
        double celsiusTemp =
            gem.getFloatParameter("celsius");
        double fahrenheitTemp = convertToFahrenheit(celsiusTemp);
        fahrenheitTemp = Double.parseDouble(
            new DecimalFormat("0.0").format(fahrenheitTemp)
        );
        String outputMessage = "Temperature in Celsius: "
```

```

    + celsiusTemp + "\n"
    + "Temperature in Fahrenheit: " + fahrenheitTemp;
    gem.showText(outputMessage, 2000);
}

//Method to convert Fahrenheit to Celsius
public void fahrenheitToCelsius(Gem gem){
    double fahrenheitTemp =
        gem.getFloatParameter("fahrenheit");
    double celsiusTemp = convertToCelsius(fahrenheitTemp);
    celsiusTemp = Double.parseDouble(
        new DecimalFormat("0.0").format(celsiusTemp)
    );
    String outputMessage = "Temperature in Fahrenheit: "
    + fahrenheitTemp + "\n"
    + "Temperature in Celsius: " + celsiusTemp;
    gem.showText(outputMessage, 2000);
}

//Actual Fahrenheit to Celsius conversion is done here
private double convertToFahrenheit(double celsiusTemp){
    double fahrenheitTemp = (celsiusTemp * (9.0/5.0)) + 32.0;
    return fahrenheitTemp;
}

//Actual Fahrenheit to Celsius conversion is done here
private double convertToCelsius(double fahrenheitTemp){
    double celsiusTemp = (fahrenheitTemp - 32.0) * (5.0/9.0);
    return celsiusTemp;
}

//Method is called when user expresses intent to exit the app
public void exitApp(Gem gem){
}

//Method to determine whether to ask the user to confirm
//his/her intent to exit the app
public boolean confirmAppExit(Gem gem){
    return true;
}

//Method to confirm whether to exit the app or not
public boolean appShouldQuit(Gem gem){
    boolean questionResponse =
        gem.getBooleanParameter("confirmExit");
}

```

```
    if (questionResponse)
        return true;
    else
        return false;
}
}
```

Appendix I

Interface Description File (IDF) of the Appointment Calendar App

```
Application = IdesOfJohar
ApplicationEngine = IdesOfJohar
IdfVersion = "1.0"
InitializationMethod = initIdesOfJohar
```

```
Table weeks = {
    Browsable = yes
    DefaultHeading = "Weeks"
    DefaultColumnNames = "Sun|Mon|Tue|Wed|Thu|Fri|Sat"
}
```

```
Table days = {
    Browsable = yes
    DefaultHeading = "Days"
    DefaultColumnNames = "Date|Number of Appointments"
}
```

```
Table appointments = {
    Browsable = yes
    DefaultHeading = "Appointments"
    DefaultColumnNames = "Time|Description"
}
```

```
CommandGroup appointment = {
    Member = addAppointment
    Member = cancel
    Member = exit
}
```

```
CommandGroup previous = {
    Member = previousMonth
    Member = previousWeek
    Member = previousDay
}

CommandGroup next = {
    Member = nextMonth
    Member = nextWeek
    Member = nextDay
}

CommandGroup goTo = {
    Member = goToSelectedWeek
    Member = goToSelectedDay
    Member = goToDate
}

Command addAppointment = {
    Label = "Add Appointment"

    BriefHelp = "Add an appointment"
    MultiLineHelp = "Add an appointment on the current day"
    OneLineHelp = "Add an appointment on the current day"

    Parameter time = {
        Type = timeOfDay
        DefaultValue = "9:00 am"
        BriefHelp = "Time of the appointment"
        MultiLineHelp = {{
            The time of the appointment. The format is hh:mm am.
        }}
        OneLineHelp = "Time of the appointment"
    }

    Parameter description = {
        Type = text
        MaxNumberOfChars = 240
        MaxNumberOfLines = 3
        BriefHelp = "Text description"
        MultiLineHelp = "A description of the appointment in text format."
        OneLineHelp = "Text description of the appointment"
    }

    Parameter numReps = {
```

```

    Type = int
    DefaultValue = 1
    Label = "Number of repetitions"
    MinValue = 1
    BriefHelp = "Number of repetitions"
    MultiLineHelp = {{
        The number of repetitions for the appointment, to be
        used with the 'Repeat Every' parameter. For example,
        2 daily repetitions will add the appointment twice, for
        two consecutive days.
    }}
    OneLineHelp = "Number of repetitions of the appointment"
}

Parameter repeatEvery = {
    Type = choice
    Choices = "day|week|month"
    DefaultValue = day
    BriefHelp = "Repetition period"
    MultiLineHelp = {{
        The period of repetition of the appointment, which can be
        'day', 'week' or 'month'. For example, 3 repetitions with
        period 'day' will add the appointment for 3 consecutive
        days; 3 repetitions with period 'week' will add the
        appointment for 3 consecutive weeks on the same day of
        the week.
    }}
    OneLineHelp = "Period of repetition of the appointment"
}

Command cancel = {
    Label = "Cancel Appointment"
    BriefHelp = "Cancel appointment"
    OneLineHelp = "Cancel the selected appointment"
    MultiLineHelp = {{
        Cancel the appointment that is selected from the
        appointment table.
    }}
}

Parameter appt = {
    Type = tableEntry
    SourceTable = appointments
    BriefHelp = "Selected appointment"
    MultiLineHelp = {{

```

```
        The selected row of the appointment table for cancellation.
    }}
    OneLineHelp = "The appointment to be cancelled"
}
}

Command previousMonth = {
    BriefHelp = "Go to previous month"
    MultiLineHelp = "Go to previous month"
    OneLineHelp = "Go to previous month"
}

Command nextMonth = {
    BriefHelp = "Go to next month"
    MultiLineHelp = "Go to next month"
    OneLineHelp = "Go to next month"
}

Command previousWeek = {
    BriefHelp = "Go to previous week"
    MultiLineHelp = "Go to previous week"
    OneLineHelp = "Go to previous week"
}

Command nextWeek = {
    BriefHelp = "Go to next week"
    MultiLineHelp = "Go to next week"
    OneLineHelp = "Go to next week"
}

Command previousDay = {
    BriefHelp = "Go to previous day"
    MultiLineHelp = "Go to previous day"
    OneLineHelp = "Go to previous day"
}

Command nextDay = {
    BriefHelp = "Go to next day"
    MultiLineHelp = "Go to next day"
    OneLineHelp = "Go to next day"
}

Command goToSelectedWeek = {
    BriefHelp = "Go to selected week"
    MultiLineHelp = "Go to the week selected from the Weeks table."
}
```



```

OneLineHelp = "Go to the week selected from the Weeks table"

Parameter week = {
    Type = tableEntry
    SourceTable = weeks
    BriefHelp = "Selected week"
    MultiLineHelp = "The week to go to, selected from the Weeks table."
    OneLineHelp = "The week to go to"
}
}

Command goToSelectedDay = {
    BriefHelp = "Go to selected day"
    MultiLineHelp = "Go to the day selected from the Days table."
    OneLineHelp = "Go to the day selected from the Days table"

    Parameter day = {
        Type = tableEntry
        SourceTable = days
        BriefHelp = "Selected day"
        MultiLineHelp = "The day to go to, selected from the Days table."
        OneLineHelp = "The day to go to"
    }
}

Command goToDate = {
    BriefHelp = "Go to specific date"
    MultiLineHelp = "Jump to an arbitrary date."
    OneLineHelp = "Go to a specific date"
    Parameter dateToGoTo = {
        Type = date
        BriefHelp = "Date to go to"
        MultiLineHelp = "Date to go to."
        OneLineHelp = "Date to go to"
    }
}
}

Command exit = {
    QuitAfter = yes
}

```

Appendix J

“StarX” Interface Interpreter: Requirements Specification of the StarX GUI

StarX is an eXtended version of the Star interface interpreter with extra functionality for keyboard-only interactions. Thus, if a user prefers to use only the keyboard (without the mouse) to interact with interface components, StarX interface interpreter will fully support the user by activating keyboard shortcuts (see below). StarX also supports users that prefer using both keyboard and mouse to interact with interface components, but the keyboard shortcuts (defined in this document) will not be active.

This document extends the requirements for Star (see Star GUI Specification document in Appendix E). Note that all the keyboard shortcuts defined in this document are not case-sensitive.

J.1 Main Panel

All the requirements (except 1 and 14) under under the section labeled “The Main Panel” in the Star GUI Specification document are applicable under this section, in addition to the following:

- (1) At the top is the Menu Bar, which shows one menu for each command group in the application, and one menu named “StarX” (for services provided by the StarX interpreter for all Johar applications). The non-StarX menus are referred to as application menus.
- (2) On the “StarX” menu, there are three menu items: *Help*, *Enable Keyboard-Only Interaction*, and *Show Hotkeys Pop-Up Table*.

- (3) The supported keyboard shortcuts for the main panel and their respective functions are specified in the following sub-sections.
- (4) Whenever either the Menu Bar, Text Display Area, or Table Area receives focus, a red border is displayed around it to serve as a visual cue for the user.

J.1.1 The Menu Bar

Table J.1 shows the keyboard shortcuts required for interacting with the Menu Bar.

Keyboard Shortcut	Function
Alt + M	Selects the first menu on the menu bar. (Note that <i>Alt + M</i> means the user holds down <i>Alt</i> key while pressing <i>M</i>).
left arrow or Z	Selects the preceding menu.
right arrow or X	Selects the next or succeeding menu.
down arrow	Moves down to the next command of a selected menu.
up arrow	Moves up to the previous command of a selected menu
S or Return	Selects a command.

Table J.1: Keyboard shortcuts for the Menu Bar

J.1.2 The Text Display Area

Table J.2 shows the keyboard shortcuts required for interacting with the Text Display Area.

Keyboard Shortcut	Function
T	Causes the Text Display Area to receive focus.
down arrow	Scrolls down the Text Display Area.
up arrow	Scrolls up the Text Display Area.

Table J.2: Keyboard shortcuts for the Text Display Area

J.1.3 The Table Area

Table J.3 shows the keyboard shortcuts required for interacting with the Table Area.

Keyboard Shortcut	Function
H	Causes the Table Area to receive focus, and then selects the top table.

F	Selects the first row of the current table.
C	Selects the current row of the current table.
L	Selects the last row of the current table.
D	Deselects the current row of the current table.
down arrow	Moves down to the next row without selecting it.
up arrow	Moves up to the previous row without selecting it.
left arrow or Z	Selects the preceding table.
right arrow or X	Selects the next or succeeding table.

Table J.3: Keyboard shortcuts for the Table Area

J.2 The Command Dialog Box

Requirements 1 to 15 under the section labeled “The Command Dialog Box” in the Star GUI Specification document are applicable under this section, in addition to the keyboard shortcuts in Table J.4, J.5, and J.6. Note that *Control* + <key> means the user holds down *Control* while pressing the key.

Keyboard Shortcut	Function
Control + Q	Presses the Cancel button.
Control + P	Presses the Previous button.
Control + N	Presses the Next button.
Control + K	Presses the OK button.

Table J.4: Keyboard shortcut for the Cancel, Previous, Next and OK buttons

J.2.1 Parameter Section of the Command Dialog Box

Requirements 1 to 5 under the section labeled “Parameter Section of the Command Dialog Box” in the Star GUI Specification document are applicable under this section, in addition to the keyboard shortcuts in Table J.5.

Keyboard Shortcut	Function
-------------------	----------

F2	<ul style="list-style-type: none"> • Pressing this key for the first time causes the widget for the topmost repetition of the current stage’s 1st parameter section to receive focus. • Continuous pressing of the key causes the widget for the topmost repetition of the next parameter section to receive focus. • If the current parameter section is the last parameter section of the stage, pressing the key causes the widget for the topmost repetition of the 1st parameter section to receive focus again.
F3	Transfers focus to the widget for the next repetition of the current parameter section.
F4	Transfers focus to the widget for the previous repetition of the current parameter section.

Table J.5: Keyboard shortcuts for the Parameter Section

Whenever the widget for the topmost repetition of a stage’s parameter section receives focus, a red border is shown around the current repetition section to serve as a visual cue for the user.

J.2.2 Repetition Section of the Parameter Section

Requirements 1 to 5 under the section labeled “Repetition Section of the Parameter Section” in the Star GUI Specification document are applicable under this section, in addition to the keyboard shortcuts in Table J.6. Note that *Control* + <key> means the user holds down *Control* while pressing the key.

Keyboard Shortcut	Function
Control + A	Presses the “Add another” button of the current parameter section.
Control + X	Presses the Delete button of the current repetition.
Control + U	Presses the Move Up button of the current repetition.

Control + D	Presses the Move Down button of the current repetition.
-------------	---

Table J.6: Keyboard shortcuts for buttons in the Repetition Section of a Parameter Section

Refer to Section J.5 below for details on the keyboard shortcuts required for interacting with certain widgets which the user can use to select the value of a repetition (depending on the parameter type).

J.3 Question Dialog Box

Requirements 1 to 4 under the section labeled “Question Dialog Box” in the Star GUI Specification document are applicable under this section, in addition to the keyboard shortcuts in Table J.7. Note that *Control* + *<key>* means the user holds down *Control* while pressing the key.

Keyboard Shortcut	Function
F2	Causes the parameter widget to receive focus.
Control + Q	Presses the Cancel button.
Control + K	Presses the OK button.

Table J.7: Keyboard shortcuts for the Question Dialog Box

When the parameter widget receives focus, a red border is displayed around it to serve as visual cue for the user.

Refer to Section J.5 below for details on the keyboard shortcuts required for interacting with certain widgets which the user can use to select the value of a question (depending on the question type).

J.4 Help Box

The Help Box has three states (see the section labeled “Help Box” in the Star GUI Specification document), and each of the states (except the Top-Level state which has the OK button only) has both the Back and OK buttons. The keyboard shortcuts in Table J.8 below are required for pressing the two buttons. Note that *Control* + *<key>* means the user holds down *Control* while pressing the key.

Keyboard Shortcut	Function
Control + B	Presses the Back button.
Control + K	Presses the OK button.

Table J.8: Keyboard shortcuts for the Help Box buttons

J.4.1 Top-Level State

Requirements 1 to 5 under the subsection labeled “Top-Level State” in the Star GUI Specification document are applicable under this section, in addition to the keyboard shortcuts in Table J.9.

Keyboard Shortcut	Function
C	Causes the Commands table to receive focus.
F	Clicks the command label in the first row of the table.
S	Clicks the command label in the current row of the table.
L	Clicks the command label in the last row of the table.
down arrow	Moves down to the command label of the next row without clicking it.
up arrow	Moves up to the command label of the previous row without clicking it.

Table J.9: Keyboard shortcuts for the Top-Level State’s Commands table

Whenever the Commands table receives focus, a red border is shown around it to serve as a visual cue for the user.

J.4.2 Command State

Requirements 1 to 10 under the subsection labeled “Command State” in the Star GUI Specification document are applicable under this section, in addition to the keyboard shortcuts in Table J.10, J.11, and J.12.

Keyboard Shortcut	Function
T	Causes the Text Area containing the <i>MultiLineHelp</i> for the current command to receive focus.
down arrow	Scrolls down the Text Area.

up arrow	Scrolls up the Text Area.
----------	---------------------------

Table J.10: Keyboard shortcuts for the Command State’s Text Area

Keyboard Shortcut	Function
P	Causes the Parameters table to receive focus.
F	Clicks the parameter label in the first row of the table.
S	Clicks the parameter label in the current row of the table.
L	Clicks the parameter label in the last row of the table.
down arrow	Moves down to the parameter label of the next row without clicking it.
up arrow	Moves up to the parameter label of the previous row without clicking it.

Table J.11: Keyboard shortcuts for the Command State’s Parameters table

Keyboard Shortcut	Function
Q	Causes the Questions table to receive focus.
F	Clicks the question label in the first row of the table.
S	Clicks the question label in the current row of the table.
L	Clicks the question label in the last row of the table.
down arrow	Moves down to the question label of the next row without clicking it.
up arrow	Moves up to the question label of the previous row without clicking it.

Table J.12: Keyboard shortcuts for the Command State’s Questions table

Whenever any widget in the Command State receives focus, a red border is shown around that widget to serve as a visual cue for the user.

J.4.3 Parameter/Question State

Requirements 1 to 6 under the subsection labeled “Parameter/Question State” in the Star GUI Specification document are applicable under this section, in addition to the keyboard shortcuts

in Table J.13.

Keyboard Shortcut	Function
T	Transfers focus to the Text Area containing the <i>MultiLineHelp</i> for the current Parameter/Question.
down arrow	Scrolls down the Text Area.
up arrow	Scrolls up the Text Area.

Table J.13: Keyboard shortcuts for the Parameter/Question State's Text Area

When the *MultiLineHelp* widget of the Parameter/Question state receives focus, a red border surrounds it.

J.5 Keyboard Shortcuts for interacting with certain Widgets

The following subsections define the keyboard shortcuts required to interact with widgets to be used in StarX for selecting (or typing in) values for each repetition of a parameter. Note that the keyboard shortcuts defined below automatically become active when the corresponding widgets receive focus.

J.5.1 Boolean Widget

This widget is used to select value for parameters (or questions) of type `boolean`. This widget is a *RadioButton* with “Yes” and “No” options.

Keyboard Shortcut	Function
Y	Selects the Yes option.
N	Selects the No option.

Table J.14: Keyboard shortcuts for the Boolean Widget

J.5.2 Choice Widget and TableEntry Widget

These widgets are used to select values for parameters (or questions) of type `choice` and `tableEntry` (with a non-browsable `SourceTable`) respectively. Each of these widgets is a *ComboBox* with a set of values.

Keyboard Shortcut	Function
-------------------	----------

V	Presses the drop-down arrow of the widget, thereby making the values (currently in the widget) visible. If some values are hidden in the current view, the arrow keys (see below) can be used to view them.
S	Sets the current value as the selected value.
down arrow	Moves down to the next value in the widget.
up arrow	Moves up to the previous value in the widget.

Table J.15: Keyboard shortcuts for the Choice/TableEntry Widget

J.5.3 Date Widget

This widget is used to select value for parameters (or questions) of type `date`. The widget is a *DatePicker* that consists of a textfield that shows the date and a button for displaying a calendar.

Keyboard Shortcut	Function
F5	Pops up the calendar.
down arrow	Selects the date of next or succeeding week.
up arrow	Selects the date of preceding week.
left arrow	Selects the date of preceding day.
right arrow	Selects the date of next or succeeding day.
Enter or Return	Confirms the selected date and shows it in the textfield.

Table J.16: Keyboard shortcuts for the Date Widget

J.5.4 File Widget

This widget is used to select value for parameters (or questions) of type `file`. The widget is a *FileChooser* with a textfield and a “browse” button (for file selection).

Keyboard Shortcut	Function
F5	Pops up the FileChooser Dialog Box.
Tab	Used to move around the FileChooser Dialog Box, such as moving from the <i>Look In</i> box to the <i>Folder & File List</i> box or to the <i>File Name</i> box or to the <i>File Type</i> box, etc.

down arrow	Selects the next file/folder.
up arrow	Selects the previous file/folder.
left arrow	Selects the file/folder to the left of the currently selected file/folder.
right arrow	Selects the file/folder to the right of the currently selected file/folder.
Enter or Return	If a folder is currently selected, pressing this key shows the content of the folder. Otherwise, if a file is currently selected, pressing this key confirms the selection and displays the absolute file name in the textfield.

Table J.17: Keyboard shortcuts for the File Widget

J.5.5 Number Widget

This widget is used to enter value for parameters (or questions) of type `int` and `float`. This widget is a *formatted textfield* that allows integers and floating-point (negative or non-negative) values only. On most platforms, the insertion point becomes available inside the widget whenever it receives focus. However, if the insertion point is not available when the widget receives focus, pressing the keyboard shortcut below will show it.

Keyboard Shortcut	Function
F6	Shows the insertion point in the widget, thereby making data entry possible.

Table J.18: Keyboard shortcuts for the Number Widget

J.5.6 Text Widget

This widget is used to enter value for parameters (or questions) of type `text`. This widget is a *textfield* that allows both single-line and multi-line texts. On most platforms, the insertion point becomes available inside the widget whenever it receives focus. However, if the insertion point is not available when the widget receives focus, pressing the keyboard shortcut below will show it.

Keyboard Shortcut	Function
-------------------	----------

F6	Shows the insertion point in the widget, thereby making data entry possible.
----	--

Table J.19: Keyboard shortcuts for the Text Widget

J.5.7 Time Widget

This widget is used to select value for parameters (or questions) of type `timeOfDay`. The widget is a *spinner* that shows the hour, minute, and AM/PM.

Keyboard Shortcut	Function
left arrow	Moves the insertion point to the left, and can be used to position the insertion point in front of the hour or minute or AM/PM.
right arrow	Moves the insertion point to the right, and can be used to position the insertion point in front of the hour or minute or AM/PM.
down arrow	If the insertion point is positioned in front of the hour or minute, pressing this key decreases the hour or minute by 1. Otherwise, if the insertion point is positioned in front of the AM/PM, pressing this key changes AM to PM or vice-versa.
up arrow	If the insertion point is positioned in front of the hour or minute, pressing this key increases the hour or minute by 1. Otherwise, if the insertion point is positioned in front of the AM/PM, pressing this key changes AM to PM or vice-versa.

Table J.20: Keyboard shortcuts for the Time Widget

J.5.8 The Message Dialog Box

The Message Dialog Box shows high-priority messages to the user. The dialog box has an *OK* button only.

Keyboard Shortcut	Function
Enter or Return	Presses the OK button.

Table J.21: Keyboard shortcut for the Message Dialog Box

J.5.9 Hotkeys Pop-Up Table

This table informs the user about the keyboard shortcuts or hotkeys that are applicable to various states of a user interface. Refer to the subsection labeled “The Hotkeys Pop-up Table” in Chapter 5 (under the “The StarX GUI” section) for more information about this dynamic table.

- (1) The table is activated when the user selects *Show Hotkeys Pop-up Table* under the *StarX* menu, and deactivated when the user selects the *Hide Hotkeys Pop-up Table*.
- (2) If activated, the table is displayed automatically as a pop-up immediately the user focuses on a section or launches a window in the user interface, and its content dynamically changes as the user transfers focus to interface elements within the active section or window.
- (3) The table can be dismissed or hidden by pressing the *ESC* or *Escape* key on the keyboard. Pressing *F1* shows the table again.

Appendix K

“Grupo” Interface Interpreter: Requirements Specification

Grupo is a “batch mode” interface interpreter that accepts a text file, containing all the desired commands and data, as input. Grupo reads each line of command from the input file, executes the command against the application engine, and then writes output messages to the standard output.

The syntax and semantics of Grupo commands allowed in an input file are discussed in the following sections.

K.1 Commands in Grupo

- (1) Commands, with their respective syntax, allowed in an input file are shown in Table K.1 below. Each of these commands occupies a separate line in the file.
- (2) All command keywords must be in lowercase.

Command	Syntax
browse	browse <tablename>
help	help <commandname parametername>
	help -b <commandname parametername>
	help -o <commandname parametername>
	help -m <commandname parametername>
table	table <tablename>
select	select <rownumber>
deselect	deselect <rownumber>
command	command <commandname>

param	param <parametername> <value>
ok	ok

Table K.1: Grupo Commands

K.1.1 The browse command

The `browse` command displays the content of a browsable table to the user.

- (1) Its syntax consists of the *browse* keyword followed by a space, and the name of the table to display.
- (2) The table with the specified name is one of the browsable tables in the IDF.

K.1.2 The help command

The `help` command displays the *BriefHelp*, *OneLineHelp*, or the *MultiLineHelp* of a Command or Parameter to the user.

- (1) A syntax of the `help` command consists of the *help* keyword followed by a space, an option followed by a space, and the name of a Command or Parameter in the IDF.
- (2) Another syntax of the `help` command consists of the *help* keyword followed by a space, and the name of a Command or Parameter in the IDF.
- (3) If an option is specified in the `help` command, it can be either `-b` (for *BriefHelp*), or `-o` (for *OneLineHelp*) or `-m` (for *MultiLineHelp*).
- (4) If an option is not specified in the `help` command, then the `-o` option is used.

K.1.3 The table command

The `table` command makes a table to be the current table.

- (1) The syntax of the `table` command consists of the *table* keyword followed by a space, and the table name.
- (2) The table with the specified name is one of the tables in the IDF.

K.1.4 The **select** command

The **select** command selects a row in the current table.

- (1) Its syntax consists of the *select* keyword followed by a space, and the row number to select.
- (2) The current table should have been set (see above) before issuing this command.
- (3) The row number is a number n , where $n \geq 1$. Thus, row number 1 represents the first row, 2 represents the second row, 3 represents the third row, and so on.
- (4) In order to select n rows from the current table, n **select** commands will be issued.

K.1.5 The **deselect** command

The **deselect** command deselects a previously-selected row in the current table.

- (1) Its syntax consists of the *deselect* keyword followed by a space, and the row number to deselect.
- (2) The specified row should have been selected (see above) before deselecting it using this command.
- (3) The row number is a number n , where $n \geq 1$.
- (4) In order to deselect n selected rows in the current table, n **deselect** commands will be issued.

K.1.6 The **command** command

Begins input of a command.

- (1) Its syntax consists of the *command* keyword followed by a space, and the command name.
- (2) The command with the specified name is one of the application’s commands already defined in the IDF.
- (3) Each input of the command is specified via the **param** command (see below).
- (4) After the last input of the command has been specified, the **ok** command (see below) is issued to signify the end of input.

K.1.7 The `param` command

The `param` command sets the value of a parameter.

- (1) Its syntax consists of the *param* keyword followed by a space, the name of the parameter followed by a space, and the parameter value.
- (2) If the parameter value is a string that contains one or more spaces, then it should be enclosed in double quotation marks.
- (3) The parameter with the specified name is one of the parameters of the command already defined using the `command` command (see above).
- (4) The `param` command is always placed between the `command` command and the `ok` command.
- (5) In order to set value for n repetitions of a parameter, n `param` commands will be issued to set values for the repetitions.

K.1.8 The `ok` command

The `ok` command signifies the end of input of a command.

- (1) Its syntax consists of the *ok* keyword only.
- (2) It is always placed after the `command` command and should be after the `param` command (if any) on a separate line.
- (3) An `ok` command is attached to the preceding and most recent `command` command only.

K.2 Output Message Prefixes

See Table K.2. All output messages shown to the user are prepended with the prefixes that fit their purpose. Each of the prefixes is immediately followed by a colon and a space (e.g. *ERR: invalid parameter name*) to improve readability. Note that all output messages are displayed on the Standard Output.

Prefix	Short Meaning	Purpose
COM	Command Input	Informs the user that the output message is an input of a command.

OUT	showText Output	Informs the user that the output message is the output of showText.
TAB	Table Output	Informs the user that the output message is the content of a particular table.
ERR	Error Message	Informs the user that the output message is an error that occurred during execution.
REM	Remark/Comment	Informs the user that the output message is a comment or remark.
HLP	Help	Informs the user that the output message is the help information he/she requested.

Table K.2: Prefixes of output messages

K.3 The Input File

Commands are entered into a text file.

- (1) The file name is made up of letters and/or digits and/or the underscore.
- (2) The file name has the “.gpo” extension.
- (3) A line does not contain more than one command.
- (4) A line may contain only the whitespace.
- (5) A line where the first non-whitespace is “//” is interpreted as a comment.

K.3.1 A Sample Input File

See the Sample Input File in Chapter 5 under the “Grupo Interface Interpreter” section.

Appendix L

“Grupo” Interface Interpreter: Requirements Specification (Behaviour)

L.1 Top-Level Behaviour

- (1) Read the IDF.
- (2) Create the `GemSetting` and validate it.
- (3) Call the application engine’s initialization method.

L.2 Running Commands in an Input File

When the user launches Grupo:

- (1) If an input file name with “.gpo” extension is specified by the user,
 - (i) Collect the file name in a variable `fileName`.
 - (ii) If `fileName` has a relative path, then make it absolute. (Idea: `fileName = <path to current/working directory> + fileName`).
 - (iii) If the input file referenced by `fileName` exists,
 - (a) Set a String field `currentTable` to the empty string. (Rationale: this will be for storing the name of the current table).
 - (b) Instantiate a Map field `tablesMap`. (Rationale: this will be for capturing each `currentTable` along with a list of selected rows in that table. Thus, each key in `tablesMap` is a string representing a table name, and the corresponding value is a List containing the row numbers of selected rows. In Java,

`tablesMap` can be a hash map of the form `HashMap<String, List<Integer>>`).

- (c) For each line of text in the input file, perform the “Parse-and-Execute Command” procedure with the line of text as parameter.
 - (iv) Otherwise, show an error message informing the user that the specified input file does not exist.
- (2) Otherwise, show an error message informing the user of missing input file; afterwards, call `System.exit(0)` to exit `Grupo`.

L.3 The Parse-and-Execute Command Procedure

The Parse-and-Execute Command procedure takes a parameter: the command line (`String commandLine`).

- (1) Remove any whitespace before and after `commandLine`.
- (2) If `commandLine` is empty, then Return.
- (3) Otherwise, if `commandLine` starts with “//”, then show the text after the “//” to the user, and Return.
- (4) Otherwise,
 - (i) Tokenize `commandLine`.
 - (a) If the first token is *table*,
 - (i) If the second token (i.e. the table name) exists in the IDF, then assign the second token to `currentTable`. Otherwise, inform the user (via an error message) that the specified table name does not exist in the IDF, and Return.
 - (ii) If `currentTable` is not in `tablesMap`, then put `currentTable` and a new list in `tablesMap`.
 - (iii) Return.
 - (b) Otherwise, if the first token is *select*,
 - (i) If `currentTable` is in `tablesMap`,
 - (a) Call `gemSetting.rowIsFilled` to determine whether the specified row number (i.e. second token) is filled in the current table. (Remember that row number in `GemSetting` is zero-based).

- (b) If the row is filled, then add the second token to the list of row numbers associated with `currentTable` in `tablesMap`.
 - (c) Otherwise, show an error message to the user stating that the specified row is not filled. The error message should also contain the line number that triggers the error.
 - (d) Return.
- (ii) Otherwise, show an error message to the user stating that no current table exists. The error message should also contain the line number that triggers the error.
 - (iii) Return.
- (c) Otherwise, if the first token is *deselect*,
 - (i) If `currentTable` is in `tablesMap`, then remove the second token from the list of row numbers associated with `currentTable` in `tablesMap`.
 - (ii) Otherwise, show an error message to the user stating that no current table exists. The error message should also contain the line number that triggers the error.
 - (iii) Return.
 - (d) Otherwise, if the first token is *command*,
 - (i) If the second token (i.e. the command name) exists in the IDF and a call to `gemSetting.methodIsActive` returns true, then
 - (a) Assign the second token to a String field `currentCommand`.
 - (b) Call `gemSetting.selectCurrentCommand(currentCommand)`
 - (ii) Otherwise, show appropriate error message to the user.
 - (iii) Return.
 - (e) Otherwise, if the first token is *param*,
 - (i) If the second token (i.e. parameter name) is not in the IDF, show an error message to the user, stating the invalid parameter and the corresponding line number; and Return.
 - (ii) Show `commandLine` to the user.
 - (iii) Remove any double quotation marks that enclose the parameter value (i.e. the third token).
 - (iv) Convert the parameter value to conform to the type equivalent to the parameter's type in the IDF. (See the table in the Interface Interpreter Specification document available in Appendix B).

- Show any error(s) that occurred during conversion to the user, and Return.
- (v) Validate the parameter value, as indicated in requirements 26-51 of the Interface Interpreter Specification document.
- If the value does not pass validation, show an error message to the user, and Return. (The error message must contain the line number, parameter name, and the value that failed validation).
- (vi) Call `gemSetting.getParameterRepCount` with the second token as the parameter. Store the return value in a variable `repNumber`. (Rationale: Since the repetition number is required when loading a value into the Gem, then the number returned by `getParameterRepCount` can be used as the repetition number for that value.)
- If `GemException` is thrown, then set `repNumber` to be 0. (Rationale: Since the parameter does not exist in the Gem, then use 0 as the repetition number.)
- (vii) Load the parameter value into the Gem with `repNumber` as the repetition number.
- (viii) Return.
- (f) Otherwise, if the first token is *ok*,
- Call the “Execute App Command” procedure with parameter `currentCommand`.
- (g) Otherwise, if the first token is *browse*,
- (i) Collect the entire line in a variable `browseCmdString`, removing any whitespace before and after the string.
- (ii) Call the “Execute Browse Command” procedure with parameter `browseCmdString`.
- (iii) Return.
- (h) Otherwise, if the first token is *help*,
- (i) Collect the entire line in a variable `helpCmdString`, removing any whitespace before and after the string.
- (ii) Call the “Execute Help Command” procedure with parameter `helpCmdString`.
- (iii) Return.

L.4 Execute App Command Procedure

The Execute App Command procedure takes one parameter: command name (String `commandName`).

- (1) For each stage i in the command,
 - (i) Call `gemSetting.selectCurrentStage(i)`
 - (ii) For each parameter p in stage i ,
 - (a) If p is a `tableEntry` parameter,
 - (i) Get its `SourceTable` from the IDF
 - (ii) For each key k in `tablesMap`,
 - If k is equal to `SourceTable`, then
 - (a) If the size of the list associated with k is less than `MinNumberOfReps` of p , then show an error message to the user stating that the table selection for p is incomplete, and Return.
 - (b) Otherwise, load each element (i.e. row number) in the list associated with k into the Gem. Remember to subtract 1 from each element before loading it into the Gem. (Rationale: Since row number in `GemSetting` is zero-based, it is imperative to subtract 1 from each row number before loading it into the Gem).
 - (b) Otherwise,
 - (i) Call `gemSetting.getParameterRepCount(p)` and assign its return value to a variable `currentNumberOfReps`.
 - (ii) If `currentNumberOfReps` is less than `MinNumberOfReps` of p ,
 - (a) If p has a `DefaultValue`, then fill up the remaining repetitions with `DefaultValue` up to `MinNumberOfReps`.
 - (b) Otherwise, if p has a `DefaultValueMethod`, call it and use the return value to fill up the remaining repetitions up to `MinNumberOfReps`.
 - (c) Otherwise, show an error message to the user stating the parameter with incomplete repetition, and Return.
 - (iii) Call the `ParameterCheckMethod` for stage i (if any),
 - If the `ParameterCheckMethod` returns an error message, show it to the user, and Return.
- (2) Call the `CommandMethod` of the command.

- (3) For each table t in the IDF,
 - (i) Determine if t has changed by calling `gemSetting.tableIsUpdated(t)`.
 - (ii) If t has changed, then show its contents (with header and column name(s)) to the user.
- (4) Call `gemSetting.getTopTable` to retrieve the current top table, and display “Current Top Table: ” followed by the name of the top table to the user.
- (5) Determine whether the application should quit, by checking the `QuitAfter` attribute and/or calling the `QuitAfterIfMethod` of the command.
 - (i) If the application should quit, then call the “Execute Exit App” procedure.
 - (ii) Return.

L.5 Execute Browse Command Procedure

The Execute Browse Command procedure takes one parameter: the browse command string (`String browseCommandString`).

- (1) Tokenize `browseCommandString`.
- (2) If the number of tokens is 2,
 - (i) If the second token (i.e. the table name) is a browsable table in the IDF and has content in the Gem, then show its contents (with header and column name(s)) to the user.
 - (ii) Otherwise, show appropriate error message to the user.
 - (iii) Return.
- (3) Otherwise, show an error message informing the user that the browse command is wrongly used. The error message should further guide the user by displaying the correct syntax of the browse command.

L.6 Execute Help Command Procedure

The Execute Help Command procedure takes one parameter: the help command string (`String helpCommandString`).

- (1) Tokenize `helpCommandString`.
- (2) If the number of tokens is 2,
 - (i) If the second token (i.e. the Command or Parameter name) exists in the IDF, then show its `OneLineHelp` to the user.
 - (ii) Otherwise, show an error message informing the user that the specified Command or Parameter name does not exist in the IDF.
 - (iii) Return.
- (3) Otherwise, if the number of tokens is 3,
 - (i) If the second token is `-b` and the third token (i.e. the Command or Parameter name) exists in the IDF, then show the `BriefHelp` of the specified Command or Parameter to the user.
 - (ii) Otherwise, if the second token is `-o` and the third token (i.e. the Command or Parameter name) exists in the IDF, then show the `OneLineHelp` of the specified Command or Parameter to the user.
 - (iii) Otherwise, if the second token is `-m` and the third token (i.e. the Command or Parameter name) exists in the IDF, then show the `MultiLineHelp` of the specified Command or Parameter to the user.
 - (iv) Otherwise, show appropriate error message to the user.
 - (v) Return.
- (4) Otherwise, show an error message informing the user that the `help` command is wrongly used. The error message should further guide the user by displaying the correct syntax of the `help` command.

L.7 Execute Exit App Procedure

- (1) Show “Exited ”, followed by the camel-case translation of the Application name, to the user. For example, *Exited Temperature Converter*.
- (2) Call `System.exit(0)`.

L.8 The ShowTextHandler

The ShowTextHandler for Grupo handles the display of text on the Standard Output (System.out), irrespective of the prominence of the text.

Curriculum Vitae

Name: Oladapo Oyebode

Post-Secondary Education and Degrees: Western University
London, Ontario, Canada
2012–2013 M.Sc. in Computer Science

University of Ibadan, Nigeria
2005–2008 B.Sc. in Computer Science

Federal Polytechnic Ede, Nigeria
2002–2004 National Diploma in Computer Science

Honours and Awards: Faculty of Science Dean’s Roll of Honour (for Academic Excellence)
University of Ibadan, Nigeria
2006 and 2007

Related Work Experience: Teaching Assistant and Research Assistant
Western University
2012–2013

Software Developer at Sterling Bank Plc, Lagos, Nigeria
2010–2012

IT Solutions Developer at HeteroGenius Systems Limited, Nigeria
2010