ACKNOWLEDGEMENTS

This work has been carried out with the continuous technical support from Anthony Savidis, being the designer and developer of the I-GET UIMS, as well as the lead developer of the HAWK non-visual toolkit.
# Table of Contents

ACKNOWLEDGEMENTS ........................................................................................................... I

TABLE OF CONTENTS ............................................................................................................ II

LIST OF FIGURES ................................................................................................................ V

EXECUTIVE SUMMARY ...................................................................................................... VI

CHAPTER 1 ............................................................................................................................. 1

INTRODUCTION ....................................................................................................................... 1

1.1 OBJECTIVE AND MILESTONES .................................................................................. 1

1.2 RELATED WORK .............................................................................................................. 2

1.3 THE I-GET UIMS ......................................................................................................... 5

  1.3.1 Run-time architecture ........................................................................................... 8
  1.3.2 The I-GET language and development tools ..................................................... 10
  1.4 THE DEVELOPMENT NEEDS REGARDING A WINDOWS PORTING .................... 12

1.4.1 Windows programming interface specification .................................................... 13

1.4.2 Toolkit server development .................................................................................. 16

1.4.3 Extensible Interactive programming environment ............................................. 18

1.4.4 Large-scale applications tests. ............................................................................. 20

CHAPTER 2 ........................................................................................................................... 22

THE ADVANCED WINDOWS TOOLKIT SERVER ................................................................. 22

2.1 ARCHITECTURE ............................................................................................................. 22

2.2 AUTOMATICALLY GENERATED SOURCE CODE ..................................................... 26

2.3 IMPLEMENTATION ....................................................................................................... 26

  2.3.1 Toolkit initialization and Main Loop ....................................................................... 27
  2.3.2 Instantiating Objects ............................................................................................ 28
  2.3.3 Handling Attribute Modification ........................................................................ 31
  2.3.4 Handling Output Events ..................................................................................... 33
  2.3.5 Event Interest Handlers ....................................................................................... 35
  2.3.6 Input Event Notifiers .......................................................................................... 38
  2.3.7 Attribute Modification Handlers ......................................................................... 40

  2.3.8 Method Notifiers ................................................................................................. 41

  2.4 MAINTENANCE AND EXTENSIBILITY ...................................................................... 42

CHAPTER 3 ........................................................................................................................... 44

THE INTERACTIVE PROGRAMMING ENVIRONMENT ......................................................... 44

3.1 EXTENSIBLE COMPONENT-BASED ARCHITECTURE .............................................. 44
## Table Of Contents

3.1.1 Editor Library Manager ................................................................. 45  
3.1.2 Undo / Redo Stack Manager ............................................................ 46  
3.1.3 Clipboard Manager ...................................................................... 46  
3.1.4 Find / Replace / Goto-Line Manager ............................................... 47  
3.1.5 Project Manager .......................................................................... 49  
3.1.6 Syntax Highlighting Manager .......................................................... 50  
3.1.7 Compiler Manager ........................................................................ 51  
3.1.8 Tools Manager ............................................................................. 51  
3.2 Multi-DocuMENT SyNTaX HiLItING EDItoR ...................................... 52

CHAPTER 4 ................................................................................................................ 59

APPLICATION TEST SUITE ................................................................................ 59

4.1 SCIENTIFIC CALCULATOR APPLICATION ........................................... 59  
4.1.1 General description ........................................................................ 59  
4.1.2 Scientific Calculator User Interface .................................................. 61  
4.1.3 Purpose of testing .......................................................................... 62

4.2 GRAPHICS DRAWING APPLICATION ................................................... 63  
4.2.1 General description ........................................................................ 63  
4.2.2 Graphics Drawing Application User Interface .................................. 64  
4.2.3 Purpose of testing .......................................................................... 66

4.3 GEOMETRICAL CONSTRAINTS APPLICATION ..................................... 68  
4.2.1 General description ........................................................................ 68  
4.2.2 Geometrical Constraints Application User Interface ....................... 69  
4.2.3 Purpose of testing .......................................................................... 69

CHAPTER 5 ................................................................................................................ 72

SUMMARY, CONCLUSIONS ............................................................................. 72

& FUTURE WORK ................................................................................................. 72

5.1 SUMMARY ........................................................................................... 72  
5.2 CONCLUSIONS .................................................................................. 74  
5.3 FUTURE WORK ................................................................................... 77

APPENDIX A ....................................................................................................... 79

CODE EXCERPTS FROM THE SCIENTIFIC CALCULATOR APPLICATION ... 79

APPENDIX B ....................................................................................................... 89

CODE EXCERPTS FROM THE GRAPHICS DRAWING APPLICATION .......... 89

APPENDIX C ....................................................................................................... 101

CODE EXCERPTS FROM THE GEOMETRICAL CONSTRAINTS APPLICATION ........................................... 101
| REFERENCES | 106 |
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Dual run-time model.</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>The three fundamental layers in man-machine interaction.</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Run-time UIMS architecture for interactive applications.</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>The run-time architecture of interactive applications developed through the I-GET tool.</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Code excerpts from the Windows Toolkit Interface Specification.</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>Components of the Editor Toolkit.</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>Advanced Windows Toolkit Server architecture.</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>Code excerpts from winstr.hh.</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>Inithinstance () function.</td>
<td>27</td>
</tr>
<tr>
<td>10</td>
<td>The OnIdle (LONGLTIME) function.</td>
<td>28</td>
</tr>
<tr>
<td>11</td>
<td>The TS_ServeInstantiateObject () function.</td>
<td>29</td>
</tr>
<tr>
<td>12</td>
<td>Instantiation functions and corresponding dispatch table.</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>The instantiation function for the Button object.</td>
<td>31</td>
</tr>
<tr>
<td>14</td>
<td>The TS_CommonAttributeModify () function.</td>
<td>32</td>
</tr>
<tr>
<td>15</td>
<td>The attribute modification function Set_Enable ().</td>
<td>33</td>
</tr>
<tr>
<td>16</td>
<td>The TS_ClearList_OutputEvent (GWOUT_ClearList&amp; p) function.</td>
<td>33</td>
</tr>
<tr>
<td>17</td>
<td>The TS_AllocateGraphics_OutputEvent (GWOUT_AllocateGraphics&amp; p) function.</td>
<td>34</td>
</tr>
<tr>
<td>18</td>
<td>The TS.ServeCancelEventInterest () and TS.ServeReportEventInterest () functions.</td>
<td>36</td>
</tr>
<tr>
<td>19</td>
<td>The Event interest handlers dispatch tables and the handlers CancelEventPaint (), for canceling event interest for the Paint event and ReportEventSize (), for reporting event interest for the Size event.</td>
<td>37</td>
</tr>
<tr>
<td>20</td>
<td>Input event notifier for the event LeftButtonPressed for the Button object.</td>
<td>38</td>
</tr>
<tr>
<td>21</td>
<td>Attribute Modification notifier for the Button object.</td>
<td>39</td>
</tr>
<tr>
<td>22</td>
<td>Attribute Modification handler for the text colour attribute, for the Button object.</td>
<td>40</td>
</tr>
<tr>
<td>23</td>
<td>Method notifier for the “Edited” method of the ComboBox object.</td>
<td>41</td>
</tr>
<tr>
<td>24</td>
<td>The Find Dialog.</td>
<td>47</td>
</tr>
<tr>
<td>25</td>
<td>The Replace Dialog.</td>
<td>48</td>
</tr>
<tr>
<td>26</td>
<td>The Goto-Line Dialog.</td>
<td>49</td>
</tr>
<tr>
<td>27</td>
<td>The I-GET Editor.</td>
<td>53</td>
</tr>
<tr>
<td>28</td>
<td>Windows arranged as overlapping tiles and minimized windows as icons.</td>
<td>54</td>
</tr>
<tr>
<td>29</td>
<td>Windows arranged as horizontal non-overlapping tiles and as vertical non-overlapping tiles.</td>
<td>55</td>
</tr>
<tr>
<td>30</td>
<td>The New Window feature.</td>
<td>56</td>
</tr>
<tr>
<td>31</td>
<td>The Format dialog.</td>
<td>56</td>
</tr>
<tr>
<td>32</td>
<td>The default Windows colour and font dialogs.</td>
<td>57</td>
</tr>
<tr>
<td>33</td>
<td>The Tools dialog.</td>
<td>58</td>
</tr>
<tr>
<td>34</td>
<td>The Scientific Calculator Application.</td>
<td>60</td>
</tr>
<tr>
<td>35</td>
<td>The Graphics Drawing Application.</td>
<td>63</td>
</tr>
<tr>
<td>36</td>
<td>The Tool box.</td>
<td>64</td>
</tr>
<tr>
<td>37</td>
<td>The drawing options.</td>
<td>65</td>
</tr>
<tr>
<td>38</td>
<td>The bottom pane.</td>
<td>66</td>
</tr>
<tr>
<td>39</td>
<td>The Geometrical Constraints Application.</td>
<td>68</td>
</tr>
</tbody>
</table>
EXECUTIVE SUMMARY

Within the wider scope of development efforts towards an Information Society for All, the technical challenge of universal access is one of the key goals. This goal can be practically reached through the citation and continuous support, of the development environments that offer innovative features, particularly suited to the construction of universally accessible interactions. In this perspective, the overall objective has been to support and promote such development environments. To meet this objective, it has been necessary to choose one such development tool, i.e. the I-GET UIMS being arguably the most promising tool for universal access, and subsequently, carry out a number of appropriate supporting development activities, to enhance its portability, robustness and ease of use.

The I-GET UIMS is a language-based UIMS that, amongst other facilities, enables developers to incorporate interaction toolkits through a well-defined toolkit integration process. The initial development of the I-GET UIMS has been carried out under UNIX, while the first toolkit that has been incorporated was the Xt/Xaw Athena widget set. As part of the porting effort for the Windows 2000 platform, the integration of the Windows 2000 interaction controls’ collection has been one key task. This latter, following the technical approach of the I-GET UIMS for toolkit integration, required the development of a respective toolkit server.

In this context, the primary concern of the reported work has been the design and implementation of an Advanced Windows Toolkit server for Windows 2000. Such a
toolkit server had to be designed in a way that it could easily support further extensions, while accomplishing high performance. Additionally, the development of a comprehensive application test-suite has been necessary, played a twofold role: (a) to serve as a test-bed for verifying the innovative features of the I-GET language; and (b) to perform extensive testing of the toolkit server.

Following the need for large-scale testing of the I-GET UIMS, the lack of an interactive programming environment (for the I-GET UIMS) has led to the design and implementation of the I-GET Interactive Programming Environment, delivered as a flexible source-code editor and project management tool for the I-GET language.

Some of the facilities offered by the Interactive Programming Environment are: (a) syntax highlighting; (b) workspace management; (c) editing of compiler flag files; (d) linkage to external tools; and (e) multi-document support (i.e. concurrent editing).
CHAPTER 1
INTRODUCTION

1.1 Objective and Milestones

The overall objective has been to practically support development environments for universal access. For this purpose, the I-GET UIMS has been chosen as the most appropriate subject environment, due to the advanced set of development features that it offers for the engineering of universally accessible interactions (Savidis & Stephanidis, 2000, [9]).

In this context, the development work carried out has targeted the following three key milestones, aiming at enhancing the portability, ease of use, and robustness of the I-GET UIMS:

(i) The development of an advanced toolkit server for the Microsoft Windows 2000 platform that can be easily extended to incorporate forthcoming features such as the support of multimedia files.

(ii) To provide a flexible interactive programming environment for the I-GET UIMS, enabling source-code editing and project management, with features such as syntax highlighting of language keywords.
(iii) To develop a comprehensive application test-suite to verify the powerful features of the I-GET UIMS, as well as extensive testing of the toolkit server.

1.2 Related Work

The overall objective in this work has been to support development environments for universal access. The I-GET UIMS has been chosen as the most appropriate target environment, due to the advanced set of development features that it offers for the engineering of universally accessible interactions (Savidis & Stephanidis, 2000, [9]).

In this context, the reported work had a very specific focus, being directly linked to the I-GET UIMS. The result of this work aimed to: (a) support the easier utilization of the I-GET UIMS; (b) provide necessary run-time components for porting to the Windows 2000 platform; and (c) provide large applications built with the I-GET language, that constituted both test-cases, as well as valuable development examples for other I-GET UIMS users.

In this perspective, while similar work in the UIMS arena did not affect the technical approach of the reported developments, it has been necessary to investigate existing practices and methods in the UIMS field, so as to familiarize with the available instruments and methodologies. Below, we briefly review various key UIMS tools.

The Alpha UIMS [6], which follows the Seeheim [1][3] and Arch [10] UIMS models, is a multi-platform development and execution environment specifically designed to help experienced software developers create and maintain application systems. Alpha completely separates the application domain from the physical characteristics of the
user interface technology, eliminating the need for any kind of programming with “callbacks” to induce interface behaviors. This complete separation of domains, coupled with Alpha dependency-based, object-oriented programming approach, reduces the number of lines of code that needed to be written or generated by 40-90%. Additionally, it decreases the time and resources ordinarily needed for development and maintenance, and enhances the quality and usability of the final products.

Another User Interface development tool belonging to the category of platform-independent, language-based UIMS tools is the HOMER UIMS [8], which facilitates the development of Dual User Interfaces, i.e. interfaces with a component at run-time visual and non-visual instantiation. Additionally, apart from Dual Interfaces, it supports the development of separate visual and non-visual User Interfaces as well.

![Diagram of the Dual run-time model](image)

*Figure 1: The Dual run-time model.*
The HOMER UIMS supports the integration of visual and non-visual lexical technologies, while it facilitates concurrently communication with a visual and non-visual lexical technology at run-time. The dialogue specification is carried out in the HOMER language that supports abstraction of the physical constructs, which are introduced through lexical technology integration, to completely metaphor-independent interaction objects. Consequently, sub-dialogues, which can be described with such abstract objects, are specified with shared descriptions for both the visual and non-visual environments.

The HOMER UIMS run-time architecture is based on the Dual run-time model (figure 1), which forms an extension of the Arch model for handling a Dual lexical layer. The HOMER UIMS is the predecessor of the I-GET UIMS.

The MIKE UIMS [2] does not use the semantic specifications found in most UIMS tools. Instead it provides a default syntax that is automatically generated from the definition of the semantic commands (i.e. functions) that the interaction is to support. The default syntax is refined using an interface editor that allows modification of the presentation of the interface.

The Picasso-3 UIMS [4][5] is a versatile software tool for developing and displaying dynamic graphical User Interfaces, particularly aimed at process surveillance and control systems. It runs under Microsoft Windows and various UNIX flavours including Linux, HP-UX, Solaris, Irix and Digital-Unix. Picasso-3 supports object-oriented definitions of dynamic graphical user interfaces, enabling the designer to visualise process states. Graphics, dynamic behaviour and operator dialogues are defined using an advanced GUI editor, and any aspect of the GUI can be linked dynamically to process parameter values. To support definitions of the dynamic
behaviour, the GUI designer is provided with a full-featured programming language called pTALK. pTALK has the syntax and expressive power of C++, and additionally includes constructs for graphics manipulations.

At run-time, Picasso-3 visualises dynamic GUIs on the operators’ screens, updates graphics according to the dynamic behaviour whenever new process values are received, and handles operator input according to the designer’s definitions. Additionally, Picasso-3 can be connected to simulators, SCADA (Supervisory Control and Data Acquisition) systems or real-time databases using an open, high-level Application Programmer’s Interface (API). Integration using industry standards such as OPC (OLE (object linking and embedding) for Process Control) is supported. Picasso-3 and the API are optimised to handle large amounts of data and frequent display updates.

1.3 The I-GET UIMS

In the early 80s a new category of User Interface development tools was introduced, the User Interface Management Systems (UIMS) [7]. The I-GET tool falls in that category and, as all UIMS tools do, offers development methods that are characterized as “higher-level” than typical programming languages, for interface development. One distinctive property of UIMS tools is that they usually share a common architectural vision, a meta-model; there have been two major UIMS architectural meta-models: the Seeheim model (1983) [1][3], and the Arch model (1992) [10]. Both of them are based on similar architectural principles, while the Arch model can be
seen as a specialization of the Seeheim model, rather than as an advance or enhancement.

One key principle of UIMS tools is the notion of separation between interactive and non-interactive code of software application. Non-interactive code concerns those “parts” of the implementation, which do not deal with the aspects of man-machine interaction; these parts have been traditionally called functional core or application components. This principle of separation has been a key requirement for most UIMS tools, while the way in which this separation could be either logically or implementationally preserved constituted a subject of debate for many years across UIMS researchers. The basis of the typical UIMS structure architectures has been actually this principle of separation.

![Figure 2: The three fundamental layers in man-machine interaction.](image-url)
Another key architectural principle of UIMS tools has been the layered view of an interactive application. Three main logical layers have been identified, with boundaries between them always loosely defined: semantic, syntactic and lexical (figure 2). These layers are defined as follows:

**The Semantic layer** concerns the original underlying system functionality that is offered to the user in an interactive form.

**The Syntactic layer** is related to the structure of the dialogue between the user and the interactive application, when seen in the perspective of a language (for communicating input and output) with its own syntactic rules.

**The Lexical layer** is related to the nature of physical interaction primitives involved in user-computer dialogue, analogous to the notion of lexical tokens (e.g. words, letters) in verbal languages.

![Diagram of UIMS architecture](image_url)

*Figure 3: Run-time UIMS architecture for interactive applications.*
Chapter 1: Introduction

The basic UIMS architecture is shown in figure 3 and is mainly related to the Arch meta-model. It is clear that this meta-model reflects the three fundamental layers involved in man-machine dialogue. This meta-model is related to the run-time structure of interactive applications built via a UIMS tool and not to the architecture of the UIMS system itself as a development environment / tool.

The components identified are: Application component, Application interfacing, Dialogue control, Toolkit interface and Toolkit. This is the architecture that the I-GET UIMS follows and the components are described in details within the next paragraph.

1.3.1 Run-time architecture

The I-GET tool consists of five run-time components (illustrated in figure 4).

![Figure 4: The run-time architecture of interactive applications developed through the I-GET tool.](image-url)
These components are:

?? The *Toolkit server*, which is the low-end software library providing, implemented interaction elements (e.g. Windows Object Library, OSF/Motif, Xt/Athena).

?? The *Toolkit Interface*, this component of the *I-GET* tool is not realized via a separate running process. Instead, it is based on a special purpose semantic protocol, called *Generic Toolkit Interfacing Protocol* (GTIP). This protocol has been designed in order to enable physical separation between the programming interface of a toolkit (i.e. object classes, data types and procedures), and the real implementation of physical interaction elements (i.e. rendering, device handling, display algorithms).

?? The *Dialogue Control*, which is the component that implements the interface between the user and the application component. Key issues to be addressed by such an implementation are: interface components and their structure, dialogue sequencing, presentation of data originated by the application, feedback, etc. The Dialogue Control specification is translated, by the *I-GET* compiler, into a C++ implementation, which can be compiled and linked (with the run-time library) into an executable file by using a C++ compiler.

?? The *Application Component*, which collects together all the various modules supplying non-interactive functionality. The *I-GET* tool provides a “template” structure requiring implementation of a small set of special-purpose C++ functions, which are the minimal requirements for making an Application Component for the *I-GET* run-time architecture. An Application Component
never communicates directly with the Dialogue Control component, but it may either “access” the shared space or post messages in a message channel.

?? The Application Interfacing, the role of this component is to coordinate and mediate communication between the Application Component and the Dialogue Control component; in this context, it maintains the space of shared objects and controls “flow” of messages through the message channels.

1.3.2 The I-GET language and development tools

I-GET is a language-based UIMS for dialogue control implementation. Hence, it requires that interface developers write explicitly interface code in the I-GET dialogue specification language. Interactive construction facilities are not provided, since this would require a specific toolkit to be considered, an approach conflicting with one primary objective of the I-GET tool, being openness with respect to underlying utilized toolkits. The I-GET language offers a variety of features for dialogue implementation. Key interface concepts such as “object”, “event”, “dialogue component”, “callback”, etc, directly map to built-in entity categories, while declarative programming constructs such as “constraints”, “monitor”, and “precondition” are explicitly supported and play a central role. The I-GET language consists of four key logical layers of constructs, which can be utilized and mixed by the dialogue developer in any manner (hence, there is no pre-imposed layered implementation structure that must be followed). These four layers are:

?? The API Layer, which includes all the supplied language constructs for specification of the application interfacing space. The application interfacing
space is defined by the definition of an arbitrary number of typed channels (i.e. channel names and data-types communicated), and an arbitrary number of shared data-types (i.e. types of items which may be exposed in shared space).

?? The Agent Layer, which concerns the specification of agent classes. In the I-GET language, agent classes are actually dialogue control component classes. An agent class can be generally seen as similar to class definitions in typical Object Oriented Programming (OOP) languages. Instantiation of such classes can be done either via preconditions or explicitly through instantiation statements. Agents may be hierarchically structured, while in the body of agent classes various constructs may be defined (e.g. object instances, event handlers, call-backs, constraints, monitors, member functions, data types).

?? The Objects Layer, which is related to the levels of interaction objects supported in the I-GET language. Physical objects may be either imported by an underlying toolkit (other categories of elements apart from objects may be also imported) or can be implemented explicitly by utilizing already physical interaction elements (e.g. graphics facilities and device input). Virtual objects is a built-in category of abstract objects that are totally relieved from physical properties. New virtual objects can be defined, while the mapping logic to physical classes must be provided through instantiation schemes.

?? The Common Layer, which provides language constructs that can be employed in any of the above three layers. The common layer supplies four major categories of language elements: (i) programming kernel, which supports built-in data types, data type definition, functions various
statements and expressions; (ii) constraints and monitors, which support the definitions of constraints and monitors on any type of variable; (iii) hooks and bridges, which support the powerful mechanisms for mixing I-GET language with C / C++; and (iv) prototype / implementation facility, which supports separate compilation and incremental development of dialogue implementation units.

The I-GET tool provides three main components: (i) the I-GET compiler, for translating I-GET languages specifications into C++; (ii) the run-time library, consisting of standard software libraries for the components of the run-time application architecture; and (iii) the development library, which provides the necessary support for integrating toolkits and building application components. It explicitly supports toolkit interfacing, by providing a toolkit integration facility, while it also enables the concurrent utilization (for implementing dialogue control components) of multiple toolkits, after these toolkits have been integrated through the toolkit integration process. Also, supports application interfacing, by providing both a shared space of typed objects as well as typed message channels. The application interfacing process is fixed (hence, it is supplied by the I-GET software distribution), while the type of communicated objects needs to be specified as part of application interface specification.

1.4 The Development Needs Regarding a Windows Porting

To port I-GET UIMS to run under Windows, a toolkit that implements the fundamental interaction elements is required. These elements can be separated in the following three categories:
?? **Interaction objects**, such as Buttons, Frame Windows, Edit Fields etc, having a variety of typed attributes as well as an arbitrary number of methods and callbacks.

?? **Input events**, which are notifications, from the interaction objects, that may be received during the interaction, from the toolkit, and may have an arbitrary number of event parameters.

?? **Output events**, which are the functionality offered by the toolkit to the developer. Output events have an arbitrary number of typed parameters that contain data to be sent to the toolkit, as well as an arbitrary number of returned parameters that contain data to de returned from the toolkit after the output event processing is done.

The integration of the *Toolkit Server* includes two main sub-tasks, (i) the toolkit interface specification and (ii) toolkit server development. These sub-tasks are described in the next two paragraphs.

1.4.1 Windows programming interface specification

The *Toolkit* (in our case Windows) *Interface Specification* is the definition of the appropriate lexical interaction objects classes, input events and output events, in the *I-GET* language.

The objects classes define the objects with their type attributes (like “x”, “width”, “text”, “font”, etc), and methods (like “Pressed”, “Selected”, “Edited”, etc). Methods typically characterize what the user is able to do with an interaction object, such as pressing a button or selecting an item from a list box.
Chapter 1: Introduction

Figure 5: Code excerpts from the Windows Toolkit Interface Specification.

The input events are notifications that the user receives, from the toolkit server, during the interaction. They have a number of typed parameters that contain event-specific data, and asynchronously occur in the context of objects. In Windows we have two type of input events: (a) the device oriented input (like key presses, mouse moves,
etc), and (b) the notification for window management events (like window repainting, window destruction, window size changed, etc).

The output events are functions with an arbitrary number of output parameters that are necessary for the toolkit to perform the output event, and an arbitrary number of input parameters (values possibly returned from the toolkit to the caller after an event has been served). As output events may concerned the functions for managing the interaction objects (like “MoveWindow”, “InsertString”, “SetButtonBitmap”, etc), the functions for modifying / retrieving objects’ attributes (like “SetX”, “GetX”, etc), and the functions for drawing (like “AllocateGraphics”, “DrawLine”, “PolyBezier”, “Pie”, etc).

The Toolkit Interface Specification needs to be compiled with the I-GET compiler and as a result the compiler will produce two groups of code-generated files. The first group will need to be linked with the Dialogue Control modules that we will develop, when elements from the integrated toolkit are used within the Dialogue Control implementation. The second one includes: (a) a couple of files that contain data types, various constants and data un-marshalling functions (since the toolkit server will be an independent system process, data are communicated packaged into buffers); and (b) a “template” structure of the toolkit server, in which we have to implement some specific functions.

In figure 5 we provide some code excerpts of the toolkit interface used to build the Advanced Windows Toolkit Server.
1.4.2 Toolkit server development

The development of the Toolkit Server concerns two categories of implementation tasks.

The first is to handle all requests made by the dialogue control for instantiating objects, modifying attributes and performing output events. Apart from these, the dialogue control will also report / cancel interests on input events, so that optimized notification management can be accomplished (i.e. no need to send continuously detected input events to objects, unless interest is explicitly stated by those objects); event interest is reported at the object instance level. This implementation task requires the following:

?? The implementation of the function \texttt{void TS\_ServeInstantiateObject}
\begin{verbatim}
(int obj, char* objclass, int parent)
\end{verbatim}, which is automatically called at run-time to handle requests for the instantiation of interaction objects.

?? The implementation of the function \texttt{int TS\_CommonAttributeModify(int obj, char* attr, void* data)}. This function is needed to handle attributes common to all interaction object classes (such as x, y, width and height), and to save code (i.e. no need to replicate handling of common attributes for each object class), as well as computation time.

?? The implementation of all the various attributes modification handlers. These functions follow the prototype convention: \texttt{void TS\_<object class>_AttributeModify(int obj, char* attr, void* data)}. For example, all attribute modification requests for the instances of the \texttt{EditControl} object class should be implemented within the function \texttt{void
TS_EditControl_AttributeModify(int obj, char* attr, void* data).

?? The implementation of all the various output event handlers. Those functions follow the prototype convention: TS_<output event name>_OutputEvent (OUT_<output event name>&). For instance, the InsertString output event is handled by the function void TS_InsertString_OutputEvent (OUT_InsertString&). The OUT_InsertString is a structure type, containing all output event parameters.

?? The implementation of two functions for keeping, locally at the Toolkit Server, information on event interest, those functions are: the void TS_ServeReportEventInterest(int obj, char* evclass), and the void TS_ServeCancelEventInterest(int obj, char* evclass). The first function is being called when an object instance reports its interest for receiving events, while the second is being called when an object instance cancels its interest for receiving events.

?? And the implementation of the function int TS_ObjectClass(int obj), which for any object instance int obj it returns an integer indicating the class of the supplied object instance. For each object class <class>, the returned integer value should be one of the macro constants OBJECT_No_<class>. For example, for instances of the class CheckBox, the value OBJECT_No_CheckBox should be returned.

The second implementation task is to ensure that all important notifications will be sent back to the dialogue control. Such notifications concern input events, method notifications for object instances, as well as attribute modifications due to user
interaction (in order to ensure that there are always the correct values of the object attributes at the Dialogue Control component side). For instance, when the user moves the mouse cursor within an object, or if the value of a text-field is changed, an appropriate event or attribute notification needs to be sent, respectively. Additionally, if the user selects an option from a menu, a corresponding method notification should be sent (notice here the conceptual differences between events and methods). To accomplish this behavior, we carried out the following programming tasks:

?? Within the `TS_ServeInstantiateObject()`, we registered the necessary event handlers, which, for each event handling, will simply post a notification to the Dialogue Control component.

?? We also registered the appropriate methods / callbacks, which will notify the Dialogue Control component that the particular method, for the specific object instance, has been triggered.

?? Similarly, within the same function, we registered monitor functions, which will grasp the change of those object attributes that may change due to run-time interaction, and post an attribute notification message to the Dialogue Control component.

1.4.3 Extensible Interactive programming environment

In addition to the integration of the Windows Toolkit Server, we implemented an extensible interactive programming environment. This environment is an editor with some extra features such as colour syntax highlighting, for the `I-GET` language keywords, and linkage to the `I-GET` compiler. This editor also offers the functionality
for finding and replacing text either in the whole edited file or within a selection. Although the editor was first built, to support the I-GET language, and to be used for the I-GET tool, it is designed and implemented in a way that it can be adapted and used by other programming languages and tools. The editor toolkit is composed of the following components:

- **Editor Manager**, which is the basic component of the editor toolkit, and within it all the processing takes place.

- **Undo / Redo stack manager**. This component is responsible for the undo / redo operations.

- **Clipboard Manager**. This component handles the cut, copy and paste operations within the text.

- **Syntax Highlighting Manager**, which is responsible for the highlighting of the text’s keywords.

- **Find / Replace / Goto-Line Manager**. This component handles the find / replace operations within the text, as well as, the goto-line requests.

- **Project Manager**. The project manager handles the insertion / deletion of the files within a project.

- **Compiler Manager**. This component makes the linkage between the editor manager and the compiler.

- **Tools Manager**. This component makes the linkage between the editor manager and external tools.
? Editor Application. The editor application is the editor tool interface where the interaction with the user takes place.

Figure 6: Components of the Editor Toolkit.

Figure 6 shows the connection between these components. The architecture and the functionality of the editor and the role of each component are described with more details within Chapter 3 – The Interactive Programming Environment.

1.4.4 Large-scale applications tests

The final task we had to accomplish was the built of extensive tests that could show the capabilities of the I-GET UIMS, on the one hand, and on the other, could help us to perform large-scale extensive testing of the Advanced Windows Toolkit Server.

The first test is the Scientific Calculator Application, which can compute all simple operations, as well as, statistic, trigonometric and other scientific operations. In this
test we used hooks and bridges that mix the I-GET language with C / C++, to perform and compute the mathematical operations in the C / C++ language.

The second test is the Graphics Drawing Application, which was built to test the Windows Toolkit imported graphics drawing capabilities.

The application provides the most common capabilities of simple drawing, as well as the use of some drawing tools such as pen, eraser, colour box, flood fill, and tools for drawing geometrical shapes (polygons, rectangles, ellipses). It also gives the ability to load and save images in a specific format (Bitmap file format).

The final test is the Geometrical Constraints Application. This application is a demonstration of the Line and Rectangle objects, and constraints among their attributes.

In Chapter 4 – Application Test Suite, we describe with more details their functionality and the structures used to build them and in the Appendices we provide some code excerpts of the above examples.
Chapter 2: The Advanced Windows Toolkit Server

CHAPTER 2
THE ADVANCED WINDOWS TOOLKIT SERVER

2.1 Architecture

In this section we describe the architectural components that have been used to develop the Advanced Windows Toolkit Server for the I-GET UIMS (see figure 7).

![Advanced Windows Toolkit Server architecture diagram](image)

*Figure 7: Advanced Windows Toolkit Server architecture.*
Chapter 2: The Advanced Windows Toolkit Server

The Windows Server Main is the central component of the Advanced Windows Toolkit server. The initialisation of the toolkit server and the establishing of the communication link with the Dialog Control are the primarily concerns of that component.

Within the Windows Library component are performed the necessary communication checks and the dispatching of work. Additionally, there is a mapping layer (Windows type-mapping layer) that maps Windows to Dialogue Control objects such as interactive objects, bitmaps and device contexts, named DC, (Windows data structures containing information about the drawing attributes of a display device).

The Dispatcher layer consists of four components: a) objects instantiation, b) output event, c) attributes modification and d) input event interest dispatcher. The objects instantiation dispatcher holds the instantiation functions for all interactive objects in a dispatch table with the appropriate ordering. Each time a request for an object instantiation occurs, the component calls the appropriate instantiation function. Next is the output event dispatcher that calls the appropriate handler function for each output event category. Those functions follow the prototype convention:

```c
void TS_<output event name>_OutputEvent(GWOUT_<output event name>&);
```

The GWOUT_<output event name> contains all output event parameters concerning the specific output event category.

The attributes modification dispatcher calls the appropriate handler function each time a request for modifying attributes that are common to all interactive objects occurs. The handler function is the:

```c
bool TS_CommonAttributeModify(int obj, char *attr, void *data);
```
In the above handler the ‘obj’ argument is the object’s Dialogue Control integer identifier, the ‘attr’ is the attribute’s name and the ‘data’ is the buffer containing the new attribute’s value.

The event interest dispatcher manages requests for reporting or cancelling event interests. Within that component two dispatch tables are implemented; one concerning the reporting while the other concerns the cancelling of input events. Each dispatch table contains functions that follow the prototype convention:

\[
\text{void } \text{<category prefix>}\text{Event<event name> (ObjectData* data)}
\]

In the above function prototype the \text{<category prefix>} could be either Report, for reporting event interest functions, or Cancel, for cancelling event functions, and the \text{<event name>} is the input event name (i.e. for the input event MouseMove the report function is the \text{void ReportEventMouseMove()}). The ‘data’ argument is a structure that encompasses the necessary information concerning the object, for which the event would be reported or cancelled.

The last component of the Advanced Windows Toolkit Server architecture is the Notifier component. That component includes the Attribute modification, Input event and Method notifiers. The Attribute modification notifier handles all requests originated from the Dialogue Control concerning modification of object-specific attributes. For each object class the notifier component calls a function that follows the prototype convention:

\[
\text{void } \text{TS_<object class>_AttributeModify(}
\text{   int obj,}
\text{   char* attr,}
\text{   void* data);}
\]

The arguments of these functions are similar to the parameters of the function \text{TS_CommonAttributeModify()}, and the \text{<object class>} is the class-name of the
object for which the function has been called (i.e. for the Button object the function would be `TS_Button_AttributeModify()`).

The Input event notifier component calls the appropriate notifier function when an input event is detected during the interaction. After finished processing the input event, a call to the library function `TS_DoInputEvent()` should be made to post the packaged input event to the Dialogue Control component.

Finally, the Method notifiers are implemented within interaction objects classes for each method category. When during the interaction a method is triggered for an object instance locally at the toolkit server, the corresponding function will be called, thus, posting the appropriate method notification to the Dialogue Control via the library function `TS_DoMethodCall()`.

```c
#ifndef DeskTop_IGET_HEADER
#define DeskTop_IGET_HEADER

struct GWOUT_ColumnWidth {  
  int theList;  
  int column;  
  int width;  
};

struct GWEVENT_Moved {  
  int x;  
  int y;  
};

struct GWColor {  
  int red;  
  int green;  
  int blue;  
};

// Constants for lexical element classes.  
// #define OUTEVENT_ColumnWidth "ColumnWith"  
#define OUTEVENT_No_ColumnWidth 1
extern void TS_ColumnWidth_OutputEvent(GWOUT_ColumnWidth&);
...  

#define OBJECT_ListBox "ListBox"  
#define OBJECT_No_ListBox 1
extern void TS_ListBox_AttributeModify(int, char*, void*);
...

//External Reverse function prototypes.  
// extern void DeskTop_Reverse(GWOUT_ColumnWidth (*X), void* data);
extern void DeskTop_Reverse(GWFont (*X), void* data);
extern void DeskTop_Reverse(char* (*(*X)), void* data);
extern void DeskTop_Reverse(int (*(*X)), void* data);

#define INEVENT_ShowWindow "ShowWindow"
...
#endif
```

*Figure 8: Code excerpts from wintstr.hh*
2.2 Automatically Generated Source Code

When the Windows Toolkit Server Interface specification is compiled, as we mentioned before, two sets of files are generated. The first set is the one to be compiled and linked with each Dialogue Control implementation. The second set is to be used for toolkit server development, and encapsulates two C++ files: the header file \texttt{wintstr.hh} and the implementation file \texttt{wintstr.cpp}.

In the generated header file \texttt{wintstr.hh} the C++ version for the original types declared in the Windows Toolkit Interface Specification are included. This file also includes data structure types for output and input events, reflecting the various typed parameters, macro constants for event class names and output event names, as well as macro constants with both string and numeric values for object class names.

Additionally, it includes the definition of the data translation functions for all data types that the Dialog Control may at run-time post to the \textit{toolkit server}. Those functions are named Reverse functions un-marshal data values from \texttt{void*} buffers (received from the Dialogue Control). Their implementation is generated automatically by the \textit{I-GET} compiler and is included in the file \texttt{wintstr.cpp}.

Finally the \texttt{wintstr.hh} header file includes all the function prototypes corresponding to output events and object attributes modification requests; that should be implemented as part of the toolkit server. Selected code excerpts from the \texttt{wintstr.hh} generated header file are shown in figure 8.

2.3 Implementation

In this section we describe the implementation of the Advanced Windows Toolkit Server. The development of the \textit{toolkit server} has been done using the Microsoft
Visual Studio Environment. The application we have created was generated by the MFC AppWizard, and uses the MFC library.

2.3.1 Toolkit initialization and Main Loop

The first thing that we have to do is to supply an initialization function for the toolkit server (i.e. `InitInstance()`). In this function we have to explicitly call the function `TS_InitializeCommunication()`, so that the run-time communication link with the Dialogue Control Component can be established. This is a generic approach of the initialization for the toolkit server.

Firstly `InitInstance()` passes the command line arguments to the toolkit server I-GET kernel by calling the function:

```c
TS_PassCommandArgs(int argc, char** argv),
```

where the `int argc` is the total arguments and `char** argv` is the command line arguments, followed by a call to `TS_InitializeCommunication()` (see figure 9).

```
BOOL CwinServerApp::InitInstance() {
    int argc;
    char* argsCopy;
    char* argv=ParseCommandArgs(
        &argc,
        argsCopy=strdup(GetCommandLine()));
    free(argsCopy);
    TS_PassCommandArgs(argc, argv);
    TS_InitializeCommunication();
    .  .  .
}
```

*Figure 9: InitInstance () function.*
The next thing is to supply a main-loop. The program style of a main-loop may vary in different toolkits. It may be a well-defined function that is called at the end of initialization, or it can be an explicit message loop where messages should be dispatched to the appropriate handler code while it may also be implemented as a low-level event management inner loop.

The Advanced Windows Toolkit Server, being an MFC application, also requires its MFC-specific internal main-loop. Hence, in practice, we have two main loops, one required by MFC, and one required by the I-GET toolkit kernel library. In implementation, these two loops are interleaved as follows: In the OnIdle() MFC function; regularly called by the Windows library at run-time, we incorporate a call to the TS_ServeDialogueControlTransaction() function (see figure 10), which performs all incoming communication checks and dispatching work.

```c
BOOL CwinServerApp::OnIdle(LONG lTime) {
    TS_ServeDialogueControlTransaction();
    return 1;
}
```

*Figure 10: The OnIdle (LONG lTime) function.*

The OnIdle() function guarantees that the TS_ServeDialogueControlTransaction() will be called when no message processing is being done by the toolkit server application.

2.3.2 Instantiating Objects

The TS_ServeInstantiateObject() is the function that dispatches object instantiation requests (figure 11). In this function we create a toolkit object instance,
and we store in the local ObjectMapping table the Dialogue Control object identifier together with the newly created instance.

```c
void TS_ServeInstantiateObject(int obj, char* classname, int parent){
    CObject* newObject;
    ObjectData* parentData = NULL;
    int objclass;

    //Identify the macro for this class.
    //
    objclass = TS_ObjectMacro(classname);
    if (objclass == -1) {  
        TS_Message("Not served class. 
");
        return;
    }

    //Toolkit specific checks here.
    //
    if (objclass != OBJECT_No_FrameWindow && !AppWindowOpened) 
        TS_Error("Need to open first a FrameWindow.
");
    if (parent != -1)
        parentData=MappingTable.MapObject(parent);

    (*CreateObject[objclass])(newObject, parentData, obj);
    UpdateHierarchy(parentData, obj);
}
```

*Figure 11: The TS_ServeInstantiateObject () function.*

The parameters passed to that function are the object’s Dialogue Control integer identifier, its class name and the parent object Dialogue Control integer identifier, if any. The I-GET toolkit kernel calls explicitly this function to create an instance of an object of the class specified by the `char* classname` parameter.

In this function we firstly retrieve the object’s class identifier from the class name using the `TS_ObjectMacro()` function. If the retrieved identifier is invalid, we report the error and return. In the case that the object to be instantiated is not a FrameWindow and we haven’t instantiated yet a FrameWindow, we report an error message that notifies the programmer to create a FrameWindow before instantiating
other objects. If the parent parameter is not –1, it means that we have a parent object for the one to be instantiated. To retrieve the parent object a search is performed in the MappingTable that holds all instantiated objects. If the parent object identifier is –1, it means we have no parent.

```c
typedef void (*InstantiateObject)(CObject ob, ObjectData pData, int obj);
static void CreateObjectNone(CObject* ob, ObjectData* pData, int obj);
static void CreateListBox(CObject* ob, ObjectData* pData, int obj); //ListBox Instantiation func.
static void CreateExCheckBox(CObject* ob, ObjectData* pData, int obj); //ExCheckBox.
InstantiateObject CreateObject[] = {
    CreateObjectNone, // Never called.
    CreateListBox,
    CreateButton,
    CreateRadioButton,
    CreateGroupBox,
    CreateEditField,
    CreateLabel,
    CreateComboBox,
    CreateSlider,
    CreateGauge,
    CreateControlBar,
    CreateGadget,
    CreateFrameWindow,
    CreateExCheckBox,
};
```

**Figure 12: Instantiation functions and corresponding dispatch table.**

The object’s instantiation function is actually a dispatch table that holds the instantiation functions for all objects (see figure 12). To call the appropriate instantiation function we ordered the functions in the dispatch table based on the objects class identifier. We retrieve this identifier by calling the `TS_ObjectMacro(classname)` function, where the `classname` is the object’s class.

The instantiation function for the Button object is shown in figure 13. Firstly, a check is performed to determine if a parent object has been supplied. If there is not a parent object then we report an error message and return. Then we insert a new entry to the MappingTable for the new object instance and we create the Button using the Button’s class initialization function.
2.3.3 Handling Attribute Modification

Attribute modification handling concerns: (a) attributes that are common to all interaction object classes; and (b) all the rest of attributes.

The handling of common attributes is done within the function `TS_CommonAttributeModify(int obj, char* attr, void* data)` (see figure 14). To modify a common attribute we need object specific information so we use the MappingTable to get it. Based on attribute name `attr`, we decide whether it is a common attribute (i.e. calling `TS_CommonAttributeModify()`), or not (calling the appropriately class specific attribute modification handler function).

In the `TS_CommonAttributeModify()` function we call the appropriate functions to handle the details of assigning the new attribute values to the respective object instances. We will discuss one of these functions (shown in figure 15) with the

```cpp
//-----------------------------
// Create Button object.
void CreateObjectButton(CObject* ob, ObjectData* pData, int obj){
    if (!pData) {
        TS_Error("No parent passed.\n");
        return;
    }
    ObjectData* newEntry = MappingTable.RegisterObject(
        ob, pObj,
PData->tsobj,
OBJECT_No_Button,
PData->parent_dobj);

    ObjectData* odata = MappingTable.MapObject(pobj);
    if (odata) {
        odata->tsobj = new Button(pobj);
        ((Button*)odata->tsobj)->Create("",
BS_PUSHBUTTON|BS_OWNERDRAW |WS_CHILD,
CRect(0,0,0,0),
((CWnd*)pData->tsobj),
pobj);  
    }  
}
```

Figure 13: The instantiation function for the Button object.
signature static void Set_Enable(CObject* ob, void* data); all the rest follows similar implementation style.

```c
// Common attributes modification handler.
// bool TS_CommonAttributeModify(int obj, char* attr, void* data) {
//    ObjectData* odata=MappingTable.MapObject(obj);
//    if (!odata) {
//        TS_Error("Trying to modify attribute of "
//                  "non-registered object.\n");
//        return false;
//    }
//    else if (!strcmp(attr, X))
//        Set_x(odata, data);
//    else if (!strcmp(attr, Y))
//        Set_y(odata, data);
//    else if (!strcmp(attr, WIDTH))
//        Set_width(odata, data);
//    else if (!strcmp(attr, HEIGHT))
//        Set_height(odata, data);
//    else if (!strcmp(attr, FONT))
//        Set_font(odata, data);
//    else if (!strcmp(attr, ENABLE))
//        Set_Enable(odata->tsobj, data);
//    else if (!strcmp(attr, ISVISIBLE))
//        Set_Visible(odata, data);
//    else
//        return false;
//    return true;
//}
```

`Figure 14: The TS_CommonAttributeModify() function.`

In this function the `void* data` argument concerns the received data buffer, not yet translated to a typed value. To do so we use the automatically produced `Reverse` functions residing in `wintstr.cpp`. In this case, the overloaded function `Reverse (int*, void*)` is used, taking the `void*` buffer and translating it to an integer value. Subsequently, after retrieving the attribute’s value we call the Windows object’s `EnableWindow(BOOL bEnable)` function, which enables or disables the object based on a conversion of the integer attribute to boolean value.
2.3.4 Handling Output Events

Output events are handled in the implementation of the various output-event related functions, following the automatically generated function prototype definitions. There are two steps in handling an output event. Firstly, we perform the necessary toolkit specific processing and secondly, we return the result parameters back to the caller (Dialogue Control). An example of an output event function, in which no results are returned to the Dialogue Control component, is the function `void TS_ClearList_OutputEvent (GWOUT_ClearList& p)`. It should be noted that all functions follow the signature pattern `TS_<output event name>_OutputEvent (GWOUT_<output event name>&)`, in this example the output event name is the `ClearList`. The `GWOUT_ClearList` struct carries the output event parameters, encompassing a single field, i.e. the `int theList`, which corresponds to the Dialogue Control object instance (either ListBox or ComboBox) unique identifier.

```c
void TS_ClearList_OutputEvent(GWOUT_ClearList& p) {
    ObjectData* odata=MappingTable.MapObject(p.theList);
    if (!odata)
        TS_Message("Object for 'ClearList' not found.");
    else if (odata->objclass==OBJECT_No_ListBox)
        ((CListbox*)(odata->tsobj))->ResetContent();
    else if (odata->objclass==OBJECT_No_ComboBox)
        ((CCombobox*)(odata->tsobj))->ResetContent();
    else
        TS_Message("Object for 'ClearList' must be "
                   "ListBox or ComboBox.");
}
```

Figure 15: The attribute modification function Set_Enable ().

```
// Set_Enable() Attribute modifier handler.
static void Set_Enable(CObject* ob, void* data) {
    int enable;
    Reverse(&enable, data);
    ((CWnd*)ob)->EnableWindow(enable);
}
```

Figure 16: The TS_ClearList_OutputEvent (GWOUT_ClearList& p) function.
In figure 16, the implementation of the `TS_ClearList_OutputEvent()` output event function is shown. This output event maybe called either for a ListBox or for a ComboBox. Firstly, we extract the run-time object information from the `/MappingTable` using the `p.theList` object identifier; if the result of this call is null, we report the appropriate error message and return. Else, we test the `objclass` (object class) member, of the `ObjectData` structure, against a ListBox or a ComboBox class identifier. Based on the result of the test, we call the appropriate class specific function (to clear the ListBox or the ComboBox). If the object class identifier supplied does not correspond to a ListBox or a ComboBox, an error message is displayed.

Next, we will study an example of an output event, in which we have to return result parameters to the Dialogue Control component. The `void TS_AllocateGraphics_OutputEvent(GWOUT_AllocateGraphics& p)` is one such output event function.

```c
void TS_AllocateGraphics_OutputEvent(GWOUT_AllocateGraphics& p)
{
    ObjectData* odata = MappingTable.MapObject(p.obj);
    if (!odata) {
        TS_Message("Object not found for AllocateGraphics");
        return;
    }
    CDC* dc = ((CWnd*)odata->tsobj)->GetDC();
    DCHandler.RegisterDC(DCSerial, dc, odata->tsobj);
    int size;
    void* senddata = Transform(DCSerial, &size);
    DCSerial++;
    TS_DoReturnedParams(senddata, size);
    ((CWnd*)odata->tsobj)->ReleaseDC(dc);
    CLEAR_BUFFER(senddata);
}
```

*Figure 17: The TS_AllocateGraphics_OutputEvent (GWOUT_AllocateGraphics& p) function.*

The output event parameters for this function (see figure 17), are encompassed in the `struct GWOUT_AllocateGraphics`, being the `int obj` member, which is the object
identifier from which we will allocate the graphics resources. Firstly, we retrieve the
run-time object data from the MappingTable using the p.obj object identifier; if no
such data are found we report the error and return. Then we get the device context
from the object and register it in the DCTable structure. After registering the device
context we should send back the Dialogue Control device context integer identifier,
through which further references by Dialogue Control will be made possible.

To do that, firstly, we have to transform the device context identifier to a void*
buffer. Because the returned parameter is of a built-in type (in this case int), we may
use the built-in overloaded Transform function. If the returned parameters are not of a
built-in type we have to explicitly implement these functions.

The next step is to perform DCSerial++, which increases the serial number, to ensure
that unique identifiers will be assigned to the device context identifiers. Subsequently,
we use the library function call: TS_DoReturnedParms(senddata, size); to send
the encoded buffer to the Dialogue Control, we release the allocated graphics device
context, and, finally, we free the buffer with a call to the CLEAR_BUFFER
(senddata) macro.

2.3.5 Event Interest Handlers

Input event notifiers are registered / unregistered within the event reporting /
canceling interest functions. Initially, when interaction objects are instantiated within
the TS_ServeInstantiateObject() function, no event notifier is registered. When
interest is reported for an event category, the corresponding interest counter is
increased. If it becomes 1, we know that we have to register the corresponding event
notifier too; else (i.e. above 1), no further action is taken. Similarly, if event interest is
canceled, the interest counter is decreased. If it becomes 0, the corresponding event
notifier is removed; else (above 0), no further action is taken. The implementation of
the `TS_ServeReportEventInterest()` and `TS_ServeCancelEventInterest()`
event interest handlers is shown in figure 18.

In both implemented functions, firstly, we obtain the object for which the event
interest will be reported or cancelled. Then we convert the event class name to the
event integer identifier using the function `TS_InEventMacro(char* evclass)` and
we increase by one the event integer identifier. We do that, because the input event
handlers, for each event class, are placed in a dispatch table in which the first element,
at zero index, is a dummy function and the event integer identifier, which is returned
by the `TS_InEventMacro()` function, is zero-based. After retrieving the event
identifier, we check if the obtained object data, for the object that the event interest is
reported or cancelled is a valid one; if not, the call returns. Else, we call the

```c
void TS_ServeCancelEventInterest(int obj, char* evclass) {
    ObjectData* odata=MappingTable.MapObject(obj);
    int event=TS_InEventMacro(evclass);
    event++;
    if (!odata)
        return;
    (*SCancelEvent[event])(odata);
}

void TS_ServeReportEventInterest(int obj, char* evclass) {
    ObjectData* odata=MappingTable.MapObject(obj);
    int event=TS_InEventMacro(evclass);
    if (!odata) {
        TS_Error("Object in reporting event interest not found");
        return;
    }
    (*SReportEvent[event])(odata);
}
```

*Figure 18: The TS_ServeCancelEventInterest() and TS_ServeReportEventInterest() functions.*

appropriate event interest handler with the call: (a)

`(*SReportEvent[event])(odata)` to report an event interest, and (b)
(*SCancelEvent[event]*)*(odata)* to cancel an event interest. In both calls the *event* parameter is the event identifier and the *odata* parameter is the retrieved object data structure.

```c
// Dispatch table for report event interest handlers.
// ReportEventKillFocus, ReportEventDestroyed};
// Dispatch table for cancel event interest handlers.
// ServeCancelEvent SCancelEvent[]={CancelEventNone, CancelEventPaint,
// CancelEventKillFocus, CancelEventDestroyed};

// Cancel event interest handler for paint event.
// void CancelEventPaint(ObjectData* ob) {
//     if (!ob->PaintInterest)
//         return;
//     if (!--ob->PaintInterest)
//         ob->PaintInterestFlag=0; //Remove event handler.
// }

// Report event interest handler for size event.
// void ReportEventSize(ObjectData* ob) {
//     if (!ob->SizeInterest++)
//         ob->SizeInterestFlag=1; //Add event handler.
// }
```

*Figure 19: The Event interest handlers dispatch tables and the handlers CancelEventPaint(), for canceling event interest for the Paint event and ReportEventSize(), for reporting event interest for the Size event.*

Now we will study specific examples for event interest handlers. In figure 19, code excerpts from the relevant dispatch tables, and two such handlers, one for canceling and one for reporting event interest, are shown. The first handler is the *CancelEventPaint(ObjectData ob)*. In this function, firstly, we check if we have event interest for the Paint event. In a positive case, we decrease the *PaintInterest* counter and check again if it is 0. If it does, we remove the event handler too, by setting the *PaintInterestFlag* to 0. In the event interest report handler (in our example, *ReportEventSize (ObjectData ob)*), we firstly check if the *SizeInterest* counter, for the Size event, is equal to zero in which case we register
the event handler function and then we increase that counter. If it was zero then we add the event handler for that event, otherwise we return.

### 2.3.6 Input Event Notifiers

Input event notifiers are called when an input event is detected during run-time interaction. Input event notifiers are implemented within each object class for all the various event classes. In each such notifier, we firstly check if there is an event interest reported, in which case we make a call to the library function \textit{TS_DoInputEvent()}, to post the packaged input event to the Dialogue Control component. If no interest has been reported, the detected event is just ignored.

```cpp
// Event interest notifier for object Button,
// event LeftButtonPressed.
void Button::OnLButtonDown(UINT nFlags, CPoint point) {
    .  .  .
    ObjectData* odata=MappingTable.MapObject(m_obj);
    if (odata&&odata ->LeftButtonPressedInterestFlag==1) {
        GWEVENT_LeftButtonPressed gw_LButtondown;
        void* data;
        int size;
        gw_LButtondown.x = point.x;
        gw_LButtondown.y = point.y;
        gw_LButtondown.control= nFlags & MK_CONTROL;
        gw_LButtondown.lButton= nFlags & MK_LBUTTON;
        gw_LButtondown.mButton= nFlags & MK_MBUTTON;
        gw_LButtondown.rButton= nFlags & MK_RBUTTON;
        gw_LButtondown.shift = nFlags & MK_SHIFT;
        data = Transform(gw_LButtondown,&size);
        TS_DoInputEvent(m_obj,INEVENT_LeftButtonPressed, data, size);
        CLEAR_BUFFER(data);
    }
    Cbutton::OnLButtonDown(nFlags, point);
}
```

\textbf{Figure 20: Input event notifier for the event LeftButtonPressed for the Button object.}

An example of an input event notifier is shown in figure 20. The function \texttt{BOOL Button::OnLButtonDown (UINT nFlags, CPoint point)} is the event handler for the Left Button Pressed event. In this function we check if there is event interest for
the LeftButtonPressed event for the calling object (i.e. the object in which the event occurs). If there is interest, we post to the Dialogue Control component the event. To do so, we create firstly a `void*` buffer filled with the appropriate data from the LeftButtonPressed input event. That buffer is created using the Transform function `Transform(GWEVENT_LeftButtonPressed, int* size)`, that has been specifically implemented for the LeftButtonPressed event structure.

The `GWEVENT_LeftButtonPressed` structure is a structure containing information about the LeftButtonPressed event: the mouse cursor point at the moment the event occurred, and the flags that indicate whether the shift or/and the control key is pressed too.

```c
// Attribute Modification notifier
// for the button object.
//
void TS_Button_AttributeModify(int obj, char* attr, void* data) {
    ObjectData* odata=MappingTable.MapObject(obj);
    if (!odata) {
        TS_Error("Trying to modify attribute of non-registered object.");
        return;
    } else if (!strcmp(attr,LABEL))
        Set_label(odata->tsobj,data);
    else if (!strcmp(attr,BKCOLOR))
        Set_bkColor(odata->tsobj,data);
    else if (!strcmp(attr,TEXTCOLOR))
        Set_textColor(odata->tsobj,data);
    ... else if (!strcmp(attr, HASBITMAP))
        Set_ButtonHasBitmap(odata,data);
    else if (!strcmp(attr, BITMAPWIDTH))
        Set_ButtonBitmapWidth(odata,data);
    else if (!strcmp(attr, BITMAPHEIGHT))
        Set_ButtonBitmapHeight(odata,data);
}
```

Figure 21: Attribute Modification notifier for the Button object.

After transforming the input event data to the `void*` buffer, we make the call to the library function `TS_DoInputEvent()`, to post that data to the Dialogue Control component. Then, we call the `CLEAR_BUFFER()` to free the memory allocated for
the `void*` buffer, and finally we call the `CButton::OnLButton()` function which is the MFC default handler for the `LeftButtonDown` event.

### 2.3.7 Attribute Modification Handlers

These functions concern handling of attributes, which are not common to all object classes. In each such function, we firstly check if the object, for which the attribute modification request is made, is valid. Then, based on attribute name, we call the appropriate handler function.

In figure 21, the implementation of the `TS_Button_AttributeModify()` function, for the button object class, is shown. As it can be observed, we make calls to some “helper” functions, which perform the real modification work. One such function handling the modification of the text colour attribute, for the Button object, is the `Set_textColour()` function (figure 22).

```c
// Button attribute modification handler // for the text color attribute. 
static void Set_textColor(CObject* ob, void* data) {
    GWColor color;
    DeskTop_Reverse(&color, data);
    ((Button*ob)->SetTextColour(RGB(color.red, color.green, color.blue));
    ((Button*ob)->RedrawWindow();
}
```

*Figure 22: Attribute Modification handler for the text colour attribute, for the Button object.*

In this function, we extract the RGB colour value using the `Reverse` function `DeskTop_Reverse` (GWColour* colour, void* data). The `GWColour` colour structure contains values for red (colour.red), green (colour.green) and blue (colour.blue). Then we call the `SetTextColour()` Button class member function to actually set the text colour for the button object instance. Finally, we call the `RedrawWindow()` member function, so that the change may take effect.
2.3.8 Method Notifiers

In contrast to event notifications, where, interest reporting is explicitly required, method notification requires no interest reporting.

When instantiating an object, all method notifiers, applicable to its respective class, are automatically registered. As a result, if during interaction a method is triggered, its notification callback will be called, to post the corresponding notification to the Dialogue Control component.

Figure 23 provides the method notifier for the ‘Edited’ method of the ComboBox object. Firstly, we extract the text of the ComboBox label or edit control. Then, we transform it to a void* buffer, using the function Transform(char*, int*) and we perform the library function call TS_DoModifiedAttribute() to send the new text value to the Dialogue Control component (the EDITTXT is a macro constant with the value ‘labelOrUserText’). Finally we post the method notification to the Dialogue Control component via the library function TS_DoMethodCall() (the COMBOBOX_EDITED is a macro constant with the value ‘Edited’). It should be noted that when a user action causes a method to be triggered, and also, changes an

```c
// Method notifier for the Edited method of the ComboBox object.
void ComboBox::OnEditchange()
{
    CString txt;
    GetWindowText(txt);
    char *str = _strdup(txt);
    int size;
    void* senddata = Transform(str,&size);
    TS_DoModifiedAttribute(m_obj,EDITTXT,senddata,size);
    TS_DoMethodCall(m_obj,COMBOBOX_EDITED);
    CLEAR_BUFFER(senddata);
    free(str);
}
```

Figure 23: Method notifier for the “Edited” method of the ComboBox object.
object attribute, we firstly post the attribute modification notification followed by the method notification.

2.4 Maintenance and Extensibility

The Advanced Windows Toolkit Server is designed in a way that it can be easily extended. New components and new features (always under the Windows Programming environment) can be added to extend the capabilities of the Advanced Windows Toolkit Server.

New objects and/or output events can be added, by placing the appropriate code segments in the Windows Toolkit Interface specification. Then the interface specification should be recompiled, with the I-GET compiler, to generate the files needed by the Advanced Windows Toolkit Server to support the new objects. At the toolkit server side, must be implemented the appropriate object initialisation functions, as well as the attribute modification handlers and notifiers. In addition, within the new objects’ classes both the input event and method notifiers should be supplied. If there are any new output events for the newly inserted objects, they have to be implemented too.

The implementation of the objects initialisation functions should be placed within the Windows Toolkit Server Main component and added to the initialisation functions dispatch table. The attribute modification handlers and notifiers should be placed within the Attributes Modification Handlers component. If the new objects have any of the common attributes, these attributes are handled by the
TS_CommonAttributeModify() function. The attribute modification notifiers for the new objects should follow the prototype:

void TS_<object class >_AttributeModify ()

The input event and method notifiers should be implemented within the classes of the new objects. All notifiers for the applicable input events should be implemented as well as the methods defined within the Windows Toolkit Server Interface specification.

Finally, if there are any new defined output events, they have to be implemented too. The implementation of these functions should take place within the Output Events Handler component.
CHAPTER 3

THE INTERACTIVE PROGRAMMING ENVIRONMENT

3.1 Extensible Component-based Architecture

In this chapter we will discuss the development of the Interactive Programming Environment, which is realized as an editor with various capabilities. Firstly, it has been designed to support other languages too, apart from the I-GET language. Additionally, the editor provides syntax highlighting for the language keywords, while it can be linked to any particular compiler.

The components of the editor are illustrated in figure 6. The Editor Library Manager is the main implementation component of the editor and is the only programmatically visible component. The Editor Application reflects the “document / view” architecture of MFC. The ‘view’ class of that application is derived from the EditorView class, while the CMainFrame class of the application is derived from the EditorMainFrame class. Both classes are implemented within the Editor Library Manager component.

All the rest of the implementation components, which serve particular functional goals, reflect the internal object-oriented architecture of the Interactive Programming Environment.
3.1.1 Editor Library Manager

Although, the Editor Library manager encloses all other components, any editor developer can use them separately. The only constrain on using these components is that developers should use the CRichEditCtrl as the edit control of the application.

The EditorView and EditorMainFrame classes are implemented in this component. The first class (EditorView) is necessary for the application to support the Editor Toolkit capabilities such as the syntax highlighting, undo / redo operations and operations concerning the clipboard (cut, copy and paste text among files). The second class (EditorMainFrame) is optional, and the developer may not use it to derive his application’s CMainFrame class. This class provides the Editor with left and bottom panes. The left pane contains a Tree View where the files of a project are displayed, while the second one can be used to display messages such as compile and run-time messages.

Another class that is implemented within the Editor Library component is the EditorMsgHandler class. EditorMsgHandler is a template class so that it can be used within any ‘view’ class, and it supports the syntax highlighting and the undo / redo capabilities of the editor. The key messages WM_KEYDOWN and WM_CHAR are handled within this class by the functions CallOnKeyDownCallback and CallOnCharCallback respectively. In those functions we perform real-time parsing of the typed text, according to the keywords given, and subsequently add the text in the undo / redo stacks appropriately.
3.1.2 Undo / Redo Stack Manager

The Undo / Redo Stack Manager component implements a stack structure that handles the undo and the redo operations. Basically, there are two stacks maintained in this component, one for the undo and the other for the redo operations. Each of those stacks stores `UndoRedoInfo` structure items, which contain all the information regarding the operations to be performed when an undo or redo operation is requested: (a) text typed, (b) text replaced, (c) the start and the end indices of the typed text, and (d) the operation type with which the item inserted into the stack (i.e. typing, deleting, cutting or pasting).

This component is implemented in the `UndoRedoStack` class, which uses the `CRichEditCtrl` control to obtain information about the text to be inserted into the appropriate stack.

3.1.3 Clipboard Manager

The Clipboard Manager supports the cut, copy and paste operations. Especially, the paste operation is the one that needs more attention because the data that are to be paste from the clipboard, to the editor, should be only of the text type. If the user tries to paste another type of data the operation will be ignored.

To perform the copy operation we retrieve the editor’s selected text, and add it to the clipboard. If there is not any selected text we copy the current editor line to the clipboard. Finally, on cut operation the currently selected text is deleted from the editor and added to the clipboard. The cut operation is ignored when there is not any selected text in the editor.
After the completion of a clipboard operation (cut, copy or paste) the text is parsed again for keyword highlighting. The Clipboard Manager is implemented in the EditorClipboard class, which can be used by other developers, with the only restriction to have the edit control derived from the CRichEditCtrl Windows object.

3.1.4 Find / Replace / Goto-Line Manager

The Find / Replace / Goto-Line component encompasses the interface implementation and the internal functionality for find, replace and goto line operations.

![Find Dialog](image)

Figure 24: The Find Dialog.

Upon a find user request, the Find Dialog is displayed (figure 24). In this dialog the user may supply the text to be searched. If there is text selected in the document, it is automatically placed in the text field (else, the text field is initially empty). The Find Dialog provides two check boxes, one for matching the whole word, and another for case-sensitive matching. There are also two radio buttons with which the user can control the direction of the search (i.e. up or down). Finally, the Find Dialog provides two buttons: the Find Next, which finds the next occurrence of the supplied text, and the Cancel button that, closes the search dialog.

The Replace Dialog is shown in figure 25. In this dialog, the user should specify the text to be found and the text that will replace it. As in the Find Dialog, there are the
Match whole word only and the Match case check boxes. Additionally, there are two radio buttons, to specify whether the text replacement should take place within the current selection or within the whole file. There are also four buttons:

- **Find Next**, which searches the document to find the next occurrence of the acquired text.
- **Replace** button, which replaces the found text.
- **Replace All** button, which replaces all occurrences of the given text within the whole file or within the current selection, according to the Replace In radio buttons value.
- **Cancel** button which closes the Replace Dialog.

![Image of Replace Dialog]

Figure 25: The Replace Dialog.

Another feature provided by the Find / Replace / Goto-Line Manager component is the Goto-Line. With this operation the user can jump from one line in the document to another, and it is useful when the user needs to move from the current line, 20 or more lines up or down.
The Goto-Line dialog (figure 26) is a simple dialog with a text field to specify the line number, and two buttons: (a) the *Go To*, which performs the Goto action, and (b) the *Cancel*, which closes the dialog.

This component is implemented in the class `FindReplaceGotoManager` and can be used independently, by any other editor developer that uses the `CRichEditCtrl` as the edit control of the application.

### 3.1.5 Project Manager

The Project Manager component is the one responsible for handling the files of the projects created by the Editor.

That component implements a list data structure of `ProjectInfo` struct items, which hold information about the projects. That information concerns the files included in the project, the project name and the full path of the project file. The functionality provided to the developer, by this component, is separated into two categories: (a) functionality concerning projects (i.e. load, store, add, and remove), and (b) functionality concerning project files (i.e. add and remove).

The Project Manager is a reusable component that can be used by any editor developer, which desires to support project and file management, with no restrictions, since it is completely independent of any Windows or a particular language specific object.
3.1.6 Syntax Highlighting Manager

The Syntax Highlighting Manager component is the one responsible for parsing the typed text and highlighting the language keywords appropriately.

The policy we used for the parsing was to parse the line of the last typed character. We’ve used other methods but they proved to be ineffective especially when needed to parse a large file. One of those methods was to parse the current word. The disadvantage of that method was that we ought to extract the current word from the current line. The cost of that method was more than obvious when we typed in the middle of a word or when we split a word (i.e. by inserting a space character in between), which means that we had to extract two words. With that method the most of the processing time was spend on extracting the words from the text and not on parsing them. Another method we tried to use was the parsing of the whole file, but also that method proved time costly for files larger than 2Ks.

The component reads the keywords for highlighting through a file that must be of a specific format, so that the component can read it successfully. The format of that file is simple; firstly the total number of keywords should be written and then, the keywords separated by a white space character. Besides the keywords highlighting the component provides highlighting for single line comments, numbers, characters enclosed in apostrophes (i.e. ‘c’) and strings enclosed in quotation marks (i.e. “some string”).

Concluding the description of the Syntax Highlighting Manager component it should be noted that this component is based on the \texttt{CRichEditCtrl} Windows Library object,
which means that developers must use CRichEditCtrl, as the edit control of their application, in order to support syntax highlighting.

3.1.7 Compiler Manager

The Compiler Manager component is the one that handles all compilation tasks. This component provides functionality for setting and/or retrieving the compiler executable file, the command line arguments and the flag that indicates whether the compiler output can be redirected to a file.

This component is implemented within the CompilerManager class and serves all compilation requests. When a compile request occurs, the component runs the compiler executable file with the appropriate command line arguments. The Compiler Manager is an independent component and can be used by developers that wish to include compiler support in their editor applications.

3.1.8 Tools Manager

The Tools Manager component handles the linkage to external tools. Although, the component is able to handle more than one tool, the mechanism used is similar to the one described in the previous paragraph concerning the Compiler Manager.

The component maintains a list of ToolInfo struct items, which contains information concerning each tool: the executable file, the command line arguments and the flag that indicates whether the tool’s output can be redirected to a file. Furthermore, it provides editor developers with the necessary functions for managing those tools such as execution functions, functions for retrieving the output corresponding to each tool and functions for iterating through the tools’ list (i.e. to add the tools in a menu).
However, the developers are responsible to provide the users of the editor with the appropriate interface to handle the features supplied by this component (i.e. add, remove, execute, etc).

Finally, there are two functions concerning the initialization and cleanup of the Tools Manager component. The initialization function should be called at startup to perform the necessary initialization of member attributes, while the cleanup function should be called at shutdown of the application to store the appropriate information concerning the tools.

### 3.2 Multi-Document Syntax Highlighting Editor

The *I-GET* Editor is a Multiple Document Interface (MDI) Application, which supports multiple files to be opened at the same time. Although the *I-GET* Editor is a standalone application, it is based on the EditorLibrary component described in the previous paragraph and has all features provided by that component. There are also some features added particularly for the *I-GET* Editor. Those features concern mostly the editor’s interface and appearance, and not so much its language specific functionality.

The editor application can be seen as a composition of three frames (figure 27): a) the left frame, which is a tree view and displays the files of a project, b) the bottom frame, in which the compile or run-time messages are displayed, and c) the right frame, in which the document windows are shown.
Figure 27: The I-GET Editor.

The left frame is actually a TreeView control (implemented by the EditorLibrary component) that displays the files of the particularly loaded project. To have this frame created in the I-GET Editor application, we derived the CMainFrm class from the EditorMainFrame class implemented in the EditorLibrary component. Within that class the left (tree view) and the bottom (messages) frames are created after the creation of the applications main window. The tree view frame constantly communicates with the ProjectManager component of the EditorLibrary. The ProjectManager component holds all needed information for the projects and their files, and informs the frame of any changes that need to be shown in the tree view such as the loading and closing of a project or the addition and removal of a file.

The bottom frame is a tab of two list boxes one for displaying the compile messages and the other for displaying the run-time messages. That frame is created also at the creation of the editor’s main window and is implemented by the EditorLibrary as the left frame (files tree view) does. Each of the two list boxes provides a callback to the
CompilerManager component of the EditorLibrary. Whenever a compile or a run-time message occurs, those callbacks are called to process it and add it to the proper list box.

The right frame provides the functionality of a Windows Multiple Document Interface (MDI) frame window. The MDI frame window is the direct parent of MDI child frame windows (document windows) and holds its own menu bar. When there is an active document window, the MDI frame window’s menu bar is automatically replaced by the MDI child window’s menu.

![Figure 28: Windows arranged as overlapping tiles and minimized windows as icons.](image)

Both of the menu bars provide some features for managing the MDI child windows. When more than one document files are opened, the menus provide options to arrange those in alternative ways: (a) as overlapping windows (Cascade option from Windows menu, figure 28), (b) as horizontal non-overlapping tiles (Tile Horizontally option, figure 29), (c) as vertical non-overlapping tiles (Tile Vertically option, figure 29), and
(d) as minimized document windows, using the Arrange Icon option from the Windows menu (figure 28).

![Figure 29: Windows arranged as horizontal non-overlapping tiles and as vertical non-overlapping tiles.](image)

The implementation of the above features was part of the I-GET Editor and part of the EditorLibrary component. Within the CMainFrm class of the application, which is derived from the EditorMainFrame class of the EditorLibrary component, we handle the messages requesting one of the above features (cascade, tile horizontally and tile vertically document windows, or arrange minimized windows’ icons). When a request for cascading document windows occurs we perform a call to the CMDIFrameWnd class member function MDICascade(), which takes no parameters, and all document windows arranged as overlapped windows. To arrange document windows as horizontal or vertical tiles we used the MDITile() function with parameters MDITILE_HORIZONTAL and MDITILE_VERTICAL respectively. To arrange all minimized document windows we used the function MDIArrangeIcons(), which also doesn’t take any parameters.
In addition to the options provided for arranging the document windows, there is another option provided by the Window menu, the New Window command, which creates a new frame and view of the current document. The results of the execution of the New Window command are shown in figure 30, where the foreground child window is a clone of that at the background.

Figure 31: The Format dialog.
The *I-GET* Editor also provides the options to the user, to change the formatting of the text (text colour and font), via the format option from the Options menu item, opening the Format dialog box (shown in figure 31).

![Format dialog box](image)

*Figure 31: The default Windows colour and font dialogs.*

In the above dialog box, when the user presses one of the Text Colour or the Background Colour button, the default Windows colour dialog, shown in figure 32 is displayed and the user can pick the desired colour. When the user presses the Text Font button the default Windows font dialog (figure 32) is displayed and the user can specify the desired font.

From the *I-GET* Editor’s Options menu, the user may also specify the compiler options (the compiler executable file, and the compiler command line arguments), through the Compiler option. Those options are required to support the linkage with the *I-GET* language compiler. If the user does not specify those options then, it will not be possible to compile the projects created within the editor.

Finally, from the Options menu the user can add external tools by selecting the Add Tool option. To add such a tool, the user, should specify the tool name, the executable
file and the command line arguments (if any), in the Tools dialog (figure 33). The tool name must be a descriptive name, since it will be added to the tools menu. When there are some tools defined, a new menu option is created in the editor’s menu (entitled Tools), through which the user can use those tools.

Figure 33: The Tools dialog.
CHAPTER 4
APPLICATION TEST SUITE

As mentioned before, apart from the various tests built for the Advanced Windows Toolkit server for testing various functional features, more elaborate large scale applications have been developed, in order to test and validate the development features of the I-GET UIMS, as well as, to carry out extensive testing of the Advanced Windows Toolkit server. Those specific applications are: (a) the Scientific Calculator Application, (b) the Graphics Drawing Application, and (c) the Geometrical Constraints Application. In this paragraph we describe the Scientific Calculator Application.

4.1 Scientific Calculator Application

4.1.1 General description

A Scientific calculator can perform all basic arithmetic operations as well as more advanced mathematical and statistical functions. The basic as well as mathematical operations are available through the calculator main window (figure 34). In case, the users wish to perform statistical operations, they may press the ‘Stat’ button to activate the relevant dialog box. In this box, the numbers sequence inserted by the
user is displayed. The statistical operations the users can perform are the calculation of: (a) the summary of the inserted numbers, (b) their average, and (c) the standard deviation with the population parameter as $n-1$ where $n$ is the total number of the inserted numbers. The users can change these numbers by inserting new values, using the ‘Dat’ button of the calculator window, or by deleting one or more of the existing numbers using the buttons ‘CD’ and ‘CAD’ respectively, which are displayed on the statistic box (figure 33).

![Figure 34: The Scientific Calculator Application.](image)

Figure 34: The Scientific Calculator Application.
The calculator supports four radixes; the hexadecimal, the decimal, the octal and the binary radix. Users can switch between those radixes by selecting the corresponding radio button from the calculator window. Additionally, three trigonometric modes for decimal numbers; the degrees, the radians and the gradients, are supported as well as three representation modes for hexadecimal, octal and binary numbers; the DWORD, WORD and BYTE.

4.1.2 Scientific Calculator User Interface

The design of the calculator user interface was based on the Windows calculator, and as a result it supports most of its functionalities.

The calculator window (see figure 34) can be seen as the composition of four layers. Firstly, there is the top layer containing the display box, in which the results of all operations are shown. Next, is the layer, which includes three group boxes that contain: a) the radix radio buttons, b) the inverse and hyperbolic functions check boxes, and c) the trigonometric modes radio buttons. This layer contains also two display boxes; one that shows the number of opening (left) parentheses without the corresponding closing (right) ones, and the other that displays an $M$ whenever a number is stored in memory. Subsequently, there is the layer that contains buttons for the scientific operations (i.e. summary or average calculation, logarithmic functions, etc), for memory operations (i.e. adding, removing or recalling a number from the memory) and buttons for displaying the hexadecimal numbers (A, B, C, D, E and F). The last layer is the bottom one, which contains the numeric keypad, the simple
arithmetical operations (i.e. add, subtract, divide and multiply) and buttons for
deleting or clearing the currently displayed number.

As mentioned before, besides the calculator window there is the statistic box (see
figure 34). This window contains a list box that holds the numbers inserted by the
user, and four buttons: (a) the ‘Ret’, which switches to the main calculator and retains
the entries in the statistic box, (b) the ‘Load’, which changes the number in the
calculator display area to the number selected in the statistic box, (c) the ‘CD’, which
deletes the selected number, and (d) the ‘CAD’, which deletes all numbers from the
statistic box. Finally, at the bottom of that window there is a label that displays the
number of items in the statistic box.

4.1.3 Purpose of testing

The calculator was implemented to test the functionality of some common objects
such as the Button and the behavior of method notifiers at the Windows Toolkit
Server side. Additionally, hooks and bridges, which mix the I-GET language with
C++ code, were used, to compute the mathematical operations and to implement some
C++ specific functions such as the computation of the factorial of a number.
4.2 Graphics Drawing Application

4.2.1 General description

The Graphics Drawing Application provides the most common capabilities of simple drawing (figure 35). The application uses simple drawing tools such as pen, eraser, and flood-fill as well as tools for drawing basic geometrical shapes such as polygons, rectangles and ellipses. Additionally, it provides the capability for loading and storing image files in a specific file format (bitmap files).

![Figure 35: The Graphics Drawing Application.](image-url)
Finally, the user has a variety of drawing options to select, such as the line width and style (i.e. solid, dashed, dotted, etc), the brush mode (i.e. diagonal, cross, vertical, etc) and the geometrical shape (simple, filled or filled with border).

4.2.2 Graphics Drawing Application User Interface

The Graphics Drawing Application user interface can be seen as the composition of four panes: (a) the left, containing the tool buttons, (b) the middle, containing the drawing canvas, (c) the right, containing the drawing options, and (d) the bottom pane, containing the colour box, the file operations (new, load and store) and the message board where information concerning the currently select tool, the mouse position in the drawing canvas, and the width and height of the currently being drawn shape (only for rectangle, ellipse or round rectangle shapes) is displayed.

![Figure 36: The Tool box.](image)

The buttons contained in the toolbox on the left pane (see figure 36) are CheckButton objects. Those objects are actually buttons that simulate check boxes. They have two states, as check boxes do, the ButtonOn meaning pressed, and ButtonOff meaning un-pressed.
Figure 37: The drawing options.

The drawing canvas is the area in which all the drawing takes place. This canvas is a white rectangle with a black border, which is drawn at startup or whenever a redraw message for the application window is received.

Objects corresponding to drawing options occupy the right pane (see figure 37). There are three list boxes for selecting the line style, the brush mode and the drawing mode, and an edit field in which the user can specify the line width.

The colour box, from where the user can select the foreground and background colours (see figure 38), occupies the left side of the bottom pane. The selected foreground and background colours appear in the left side of this box. The top square represents the foreground, while the bottom represents the background colour. The users can set the foreground colour by performing a left click, and the background colour by performing a right click, on the desired colour button. On the top-right side of that pane, the message board is placed. On this board, the messages concerning the currently select painting tool and the position of the mouse cursor inside the drawing
canvas are displayed. Additionally, if the selected tool is the rectangle, the ellipse or the round rectangle, then, while the user is holding down the mouse button and drags the pointer diagonally, the width and height of the corresponding shape are displayed. On the bottom-right side, there are three buttons and an edit field concerning file operations. Those buttons are:

?? The **New** button, which clears the drawing canvas.

?? The **Open** button, which opens the image with filename the one displayed in the edit field.

?? And the **Save** button, which stores the current picture in a file, with name the one displayed in the edit field.

![Figure 38: The bottom pane.](image)

Finally, the application window has also a menu (named File) with two options: (a) the New, which clears the drawing canvas for a new image, and (b) the Exit, which closes the application.

4.2.3 Purpose of testing

With the development of this application, the graphics drawing capabilities of the Advanced Windows Toolkit server were tested. That was the primary concern while implementing the Graphics Drawing Application. However, with such a test, we had the opportunity to test the event handling (input and output events), too.
As mentioned before, in the *I-GET* language there are two categories of event classes: (a) the output event, and (b) the input event classes. While output events can be ‘called’ as conventional function-calls, the input events must be associated with event handlers. In the current application event handlers are registered for the main window, which holds the drawing canvas. The event classes necessary for the drawing are: (a) the *Paint*, (b) the *MouseMove*, (c) the *Left/Right ButtonPressed*, (d) the *Left/Right ButtonUp*, and (e) the *Left/Right ButtonDoubleClick*.

The difference between the drawing using the left button and the drawing using the right button is that in the second case the foreground and background colours are used in reverse.

The graphics drawing capabilities of the Advanced Windows Toolkit server were tested with the use of output events. All functions used by the application and concern the drawing capabilities, provided by the toolkit server, are defined as output events.

Finally, another part of the toolkit server that was tested was the attribute modification handlers. For most of the objects used we needed to change some of their attributes at run-time, such as the state of the CheckButtons and the colours of the boxes that display the foreground and background colours.
4.3 Geometrical Constraints Application

4.2.1 General description

The final test application we have developed is the Geometrical Constraints Application (see figure 39), which is a demonstration of the Line and Rectangle objects, and the constraints among their attributes.

![The Geometrical Constraints Application](image)

*Figure 39: The Geometrical Constraints Application.*

The Diamond (Lines) is a mathematical test that demonstrates how the Line object can be used with constraints, and it shows a parallelogram, which is surrounded by another one. The outer parallelogram can be resized in any direction while the inner one, which its edges begin and end from the middle of the edges of the outer
parallelogram, is resized and changes shape but it remains a parallelogram, although
the outer one does not.

The Rectangles is a visual test where four rectangles are contained in an outer one,
with their widths and heights being constrained with those of the outer rectangle.
When the outer rectangle is resized, the inner rectangles change their width and height
but remain in the same relative position.

4.2.2 Geometrical Constraints Application User Interface

The Geometrical Constraints Application has the less complex user interface of the
three test applications described in this chapter. The application has a main window
with three buttons: a) the ‘Lines’ button, which opens the Lines demonstration
window, b) the ‘Rectangles’ button, which opens the Rectangles demonstration
window and c) the ‘Quit’ button for closing the application.

The Lines window is a simple FrameWindow object, which in its client area the
parallelograms are displayed. These parallelograms consist of Line objects (Windows
expansion interactive object) that have interactive behavior.

The Rectangles window is also a FrameWindow object and displays four rectangles
surrounded by another. Each of those shapes is a Rectangle object (Windows
expansion interaction object) with interactive behavior.

4.2.3 Purpose of testing

The purpose of building that application was to test the imported Windows expansion
interactive objects (Lines and Rectangles) and the constraints among their attributes.
Constraints are relationships, which are forced on engaged variables. In the Lines demonstration application the declared constraints are: a) the ending point of a line is the starting for another line and b) the middle point of a line is the starting point for another and the ending point for a third line.

Assume that we have three Line objects i.e. l1, l2, l3, and they have the coordinates x1, y1 and x2, y2 for the starting and ending point respectively. Then the constraints between those coordinates for the first case are as follows:

\[
\{l2\}.x1 := \{l1\}.x2; \quad \{l1\}.x2 := \{l2\}.x1; \\
\{l2\}.y1 := \{l1\}.y2; \quad \{l1\}.y2 := \{l2\}.y1;
\]

And for the second case are:

\[
\{l3\}.x1 := (\{l1\}.x1 + \{l1\}.x2)/2; \quad \{l3\}.y1 := (\{l1\}.y1 + \{l1\}.y2)/2; \\
\{l2\}.x2 := (\{l1\}.x1 + \{l1\}.x2)/2; \quad \{l2\}.y2 := (\{l1\}.y1 + \{l1\}.y2)/2;
\]

In the Rectangles demonstration we have two categories of constraints: (a) those concerning a single rectangle, and (b) those concerning the relationships between the inner rectangles and the outer one. The constraints for a single rectangle with top-left point the (x1, y1) and bottom-right point the (x2, y2) are: a) the width of the rectangle is always the absolute value of (x2 – x1), and b) the height is always the absolute value of (y2 – y1).

Finally, the relationships between the outer and the inner rectangles are: a) the width and height, of an inner rectangle, are the 1/5 of the width W and height H of the outer rectangle respectively (i.e. \(W_{\text{inner}} := 1/5*W_{\text{outer}}, \quad H_{\text{inner}} := 1/5*H_{\text{outer}}\)), and b) the
coordinates of an inner rectangle $R_{\text{inner}}$ are constrained, with those of the outer one $R_{\text{outer}}$, as follows:

**Top-Left inner rectangle:**

\[
\begin{align*}
R_{\text{inner}}'_{x1} &= R_{\text{outer}}.x1 + W_{\text{inner}}; \\
R_{\text{inner}}'_{y1} &= R_{\text{outer}}.y1 + H_{\text{inner}}; \\
R_{\text{inner}}'_{x2} &= R_{\text{outer}}.x1 + 2* W_{\text{inner}}; \\
R_{\text{inner}}'_{y2} &= R_{\text{outer}}.y1 + 2*H_{\text{inner}};
\end{align*}
\]

**Top-Right inner rectangle:**

\[
\begin{align*}
R_{\text{inner}}'_{x1} &= R_{\text{outer}}.x1 + 3*W_{\text{inner}}; \\
R_{\text{inner}}'_{y1} &= R_{\text{outer}}.y1 + H_{\text{inner}}; \\
R_{\text{inner}}'_{x2} &= R_{\text{outer}}.x1 + 4* W_{\text{inner}}; \\
R_{\text{inner}}'_{y2} &= R_{\text{outer}}.y1 + 2*H_{\text{inner}};
\end{align*}
\]

**Bottom-Left inner rectangle:**

\[
\begin{align*}
R_{\text{inner}}'_{x1} &= R_{\text{outer}}.x1 + W_{\text{inner}}; \\
R_{\text{inner}}'_{y1} &= R_{\text{outer}}.y1 + 3*H_{\text{inner}}; \\
R_{\text{inner}}'_{x2} &= R_{\text{outer}}.x1 + 2* W_{\text{inner}}; \\
R_{\text{inner}}'_{y2} &= R_{\text{outer}}.y1 + 4*H_{\text{inner}};
\end{align*}
\]

**Bottom-Right inner rectangle:**

\[
\begin{align*}
R_{\text{inner}}'_{x1} &= R_{\text{outer}}.x1 + 3*W_{\text{inner}}; \\
R_{\text{inner}}'_{y1} &= R_{\text{outer}}.y1 + 3*H_{\text{inner}}; \\
R_{\text{inner}}'_{x2} &= R_{\text{outer}}.x1 + 4* W_{\text{inner}}; \\
R_{\text{inner}}'_{y2} &= R_{\text{outer}}.y1 + 4*H_{\text{inner}};
\end{align*}
\]
CHAPTER 5
SUMMARY, CONCLUSIONS & FUTURE WORK

5.1 Summary

The overall objective of this work is to support development environments for universal access. In this context, the I-GET UIMS has been selected as the tool offering the most advanced and promising features for the engineering of universal access, while it served as the target development environment for the work that has been carried out.

More specifically, the work carried out concerned: (a) the development of an Advanced Windows toolkit server for the I-GET UIMS; (b) the design and implementation of a large-scale application test-suite for the I-GET UIMS; and (c) the construction of an appropriate Interactive Programming Environment for the I-GET UIMS.

While building the Windows toolkit server the main implementation tasks have been:

(a) The Windows Toolkit programming interface specification

This task concerns the specification of the various imported (lexical-level) interaction objects classes, input events and output events, following the formation of the I-GET language.
(b) The development of the Windows Toolkit server

The development of the Toolkit Server is logically split in two parts. The first concerns handling of all requests made by the dialogue control for instantiating interaction objects, modifying object attributes and performing output events. The second concerns posting of all the various run-time notifications back to the dialogue control; such notifications concern input events, method notifications for object instances, as well as attribute modifications due to user interaction.

(c) The development of the Interactive Programming Environment

The Interactive Environment is a source code editor, with various capabilities such as: syntax highlighting of language keywords, workspace management, and external compiler linkage. A distinctive implementation feature of the editor is that it is composed of programmably extensible components.

(d) The construction of extensive large-scale application tests

Those played a twofold role. On the one hand they have demonstrated the powerful features of the I-GET language, while on the other hand they have served as a comprehensive test-bed for the Advanced Windows toolkit server. Those applications are:

?? Scientific Calculator Application, which supports all basic as well as scientific operations. The design of the calculator was based on the Windows calculator, while it supports most of its functions.

?? Graphics Drawing Application, providing most common features found in drawing packages. This application offers basic drawing facilities such as pen, eraser, flood fill, while supporting common geometrical shapes like: lines,
polygons, rectangles, and ellipses. Also, it supports loading and storing of drawing files in bitmap format.

?? Geometrical Constraints Application, which demonstrates the use of Line and Rectangle objects in conjunction to the employment of constraints in the I-GET language. The Diamond application shows a parallelogram constrained inside another parallelogram, while the Rectangle application shows four rectangles constrained inside a main rectangle.

5.2 Conclusions

The I-GET UIMS is a powerful tool for the engineering of universally accessible interactions. Reflecting the key objective of this work, being the practical support of development environments for universal access, a number of concrete milestones have been identified. During the development work to reach those milestones, it quickly became evident that those milestones engage demanding development tasks. In this process, a number of conclusions have been drawn concerning: (a) the overall technical approach for the conduct of those development efforts; (b) reporting experience regarding the use of the I-GET UIMS; and (c) run-time performance issues. A detailed discussion regarding all those issues follows.

The architecture of the toolkit server enabled vertical expansion of interaction objects, attributes, input and output events, with a well manageable complexity. Additionally, the toolkit interface specification capabilities of the I-GET UIMS enabled easily the integration of interaction elements from the Windows Object library. Moreover, the toolkit integration model of the I-GET language supports toolkit expansion (Savidis &
Stephanidis, 2000, [9]) in a way that the original toolkit interaction elements can be easily expanded without affecting the native toolkit libraries.

During the development of the Interactive Programming Environment it has been proved that the support for the syntax highlighting has been a largely demanding implementation task. The study of editors with similar features (i.e. Microsoft Visual Studio editor) helped in the interface design phase. Additionally, the decomposition of the overall architecture into independent interoperating components probably reduced the implementation complexity, while it enabled for further reusability of each distinct component.

During the development of the test-specific applications, various conclusions concerning the use of the I-GET language have been extracted. Firstly, some development has been accomplished quite easily through the mechanisms of the I-GET language like:

**?? Construction of dialogue components.** Following the interface design phase, the translation to a working implementation has been proved to be a very simple task, requiring a relatively small amount of code, engaging merely the declaration of the respective object instances, as well as the initialisation of the physical attributes.

**?? Method notification and event handling** is straightforward in the I-GET language, since it does not require declaration of handler functions with a special signature, nor the definition of the message map; instead, only the body of the handler functions is supplied, containing the event handling logic.
Object attributes modification is facilitated by directly assigning values to object attributes, without the need of calling any toolkit specific functions.

Secondly, some lower-level programming tasks, not related to the dialogue management and control, were less simple to be implemented in comparison to a general-purpose language like C++. One such example concerns the list implementation, which required implementation of the various operations such as the insertion and deletion of list elements, entirely in I-GET language.

Additionally, there were some operations that were not supported by the I-GET language, and they have been implemented using C++ functions, taking advantage of the hooks and bridges facility of the I-GET language. Although the basic arithmetical operations (i.e. add, subtract, multiply and divide) could be carried out using the I-GET language, the calculation of some mathematical functions, such as the trigonometric and logarithmic, which have been used to implement the corresponding Scientific Calculator operations, were all implemented in C++, while called inside the dialogue components implementation, the latter being pure I-GET language code.

During the testing of the Advanced Windows toolkit server particular observations reporting its run-time performance have been made. Those have mainly been related to resource demanding tasks such as:

The inter-process communication between the toolkit server and the Dialogue control component is a largely resource-demanding task, whose performance is inversely proportional to the size of the communicated data buffers.
The functionality of the mapping table, which holds information regarding the toolkit-specific object instances at run-time, affect seriously the overall performance since called by the toolkit server for almost all sorts of requests posted by the dialogue control component (object instantiation, attributes modification, methods notification, input and output event handling). Currently, the mapping is not implemented via hashing but via a sequential searching; one possible extension to increase performance would be the implementation of a fast hashing scheme.

The string comparison for attributes, object classes and output events, for the corresponding requests, is another necessary resource demanding task.

Finally, performance differences have been largely observed in the employment of the two lowest level alternative communication approaches between the toolkit server and the Dialog Control, being shared memory and TCP/IP sockets. For communication intensive applications, such as the Graphics Drawing application, that engaged inner feed-back loops, it has been observed that the performance with shared memory appeared to be superior, while for less communication demanding applications, such as the Scientific Calculator, engaging mostly method notifications and attribute modifications, performance difference has been insignificant.

5.3 Future work

Future work is mainly related to the extension of the Advanced Windows Toolkit server, as well as the Interactive Programming Environment. Those extensions are:
Integration of a Multimedia API. The introduction of multimedia support in the Advanced Windows Toolkit server is a demanding implementation task, requiring careful extension of both the Windows toolkit programming interface specification, as well as the toolkit server software implementation. The multimedia extension can be implemented using the Microsoft DirectX set of application programming interfaces (i.e. DirectDraw, DirectSound, and DirectShow) to gain access directly to system’s multimedia hardware.

Language specific extensions to the Interactive Environment. Features such as the auto-filling of language keywords, as well as tool-tip feedback over functions, keywords and program variables, are some of the potential extensions.
APPENDIX A

CODE EXCERPTS FROM THE SCIENTIFIC CALCULATOR APPLICATION

//-----------------------------------------------------
// Scientific Calculator Application.
//-----------------------------------------------------
// Iget header files.
headers [extern ["igetwin.hh", "igetrowcol.hh"]
#include "win.gdd"
#include "rowcol.hdr"
]
@
///////////////////////////////////////////////////////
// C++ hooks header files.
_c++[
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
//****************************************************
// Computes the factorial of number n.
double nParag(double n){
double parag=1;
while (n>0){
parag=parag*n;
 n--;
}
return parag;
}_{
//****************************************************
// Macros defining the operation to be taken.
#define NOACT 0
#define ADDACT1
#define SUBACT2
#define MULACT3
#define DIVACT4
.
//**********************************************************
agent Calculator;
ref agent Calculator calcInst; //a reference to Calculator agent.

//-----------------------------------------------------
// Create the calculator agent
agent Calculator create if (true) {

public:

Calculators main window.
lexical(DeskTop) FrameWindow calcWin;

Display
lexical(DeskTop) StaticControl calcdisp:parent={calcWin};
lexical(DeskTop) StaticControl ldisp:parent={calcWin};
lexical(DeskTop) StaticControl memdisp:parent={calcWin}; //Mem indicator.

Numerical buttons
lexical(DeskTop) Button zero:parent={calcWin};
lexical(DeskTop) Button one:parent={calcWin};
lexical(DeskTop) Button two:parent={calcWin};
lexical(DeskTop) Button three:parent={calcWin};
lexical(DeskTop) Button four:parent={calcWin};
lexical(DeskTop) Button five:parent={calcWin};
lexical(DeskTop) Button six:parent={calcWin};
lexical(DeskTop) Button seven:parent={calcWin};
lexical(DeskTop) Button eight:parent={calcWin};
lexical(DeskTop) Button nine:parent={calcWin};

// CALCULATOR MAIN FUNCTIONS
// Initialize the parenthesis result to zero.
void InitParenthesis(){
  parCount=0;
  last_par_act=NOACT;
  int i;
  for (i=0;i<25;i++)
  { parRes[i]=0; }
  {ldisp}.label = "";
}

// Calculates the number that divs the active number
// when we want to del the last digit of the active number.
// Parameter is the number system (Hex, Dec, Oct or Bin mode).
int delDiv(int mode){
  case(mode){
    HEXMODE: return 16;
    DECMODE: return 10;
    OCTMODE: return 8;
    BINMODE: return 2;
  }
}

// Converts the active number to the new mode (DWORD, WORD or BYTE)
// when the number system is HEX, OCT or BIN.
// The return value is the new number.
// Parameters are the number to be converted, the new number mode and the old
// number mode (DWORD, WORD or BYTE).
Appendix A: Code excerpts from the Scientific Calculator Application

```c
longint modeNum(longint num, int mode, int oldMode)
{
    case (mode)[
        DWORD: return num;
        WORD:[
            case (oldMode)[
                DWORD:[
                    _c++ "int temp;";
                    "temp=" [-] num;
                    _c++ "int res = temp >> 16;";
                    longint tmp;
                    tmp [<-] "res";
                    return tmp;
                ]
                WORD: return num;
                BYTE: return num;
            ]
            BYTE:[
                case (oldMode)[
                    DWORD:[
                        _c++ "int temp;";
                        "temp=" [-] num;
                        _c++ "int res = temp >> 24;";
                        longint tmp;
                        tmp [<-] "res";
                        return tmp;
                    ]
                    WORD:
                    BYTE:
                ]
            ]
        ]
    ]
    . . .
}

agent StatWin create if (makeStat==true)
{
    public:
    //**********************************************************
    // The statistical window
    lexical(DeskTop) FrameWindow nwin;
    //**********************************************************
    // The listbox that holds the numbers
    lexical(DeskTop) ListBox lb:parent={nwin};
    //**********************************************************
    // Static text that displays the number of items in the listbox
    lexical(DeskTop) StaticControl count:parent={nwin};
    //**********************************************************
    // Buttons appeared in the statistical window.
    // ret : sets the calculators main window active
    // load: sets the selected number of the listbox as
    // the active in calculators main window active
    // cd : deletes the select number in the listbox
    // cad : deletes all numbers from the listbox
    lexical(DeskTop) Button ret:parent={nwin};
    lexical(DeskTop) Button load:parent={nwin};
    lexical(DeskTop) Button cd:parent={nwin};
    lexical(DeskTop) Button cad:parent={nwin};
    
    // Agent that creates the statistical window when the stat button pressed.
    //
```
eventhandler {nwin} [
   {true} Destroyed [
      makeStat=false;
   ]
]

// local functions for the statistical box

// Sets the number of the items in listbox as label in statistical box
void SetNumCount()
{
   {count}.label = "n="+ncount;
}

// Returns the mean value of the numbers in the listbox
longreal GetAvg()
{
   if (ncount>0) return lb_sum/lb.N;
}

// Returns the mean of the squares of the numbers displayed in the listbox
longreal GetSqrAvg()
{
   if (ncount>0) return lb_sqrsum/ncount;
}

// method executed when the button ret pressed in statistical box.
// Just set focus to Calculators main window
method {ret}.Pressed [
   out(DeskTop).SetFocus({calcInst}.calcWin);
]

// method executed when the button load pressed in statistical box.
// Loads the selected number of the listbox to calculators display.
method {load}.Pressed [
   longreal tmp = strtolongreal({lb}.subjectItem);
   {calcdisp}.label = displayNum(tmp,num_mode);
   active_num= tmp;
   from_act=1;
]

Font sFont=
{
   "Arial",
   9,
   false,
   false,
   false,
   false,
   0,
   0
};

Colour blk=[
   0,
   0,
   0
];

Colour wht=[
   255,
   255,
   255
];
Appendix A: Code excerpts from the Scientific Calculator Application

```forth
255,
255 ];

constructor[
    aids = {myagent};
    ncount=0;
    statwin = {nwin};
    lbid={lb};
    {nwin}.x=0;
    {nwin}.y=0;
    {nwin}.width=260;
    {nwin}.height=260;
    {nwin}.status = WinNormal;
    {nwin}.hasMinimizeBox = false;
    {nwin}.hasMaximizeBox = false;
    {nwin}.title = "Statistics Box";

    {lb}.x=5;
    {lb}.y=5;
    {lb}.width = 240;
    {lb}.height = 120;
    {lb}.N = 0;
    {lb}.columnsNo = 1;

    {ret}.x=10;
    {ret}.y=150;
    {ret}.width = 50;
    {ret}.height= 40;
    {ret}.hasText=true;
    {ret}.hasBitmap=false;
    {ret}.label="RET";
    {ret}.font=sFont;
    {ret}.textColour=blk;

    . . .
    out(DeskTop).RealizeObject({nwin});
]

destructor[
    makeStat=false;
    ncount=0;
    destroy aids;
    out(DeskTop).DestroyObject({nwin});
]
]

//**********************************************************
// a reference to agent StatWin
ref agent StatWin aid=nil;

//**********************************************************
// method executed when a menu item selected in calculators main window.
method {menu}.Selected [
    if ({menu}.selectedItemId == 0){
        terminate;
    }
]

//**********************************************************
// method executed when button '0' pressed in calculators main window.
// puts the 0 in calculators display.
method {zero}.Pressed [
    if (from_act==1){
        active_num=0;
        from_act=0;
    }
]
```

Page 83

Development Environments for Universal Access, TR_319 2003, ICS-FORTH
if (commaPressed==0)
    active_num = makeNum(active_num,0,num_mode);
else [
    div_num = div_num/10;
]
{calcdisp}.label = displayNum(active_num,num_mode);

(controller)

Appendix A: Code excerpts from the Scientific Calculator Application

//**********************************************************
// method executed when button '1' pressed in calculators main window.
// puts the 1 in calculators display.
method {one}.Pressed [
    if (from_act==1)[
        active_num=0;
        from_act=0;
    ]
    if (commaPressed==0){
        active_num = makeNum(active_num,1,num_mode);
    } else {
        if (active_num>=0) active_num = active_num + div_num;
        else active_num = active_num - div_num;
        div_num = div_num/10;
    }
{calcdisp}.label = displayNum(active_num,num_mode);
]

constructor [
    calcInst={myagent};
    calcWin= {calcWin};
    InitParenthesis();
    cdsp = {calcdisp};
    Colour wincolour=[
        190,
        190,
        190
    ];
    {calcWin}.x=0;
    {calcWin}.y=0;
    {calcWin}.width = 370;
    {calcWin}.height = 650;
    {calcWin}.title="Scientific Calculator";
    {calcWin}.status = WinNormal;
    {calcWin}.hasMinimizeBox = true;
    {calcWin}.hasMaximizeBox = false;
    {calcdisp}.label = "0";
    {calcdisp}.alignment = AlignRight;
    {calcdisp}.bkColour = gray;
    {calcdisp}.font = displayFont;
    {group1}.x = 5;
    {group1}.y = 40;
    {group1}.width = 225;
    {group1}.height = 40;
    {group1}.bkColour = gray;
    {hex}.x = 10;
    {hex}.y = 50;
    {hex}.width = 50;
Appendix A: Code excerpts from the Scientific Calculator Application

```plaintext
{hex}.height = 25;
{hex}.title = "Hex";
{hex}.font = smallFont;
{hex}.bkColour = gray;

{dec}.x = 65;
{dec}.y = 50;
{dec}.width = 50;
{dec}.height = 25;
{dec}.title = "Dec";
{dec}.font = smallFont;
{dec}.bkColour = gray;

{oct}.x = 120;
{oct}.y = 50;
{oct}.width = 50;
{oct}.height = 25;
{oct}.title = "Oct";
{oct}.font = smallFont;
{oct}.bkColour = gray;

{bin}.x = 175;
{bin}.y = 50;
{bin}.width = 50;
{bin}.height = 25;
{bin}.title = "Bin";
{bin}.font = smallFont;
{bin}.bkColour = gray;

{group2}.x = 5;
{group2}.y = 80;
{group2}.width = 260;
{group2}.height = 40;
{group2}.bkColour = gray;

{degrees_dword}.x = 10;
{degrees_dword}.y = 90;
{degrees_dword}.width = 80;
{degrees_dword}.height = 25;
{degrees_dword}.title = "Degrees";
{degrees_dword}.font = smallFont;
{degrees_dword}.bkColour = gray;

{radians_word}.x = 95;
{radians_word}.y = 90;
{radians_word}.width = 80;
{radians_word}.height = 25;
{radians_word}.title = "Radians";
{radians_word}.font = smallFont;
{radians_word}.bkColour = gray;

{gradients_byte}.x = 180;
{gradients_byte}.y = 90;
{gradients_byte}.width = 80;
{gradients_byte}.height = 25;
{gradients_byte}.title = "Gradients";
{gradients_byte}.font = smallFont;
{gradients_byte}.bkColour = gray;

{group3}.x = 235;
{group3}.y = 40;
{group3}.width = 120;
{group3}.height = 40;
{group3}.bkColour = gray;

{inv}.x = 10;
{inv}.y = 10;
{inv}.width = 50;
{inv}.height = 25;
{inv}.title = "Inv";
{inv}.font = smallFont;
{inv}.bkColour = gray;
```

Page 85

Development Environments for Universal Access, TR_319 2003, ICS-FORTH
Appendix A: Code excerpts from the Scientific Calculator Application

\(\{hyp\}.x = 70;\)
\(\{hyp\}.y = 10;\)
\(\{hyp\}.width = 45;\)
\(\{hyp\}.height = 25;\)
\(\{hyp\}.title = "Hyp";\)
\(\{hyp\}.font = smallFont;\)
\(\{hyp\}.bkColour = gray;\)

\(\{ldisp\}.x = 270;\)
\(\{ldisp\}.y = 90;\)
\(\{ldisp\}.look3D = true;\)
\(\{ldisp\}.width = 30;\)
\(\{ldisp\}.height = 30;\)
\(\{ldisp\}.isVisible = true;\)
\(\{ldisp\}.font = displayFont;\)
\(\{ldisp\}.bkColour = gray;\)

\(\{memdisp\}.x = 315;\)
\(\{memdisp\}.y = 90;\)
\(\{memdisp\}.look3D = true;\)
\(\{memdisp\}.isVisible = true;\)
\(\{memdisp\}.bkColour = gray;\)
\(\{memdisp\}.width = 30;\)
\(\{memdisp\}.height = 30;\)
\(\{memdisp\}.font = displayFont;\)

\(\{stat\}.x = 45;\)
\(\{stat\}.y = 150;\)
\(\{stat\}.width = 35;\)
\(\{stat\}.height = 35;\)
\(\{stat\}.hasText = true;\)
\(\{stat\}.hasBitmap = false;\)
\(\{stat\}.label = "Stat";\)
\(\{stat\}.font = simpleFont;\)
\(\{stat\}.textColour = blue;\)

\(\{avg\}.x = 85;\)
\(\{avg\}.y = 150;\)
\(\{avg\}.width = 35;\)
\(\{avg\}.height = 35;\)
\(\{avg\}.hasText = true;\)
\(\{avg\}.hasBitmap = false;\)
\(\{avg\}.label = "Avg";\)
\(\{avg\}.font = simpleFont;\)
\(\{avg\}.textColour = blue;\)

\(\{sum\}.x = 125;\)
\(\{sum\}.y = 150;\)
\(\{sum\}.width = 35;\)
\(\{sum\}.height = 35;\)
\(\{sum\}.hasText = true;\)
\(\{sum\}.hasBitmap = false;\)
\(\{sum\}.label = "Sum";\)
\(\{sum\}.font = simpleFont;\)
\(\{sum\}.textColour = blue;\)

\(\{s\}.x = 165;\)
\(\{s\}.y = 150;\)
\(\{s\}.width = 35;\)
\(\{s\}.height = 35;\)
\(\{s\}.hasText = true;\)
\(\{s\}.hasBitmap = false;\)
\(\{s\}.label = "s";\)
\(\{s\}.font = simpleFont;\)
\(\{s\}.textColour = blue;\)

\(\{dat\}.x = 205;\)
\(\{dat\}.y = 150;\)
\(\{dat\}.width = 35;\)
\(\{dat\}.height = 35;\)
Appendix A: Code excerpts from the Scientific Calculator Application

```
{dat}.hasText=true;
{dat}.hasBitmap = false;
{dat}.label = "Dat";
{dat}.font = simpleFont;
{dat}.textColour = blue;

{bdms}.x = 45;
{bdms}.y = 190;
{bdms}.width = 35;
{bdms}.height = 35;
{bdms}.hasText=true;
{bdms}.hasBitmap=false;
{bdms}.label = "dms";
{bdms}.font = simpleFont;
{bdms}.textColour = redblue;

{bexp}.x = 85;
{bexp}.y = 190;
{bexp}.width = 35;
{bexp}.height = 35;
{bexp}.hasText=true;
{bexp}.hasBitmap=false;
{bexp}.label = "Exp";
{bexp}.font = simpleFont;
{bexp}.textColour = redblue;

{bln}.x = 125;
{bln}.y = 190;
{bln}.width = 35;
{bln}.height = 35;
{bln}.hasText=true;
{bln}.hasBitmap=false;
{bln}.label = "ln";
{bln}.font = simpleFont;
{bln}.textColour = redblue;

...

{clear_all}.x = 245;
{clear_all}.y = 440;
{clear_all}.width = 35;
{clear_all}.height = 35;
{clear_all}.textColour = red;
{clear_all}.hasText = true;
{clear_all}.hasBitmap = false;
{clear_all}.label = "C";
{clear_all}.font = simpleFont;

{del}.x = 245;
{del}.y = 480;
{del}.width = 35;
{del}.height = 35;
{del}.textColour = red;
{del}.hasText = true;
{del}.hasBitmap = false;
{del}.label = "del";
{del}.font = simpleFont;

{avg}.enabled = false;
{sum}.enabled = false;
{s}.enabled = false;
{dat}.enabled = false;
{A}.enabled = false;
{B}.enabled = false;
{C}.enabled = false;
{D}.enabled = false;
{E}.enabled = false;
{F}.enabled = false;
{dec}.state = RadioOn;
{inv}.state=CheckOff;
{hyp}.state=CheckOff;
```
flag = DEGR;
{degrees_dword}.state = RadioOn;
    out(DeskTop).RealizeObject({calcWin});
}
destructor[out(DeskTop).DestroyObject({calcWin});]
APPENDIX B

CODE EXCERPTS FROM THE GRAPHICS DRAWING APPLICATION

//----------------------------------------------------------
// Graphics Drawing Application.
//----------------------------------------------------------
// I-Get Draw header files.
headers [extern ["igetwin.hh", "igetrowcol.hh"]
#include "win.gdd"
#include "rowcol.hdr"
]
@21
...

//----------------------------------------------------------------------------
// Enumerated attribute defining the tool to be used
enum ToolMode = [Eraser, Flood,
    Line, Curve,
    Rect, FillRect,
    Ellipse, RRect,
    Pen, Brush,
    Poly, Arc,
    Pie, DText,
    PicCol, Empty];
//----------------------------------------------------------------------------
// Attributes used for Holds coordinates for drawing geometrical shapes.
//
local int rect_x; // Holds starting coordinates for rectangles, ellipses and
    round rectangles.
local int rect_y;
local int line_x; // Holds starting coordinates for lines.
local int line_y;
local int poly_x; // Holds starting coordinates for polygons.
local int poly_y;
local int pmouse_x; // Holds the previous position of the mouse pointer.
local int pmouse_y;
local int arc_Start_x;// Holds the starting point for a pie or an arc.
local int arc_Start_y;
local int arc_End_x;// Holds the ending point for a pie or an arc.
local int arc_End_y;
local int Xradius; // Defines the radius to be used for round
    rectangles.
local int Yradius;
local int mposx,mposy;// Holds the position of the mouse pointer in drawing
    rectangle.
local int *curve_x; // Holds the points of a curve.
local int *curve_y;

...
struct PointList;
struct PointList {
    int xpoint;
    int ypoint;
    int pos;
    PointList* next;
};
local PointList* head=nil;
local PointList* tail=nil;
local int pointN=0;
...
agent Draw create if (true)[
    lexical (DeskTop) FrameWindow mWin;
    lexical (DeskTop) RowColumnGroupBox tools:parent={mWin};
    lexical (DeskTop) RowColumnGroupBox bcols:parent={mWin};
    lexical (DeskTop) RowColumnGroupBox styles:parent={mWin};
    lexical (DeskTop) RowColumnGroupBox currCols:parent={mWin};
    lexical (DeskTop) RowColumnGroupBox fBox:parent={mWin};
    lexical (DeskTop) Menu fmenu:parent={mWin};
    lexical (DeskTop) Menu vmenu:parent={mWin};
    lexical (DeskTop) Menu menu;
    lexical (DeskTop) CheckButton eraser:parent={tools};
    lexical (DeskTop) CheckButton ffill:parent={tools};
    lexical (DeskTop) CheckButton line:parent={tools};
    lexical (DeskTop) CheckButton curve:parent={tools};
    lexical (DeskTop) CheckButton rect:parent={tools};
    lexical (DeskTop) CheckButton ellipse:parent={tools};
    lexical (DeskTop) CheckButton roundRect:parent={tools};
    lexical (DeskTop) CheckButton pen:parent={tools};
    lexical (DeskTop) CheckButton brush:parent={tools};
    lexical (DeskTop) CheckButton arc:parent={tools};
    lexical (DeskTop) CheckButton pie:parent={tools};
    lexical (DeskTop) CheckButton polygon:parent={tools};
    lexical (DeskTop) CheckButton piccol:parent={tools};
    lexical (DeskTop) StaticControl backlabel:parent={currCols};
    lexical (DeskTop) StaticControl forelabel:parent={currCols};
    lexical (DeskTop) StaticControl backCol:parent={currCols};
    lexical (DeskTop) StaticControl foreCol:parent={currCols};
    lexical (DeskTop) StaticControl cbLabel:parent={styles};
    lexical (DeskTop) StaticControl xycoord:parent={coords};
    lexical (DeskTop) StaticControl tooltip:parent={coords};
...
    //**********************************************************
    // Returns the previous drawing mode
    DrawMode GetPrevMode(int mode){
        case mode:{
            1: return DrawBlack;
            2: return DrawWhite;
        }
    }
]
Appendix B: Code excerpts from the Graphics Drawing Application

3:  return DrawNop;
4:  return DrawNot;
5:  return DrawCopy;
6:  return DrawNotCopy;
7:  return DrawMergeNot;
8:  return DrawMaskNot;
9:  return DrawMerge;
10: return DrawNotMerge;
11: return DrawMask;
12: return DrawNotMask;
13: return DrawXor;
14: return DrawNotXor;
}

//**********************************************************
// Checks if a point (x1,y1) is in the given rectangle (x,y,w,h).
bool PtInRect(int x, int y, int w, int h, int x1, int y1) {
  if ((x1<=w)&&((x1+w)>x)&&(y1<=h)&&(y1+h)>y)) {
    return true;
  }
  else return false;
}

//**********************************************************
// Adds a point (x,y) in the point list.
void AddPoint(int x, int y) {
  PointList* p;
  p = new(p,1);
  p->xpoint = x;
  p->ypoint = y;
  p->pos = pointN;
  p->next = nil;
  if (!head) {
    head = tail = p;
  }
  else {
    tail->next = p;
    tail = p;
  }
  pointN++;
}

//**********************************************************
// Assigns the points in list in the arrays X and Y.
void AssignPoints(int* X, int* Y) {
  PointList *p = head;
  while (p) {
    X[p->pos] = p->xpoint;
    Y[p->pos] = p->ypoint;
    p = p->next;
  }
  int i;
  for (i=0; i<pointN; i++) {
    printf("pointX["%d"] = "+X[i]+",  ");
    printf("pointY["%d"] = "+Y[i]+"
");
  }
}

//**********************************************************
// Clears the PointList structure.
void EmptyPointList() {
  PointList *aux;
}
while (head) [ 
    aux=head; 
    head=head->next; 
    release(aux); 
] 

tail=nil; 
pointN = 0; 

. . .

//**********************************************************
// Clears the drawing area .
// Ckears the drawing area .
//**********************************************************
// void NewPic()
void NewPic()
    
    out(DeskTop).SetDrawColour(gid,{mWin},spColour,white);
    out(DeskTop).SetLineStyle(gid,{mWin},Solid);
    out(DeskTop).SetLineWidth(gid,{mWin},1);
    out(DeskTop).SetBrush(gid,{mWin},Fill);
    out(DeskTop).Rectangle(gid,{mWin},xarea,yarea,warea-1,harea-1,
                 BorderFilled);
    out(DeskTop).SetLineStyle(gid,{mWin},lstyle);
    out(DeskTop).SetLineWidth(gid,{mWin},lwidth);
    out(DeskTop).SetDrawColour(gid,{mWin},fgColour,bkColour);
] 

// monitores monitores

//**********************************************************
// This monitor frees the lists of points when we
// change the drawing tool and unchecks
// the previous tool.
//**********************************************************
// currMode: [ 
// EmptyPointList(); 
// release(curve_x);
// release(curve_y);
// release(pcurve_x);
// release(pcurve_y);

currMode: [ 
    EmptyPointList();
    release(curve_x);
    release(curve_y);
    release(pcurve_x);
    release(pcurve_y);

    curveN=0;
    UnCheckTool(lastMode);

    locked=false;
]

. . .
Appendix B: Code excerpts from the Graphics Drawing Application

```c
out(DeskTop).FileLoadBitmap(picfile,&gbid);
out(DeskTop).FileDisplayBitmap(gbid, {mWin}, xarea+1, yarea+1, 1, 1, &gstatus);

//(true) LeftButtonPressed [
  bool check = PtInRect(xarea+1, yarea+1, warea-2, harea-2, LeftButtonPressed.x, LeftButtonPressed.y);
  if (check==true)[
    locked = true;
    AllocateGraphicsCheck();
    out(DeskTop).MoveTo(gid, {mWin}, LeftButtonPressed.x, LeftButtonPressed.y);
  ]

  case (currMode)[
    // For rectangles, ellipses and round rectangles we get the starting point
    Rect, Ellipse, RRect:
      rect_x = LeftButtonPressed.x;
      rect_y = LeftButtonPressed.y;
  ]

  // Fills the area around the point with the current colour
  Flood:
    out(DeskTop).FloodFill(gid, {mWin}, LeftButtonPressed.x, LeftButtonPressed.y, fgColour);
    locked=false;

  // Moves the point of the graphics context at the new one and holds the starting point
  // the line.
  Line, Pen, Brush, Eraser:
    line_x = LeftButtonPressed.x;
    line_y = LeftButtonPressed.y;

  // Gets the starting point for the arc.
  Arc:
    arc_Start_x = LeftButtonPressed.x;
    arc_Start_y = LeftButtonPressed.y;
```
Appendix B: Code excerpts from the Graphics Drawing Application

//**********************************************************
//  Gets the starting point for the pie.
//  
Pie:
arc_Start_x = LeftButtonPressed.x;
arc_Start_y = LeftButtonPressed.y;

//**********************************************************
//  Gets the starting point for the
//  polygon and adds the point to
//  the point list.
//  
Poly:
    AddPoint(LeftButtonPressed.x, LeftButtonPressed.y);
poly_x = LeftButtonPressed.x;
poly_y = LeftButtonPressed.y;

//**********************************************************
//  Gets the colour of the current point and set
//  background or foreground to that colour.
//  
PicCol:
    Colour tempCol;
    out(DeskTop).GetPixel(
      gid, (mWin),
      LeftButtonPressed.x, LeftButtonPressed.y,
      &tempCol.red, &tempCol.green, &tempCol.blue);
    fgColour = tempCol;
    out(DeskTop).SetDrawColour(
      gid, (mWin),
      fgColour, bkColour);
    {foreCol}.bkColour = fgColour;
    locked=false;
}

//**********************************************************
//  When this event is received firstly
//  we check that the point is in the drawing area,
//  then we set the current point to the tooltip box
//  and we check if we have allocate the graphics context.
//  
(true) MouseMove [
    bool check = PtInRect(
      xarea+1,
yarea+1,
warea-2,
harea-2,
MouseMove.x,
MouseMove.y);

Page 94
Development Environments for Universal Access, TR_319 2003, ICS-FORTH
if (check==true)
    mposx = MouseMove.x - xarea;
    mposy = MouseMove.y - yarea;
    {xycoord}.label = mposx","+mposy;
if (locked) {
    case (currMode) {
        //*****************************************************************************
        //  Draws the rubber band for the rectangle.
        //*****************************************************************************
        Rect: {
            int oldmode;
            if (pmouse==true) {
                out(DeskTop).Rectangle(  
                    gid,  
                    {mWin},  
                    rect_x,  
                    rect_y,  
                    pmouse_x,  
                    pmouse_y, NoFilled);
            }  
            out(DeskTop).Rectangle(  
                gid,  
                {mWin},  
                rect_x,  
                rect_y,  
                MouseMove.x,  
                MouseMove.y,  
                NoFilled);
            if (pmouse==false) {
                pmouse = true;
                out(DeskTop).SetDrawMode(  
                    gid,  
                    {mWin},  
                    DrawNotXor,  
                    &oldmode);
            }
            out(DeskTop).SetDrawColour(  
                gid,  
                {mWin},  
                fg,white);
            out(DeskTop).SetLineWidth(  
                gid,  
                {mWin},  
                1);
            out(DeskTop).SetLineStyle(  
                gid,  
                {mWin},  
                Dot);
            prevMode = GetPrevMode(  
                oldmode);
            pmouse_x = MouseMove.x;
            pmouse_y = MouseMove.y;
            int w = MouseMove.x - rect_x;
            int h = MouseMove.y - rect_y;
            {whinfo}.label = w*"x"+h;
        }
        . . .
        //*****************************************************************************
        //  Draws the rubber band for the line.
        //*****************************************************************************
        Line: {
            int oldmode;
            if (pmouse==true) {
                out(DeskTop).DrawLine(  
                    gid,  
                    {mWin},  
                    line_x,  
                    line_y,  
                    pmouse_x,  
                    pmouse_y,  
                    pmouse_z,  
                    pmouse_d,  
                    &oldmode);
            }
            out(DeskTop).SetDrawColour(  
                gid,  
                {mWin},  
                fg,white);
            out(DeskTop).SetLineWidth(  
                gid,  
                {mWin},  
                1);
            out(DeskTop).SetLineStyle(  
                gid,  
                {mWin},  
                Dot);
            pmouse_x = MouseMove.x;
            pmouse_y = MouseMove.y;
            pmouse_z = MouseMove.z;
            pmouse_d = MouseMove.d;
            prevMode = GetPrevMode(  
                oldmode);
        }
    }
}
pmouse_y);
]
out(DeskTop).DrawLine(
gid,
 {mWin},
 line_x,
 line_y,
 MouseMove.x,
 MouseMove.y);

if (pmouse==false){
    pmouse = true;
    out(DeskTop).SetDrawMode(
gid,
 {mWin},
 DrawNotXor,
 &oldmode);
    out(DeskTop).SetDrawColour(
gid,
 {mWin},
 fg,white);
    out(DeskTop).SetLineWidth(
gid,
 {mWin},
 1);
    out(DeskTop).SetLineStyle(
gid,
 {mWin},
 Dot);
    prevMode = GetPrevMode(
 oldmode);
    pmouse_x = MouseMove.x;
    pmouse_y = MouseMove.y;
}

.
.

} else {
   {xycoord}.label = "";
   case (currMode)[
     Pen, Brush, Eraser: locked = false;
     ]
]
.
.

////////////////////////////////////////////////////////////
//  Methods

//**********************************************************
// This method is called when the line button is selected
// and allows user to draw a line.
method {line}.StateChanged {
  {whinfo}.label = "";
  if ({line}.buttonState==ButtonOn) {
    lastMode = currMode = Line;
    {tooltip}.label = "Draws a line with the selected
    colour and line width.";
  } else currMode=lastMode=Empty;
}

//**********************************************************
// This method is called when the curve button is selected
// and allows user to draw a curve.
method {curve}.StateChanged {
Appendix B: Code excerpts from the Graphics Drawing Application

```java
{whinfo}.label = "";
if (({curve}.buttonState==ButtonOn) [  
    lastMode = currMode = Curve;  
    {tooltip}.label = "Draws a curved line with the  
        selected colour and line width.";
]  
else currMode=lastMode=Empty;
}

/****************************************** **
// This method is called when the Flood fill button is selected  
// and allows user to fill an area with the selected colour.
method {ffill}.StateChanged[  
    {whinfo}.label = "";
    if ({ffill}.buttonState==ButtonOn) [  
        lastMode = currMode = Flood;  
        {tooltip}.label = "Fills an area with the  
            current drawing colour.";
    ]  
else currMode=lastMode=Empty;
}

/****************************************** **
// This method is called when the eraser button is selected  
// and allows user to erase a draw.
method {eraser}.StateChanged [ 
    {whinfo}.label = "";
    if (({eraser}.buttonState==ButtonOn) [  
        lastMode = currMode = Eraser;  
        {tooltip}.label = "Erases a portion of the picture.";
    ]  
else currMode=lastMode=Empty;
}

/****************************************** **
// This method is called when the rect button is selected  
// and allows user to draw a rectangle.
method {rect}.StateChanged [ 
    {whinfo}.label = "";
    if (({rect}.buttonState==ButtonOn) [  
        lastMode = currMode = Rect;  
        {tooltip}.label = "Draws a rectangle with the  
            selected colour and line width.";
    ]  
else currMode=Empty;
}

/****************************************** **
// This method is called when the ellipse button is selected  
// and allows user to draw an ellipse.
method {ellipse}.StateChanged [ 
    {whinfo}.label = "";
    if (({ellipse}.buttonState==ButtonOn) [  
        lastMode = currMode = Ellipse;  
        {tooltip}.label = "Draws an ellipse with the  
            selected colour and line width.";
    ]  
else currMode=lastMode=Empty;
]

```

//**********************************************************
// This method is called when the user selects a menu item.  
// By selecting the first item the user creates a new picture and  
// by selecting the second exits the application.

Page 97
Development Environments for Universal Access, TR_319 2003, ICS-FORTH
method {menu}.Selected [ 
  if ({menu}.selectedItemID==4)[
    terminate;
  ]
  else if ({menu}.selectedItemID==0)[
    {whinfo}.label = "";
    NewPic();
    out(DeskTop).WriteBitmap{
      {mWin},
      xarea+1,
      yarea+1,
      area-xarea-3,
      harea-yarea-3,
      picfile,
      &gbid};
  ]
]

/**********************************************************
// This method is called when the user selects an item
// from the pen style listbox and sets the new style of
// the pen.
/**********************************************************
method {cb}.Selected [ 
  {whinfo}.label = "";
  if ({cb}.subjectIndex == 0)[
    lstyle = Solid;
    if (gid!=-1) out(DeskTop).SetLineStyle(gid,{mWin},Solid);
  ]
  else if ({cb}.subjectIndex == 1)[
    lstyle = Dash;
    if (gid!=-1) out(DeskTop).SetLineStyle(gid,{mWin},Dash);
  ]
  else if ({cb}.subjectIndex == 2)[
    lstyle = Dot;
    if (gid!=-1) out(DeskTop).SetLineStyle(gid,{mWin},Dot);
  ]
  else if ({cb}.subjectIndex == 3)[
    lstyle = DashDot;
    if (gid!=-1) out(DeskTop).SetLineStyle(gid,{mWin},DashDot);
  ]
  else if ({cb}.subjectIndex == 4)[
    lstyle = DashDotDot;
    if (gid!=-1) out(DeskTop).SetLineStyle(gid,{mWin},DashDotDot);
  ]
]

/**********************************************************
// Attribute assignment, graphics context allocation
// and bitmap initialization
/**********************************************************
constructor [ 
  printstr("pass\n");
  AllocateGraphicsCheck();
  LoadBitmaps();
  {mWin}.x = 0;
  {mWin}.y = 0;
  {mWin}.width = 900;
  {mWin}.height = 680;
  {mWin}.bkColour = bg;
  {mWin}.title="I-GET Draw";
  {mWin}.status = WinNormal;
  {mWin}.hasMinimizeBox = true;
  {mWin}.hasMaximizeBox = true;
  {mWin}.hasSystemMenu = true;
  {tools}.x = 5;
  {tools}.y = 5;
  {tools}.width = 70;
Appendix B: Code excerpts from the Graphics Drawing Application

```plaintext
{tools}.height = 300;
{tools}.isVisible = false;
{tools}.boxColour=white;
{tools}->Set(7,2);
{tools}->SetHorzBorder(2);
{tools}->SetVertBorder(2);

{styles}.x = 800;
{styles}.y = 5;
{styles}.width = 100;
{styles}.height = 140;
{styles}.isVisible = false;
{styles}.boxColour=white;
{styles}->Set(8,1);
{styles}->SetHorzBorder(2);
{styles}->SetVertBorder(2);

{bcols}.x = 90;
{bcols}.y = 550;
{bcols}.width = 210;
{bcols}.height = 60;
{bcols}.isVisible = false;
{bcols}.boxColour=white;
{bcols}->Set(2,10);
{bcols}->SetHorzBorder(2);
{bcols}->SetVertBorder(2);

{coords}.x = 330;
{coords}.y = 535;
{coords}.width = 210;
{coords}.height = 60;
{coords}.isVisible = false;
{coords}.boxColour=white;
{coords}->Set(1,3);
{coords}->SetHorzBorder(2);
{coords}->SetVertBorder(2);
```

... 

```plaintext
{polygon}.width = 30;
{polygon}.height = 30;
{polygon}.isVisible=true;
{polygon}.hasText=false;
{polygon}.hasBitmap=true;
{polygon}.textPosition = OnBitmap;
{polygon}.bkColour = bg;
{polygon}.textColour = fg;
{polygon}.font = font;
{polygon}->SetBitmap(bid10);
{polygon}->SetBitmapSize(20,20);

{arc}.width = 30;
{arc}.height = 30;
{arc}.isVisible=true;
{arc}.hasText=false;
{arc}.hasBitmap=true;
{arc}.textPosition = OnBitmap;
{arc}.bkColour = bg;
{arc}.textColour = fg;
{arc}.font = font;
{arc}->SetBitmap(bid11);
{arc}->SetBitmapSize(20,20);

{pie}.width = 30;
{pie}.height = 30;
{pie}.isVisible=true;
{pie}.hasText=false;
{pie}.hasBitmap=true;
{pie}.textPosition = OnBitmap;
{pie}.bkColour = bg;
{pie}.textColour = fg;
```
Appendix B: Code excerpts from the Graphics Drawing Application

```forth
{pie}.font = font;
{pie}->SetBitmap(bid12);
{pie}->SetBitmapSize(20,20);

{tools}->Add({dtext});
{tools}->Add({piccol});
{tools}->Add({eraser});
{tools}->Add({ffill});
{tools}->Add({pen});
{tools}->Add({brush});
{tools}->Add({line});
{tools}->Add({curve});
{tools}->Add({rect});
{tools}->Add({ellipse});
{tools}->Add({roundRect});
{tools}->Add({polygon});
{tools}->Add({arc});
{tools}->Add({pie});

...

{tools}->Realize();
{styles}->Realize();
{coords}->Realize();

out(DeskTop).RealizeObject({mWin});
out(DeskTop).SetDrawColour(gid,{mWin},spColour,bkColour);
out(DeskTop).Rectangle(gid,
{mWin},
xarea,
yarea,
warea-1,
harea-1,
BorderFilled);
out(DeskTop).SetDrawColour(gid,{mWin},fgColour,bkColour);
}

destructor []
```
APPENDIX C

CODE EXCERPTS FROM THE GEOMETRICAL CONSTRAINTS APPLICATION

//----------------------------------------------------------
// Geometrical Constraints Application.
//----------------------------------------------------------
// I-Get Draw header files.
//@30
headers{
    extern ["igetwin.hh","igetlinerect.hh"]
    #include "win.gdd"
    #include "linerect.hdr"
}

//----------------------------------------------------------
// Rectangles MACROS
/* The width of the outer rectangle */
/* The height of the outer rectangle */
/* width and height of the inner rectangles*/
/* Top Left rectangle coordinates */
/* Bottom Left rectangle coordinates */
/* Top Right rectangle coordinates */
/* Bottom Right rectangle coordinates */

#define FACTOR 1/5    
#define WIDTH_CONSTRAINT(_w,_r)  
    _w :=({_r}.x2 - {_r}.x1); 
#define HEIGHT_CONSTRAINT(_h,_r)  
    _h :=({_r}.y2 - {_r}.y1); 
#define FACTOR_CONSTRAINT(_hw, _vw, _w, _h) 
    _hw := _w * FACTOR; \ 
    _vw := _h * FACTOR; 
#define TOP_LEFT_CONSTRAINT(_r1, _r2, _hw, _vw) 
    { _r2}.x1 := _r1}.x1 + _hw; \ 
    { _r2}.y1 := _r1}.y1 + _vw; \ 
    { _r2}.x2 := _r1}.x1 + 2*_hw; \ 
    { _r2}.y2 := _r1}.y1 + 2*_vw; 
#define BOTTOM_LEFT_CONSTRAINT(_r1, _r2, _hw, _vw) 
    { _r2}.x1 := _r1}.x1 + _hw; \ 
    { _r2}.y1 := _r1}.y1 + 3*_vw; \ 
    { _r2}.x2 := _r1}.x1 + 2*_hw; \ 
    { _r2}.y2 := _r1}.y1 + 4*_vw; 
#define TOP_RIGHT_CONSTRAINT(_r1, _r2, _hw, _vw) 
    { _r2}.x1 := _r1}.x1 + 3*_hw; \ 
    { _r2}.y1 := _r1}.y1 + _vw; \ 
    { _r2}.x2 := _r1}.x1 + 2*_hw; \ 
    { _r2}.y2 := _r1}.y1 + 4*_vw;
Appendix C: Code excerpts from the Geometrical Constraints Application

\[
\begin{align*}
(_r2).x2 &= (_r1).x1 + 4*_hw; \\
(_r2).y2 &= (_r1).y1 + 2*_vw;
\end{align*}
\]

#define BOTTOM_RIGHT_CONSTRAINT(_r1, _r2, _hw, _vw) \
\{ 
(_r2).x1 := (_r1).x1 + 3*_hw;
(_r2).y1 := (_r1).y1 + 3*_vw;
(_r2).x2 := (_r1).x1 + 4*_hw;
(_r2).y2 := (_r1).y1 + 4*_vw;
\}

//**********************************************************
//  Diamond Macros
/* End of a line is the start of the other */
/* The middle point of a line is the start of a line and the end of another */
#define START_END_CONSTRAINT(_l1,_l2) \
\{ 
(_l1).x2:={_l1}.x2; 
(_l2).x1:={_l1}.x2; 
(_l1).y2:={_l2}.y1; 
(_l2).y1:={_l1}.y2;
\}

#define MID_POINT_CONSTRAINT(_la, _lb) \
\{ 
(_lb).x1:=({_l}.x1 + {_l}.x2)/2; 
(_lb).y1:=({_l}.y1 + {_l}.y2)/2; 
(_la).x2:=({_l}.x1 + {_l}.x2)/2; 
(_la).y2:=({_l}.y1 + {_l}.y2)/2;
\}

//**********************************************************
//  Global Attributes
local Colour white = [255,255,255];
local int Hw;
local int Vw;
local int height;
local int width;
local bool linetest=false;
local bool recttest=false;
local bool diamtest=false;
local Font font= ["Arial", 12, false, false, false, false, 0, 0 ];

//**********************************************************
//  Agent for the Diamond Test
agent LineTest create if (diamtest ==true) [ 
lexical (DeskTop) FrameWindow win;
lexical (DeskTop) Line l1:parent={win};
lexical (DeskTop) Line l2:parent={win};
lexical (DeskTop) Line l3:parent={win};
lexical (DeskTop) Line l4:parent={win};
lexical (DeskTop) Line l5:parent={win};
lexical (DeskTop) Line l6:parent={win};
lexical (DeskTop) Line l7:parent={win};
lexical (DeskTop) Line l8:parent={win};
START_END_CONSTRAINT(l1,l2)
START_END_CONSTRAINT(l2,l3)
START_END_CONSTRAINT(l3,l4)
START_END_CONSTRAINT(l4,l1)
]

Page 102
Development Environments for Universal Access, TR_319 2003, ICS-FORTH
Appendix C: Code excerpts from the Geometrical Constraints Application

MID_POINT_CONSTRAINT(l1,l5,l6)
MID_POINT_CONSTRAINT(l2,l6,l7)
MID_POINT_CONSTRAINT(l3,l7,l8)
MID_POINT_CONSTRAINT(l4,l8,l5)

constructor[
  {win}.x = 250;
  {win}.y = 5;
  {win}.width = 250;
  {win}.height = 250;
  {win}.bkColour = white;
  {win}.title = "Lines";
  {win}.status = WinNormal;
  {win}.hasMinimizeBox = false;
  {win}.hasMaximizeBox = false;
  {win}.hasSystemMenu = true;

  {l1}.x1={l1}.y1 = 10;
  {l2}.x1 = 10;
  {l2}.y1 = 50;
  {l3}.x1={l3}.y1=50;
  {l4}.x1 = 50;
  {l4}.y1 = 10;
  {l5}.access = {l6}.access = {l7}.access = {l8}.access = false;
	not(DeskTop).RealizeObject({win});
]

destructor[ diamtest =false;
]

//**********************************************************
//  Agent for the Rectange Test
agent RectTest create if (recttest==true) [

  lexical (DeskTop) FrameWindow win;

  lexical (DeskTop) Rectangle r1:parent={win};
  lexical (DeskTop) Rectangle r2:parent={win};
  lexical (DeskTop) Rectangle r3:parent={win};
  lexical (DeskTop) Rectangle r4:parent={win};
  lexical (DeskTop) Rectangle r5:parent={win};

  WIDTH_CONSTRAINT(width, r1)
  HEIGHT_CONSTRAINT(height, r1)

  FACTOR_CONSTRAINT(Hw, Vw, width, height)

  TOP_LEFT_CONSTRAINT(r1,r2,Hw,Vw)
  TOP_RIGHT_CONSTRAINT(r1,r3,Hw,Vw)

  BOTTOM_LEFT_CONSTRAINT(r1,r4,Hw,Vw)
  BOTTOM_RIGHT_CONSTRAINT(r1,r5,Hw,Vw)

constructor[
  {win}.x = 250;
  {win}.y = 300;
  {win}.width = 250;
  {win}.height = 250;
  {win}.bkColour = white;
  {win}.title = "Rectangles";
  {win}.status = WinNormal;
  {win}.hasMinimizeBox = false;
  {win}.hasMaximizeBox = false;
  {win}.hasSystemMenu = true;

  {r1}.x1={r1}.y1 = 10;
  {r1}.x2= 150;
  {r1}.y2 = 100;
  {r2}.access = {r3}.access = {r4}.access = {r5}.access = false;
]
Appendix C: Code excerpts from the Geometrical Constraints Application

```
out(DeskTop).RealizeObject({win});
}
destructor[recttest=false;]

/**********************************************************
//  Main Panel
agent Main create if (true) {
    lexical (DeskTop) FrameWindow win;
    lexical (DeskTop) Button rbutton:parent={win};
    lexical (DeskTop) Button lbutton:parent={win};
    lexical (DeskTop) Button mbutton:parent={win};
    lexical (DeskTop) Button quit:parent={win};

    method {rbutton}.Pressed [
        recttest = true;
    ]

    method {lbutton}.Pressed [
        diamtest = true;
    ]

    method {mbutton}.Pressed [
        linetest = true;
    ]

    method {quit}.Pressed [
        terminate;
    ]

    constructor{
        {win}.x = 5;
        {win}.y = 5;
        {win}.width = 170;
        {win}.height = 200;
        {win}.bkColour = white;
        {win}.title = "Lines & Rectangles";
        {win}.status = WinNormal;
        {win}.hasMinimizeBox = false;
        {win}.hasMaximizeBox = false;
        {win}.hasSystemMenu = true;

        {rbutton}.x = {rbutton}.y = 10;
        {rbutton}.width = 60;
        {rbutton}.height = 40;
        {rbutton}.font = font;
        {rbutton}.isVisible=true;
        {rbutton}.label = "Rectangles";

        {lbutton}.x = 80;
        {lbutton}.y = 10;
        {lbutton}.width = 60;
        {lbutton}.height = 40;
        {lbutton}.font = font;
        {lbutton}.isVisible=true;
        {lbutton}.label = "Lines";

        {quit}.x = 50;
        {quit}.y = 100;
        {quit}.width = 60;
        {quit}.height = 40;
        {quit}.font = font;
        {quit}.isVisible=true;
        {quit}.label = "Quit";
    }

destructor[]
}
```
REFERENCES


