
# AFRL-RH-WP-TR-2010-0115

# USER INTERFACE DESIGN PATTERNS

**Vincent A. Schmidt**

**Warfighter Interface Division**

**July 2010**
**Final Report**



**AIR FORCE RESEARCH LABORATORY**
**711 HUMAN PERFORMANCE WING**
**HUMAN EFFECTIVENESS DIRECTORATE,**
**WRIGHT-PATTERSON AIR FORCE BASE, OH 45433**
**AIR FORCE MATERIEL COMMAND**
**UNITED STATES AIR FORCE**

**NOTICE AND SIGNATURE PAGE**

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th Air Base Wing Public Affairs Office and is available to the general public, including foreign nationals.

Qualified requestors may obtain copies of this report from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RH-WP-TR-2010-0115 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

**FOR THE DIRECTOR**

**//signed//**
Vincent Schmidt
Program Manager
Battlespace Visualization Branch

**//signed//**
Jeffrey L. Craig
Chief, Battlespace Visualizaiton Branch
Warfighter Interface Division

**//signed//**
Michael A. Stropki
Chief, Warfighter Interface Division
Human Effectiveness Directorate
711 Human Performance Wing

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

# REPORT DOCUMENTATION PAGE

*Form Approved*
**OMB No. 0704-0188**



| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| 23-07-2010 | Final Technical Report | 8-08-2008 – 31-03-2010 |

**4. TITLE AND SUBTITLE**
User Interface Design Patterns

**5a. CONTRACT NUMBER**
In-House

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**
62202F

**6. AUTHOR(S)**
Vincent A. Schmidt

**5d. PROJECT NUMBER**
7184

**5e. TASK NUMBER**
10

**5f. WORK UNIT NUMBER**
71841012

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Air Force Materiel Command
Air Force Research Laboratory
711th Human Performance Wing
Human Effectiveness Directorate
Warfighter Interface Division
Battlespace Visualization Branch
Wright-Patterson Air Force Base, OH 45433

**10. SPONSOR/MONITOR'S ACRONYM(S)**
711 HPW/RHCV

**11. SPONSORING/MONITORING AGENCY REPORT NUMBER**
AFRL-RH-WP-TR-2010-0115

**12. DISTRIBUTION AVAILABILITY STATEMENT**
Approved for public release; distribution is unlimited.

**13. SUPPLEMENTARY NOTES**
88 ABW Cleared 10/04/10; 88ABW-2010-5331.

**14. ABSTRACT**
Graphical User Interface Design Patterns (UIDP) are templates representing commonly used graphical visualizations for addressing certain HCI issues. These patterns include substantial contributions from human factors professionals, and using these patterns as widgets within the context of a GUI builder helps to ensure that key human factors concepts are quickly and correctly implemented within the code of advanced visual user interfaces. This final report describes the concept of the UIDP and discusses how this concept can be implemented to benefit both the programmer and the end user by assisting in the fast generation of error-free code that integrates human factors principles to fully support the end-user's work environment.

**15. SUBJECT TERMS**
User Interface Design Patterns, GUI, HCI, Widget, Software Design

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Vincent Schmidt |
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | SAR | 22 | 19b. TELEPHONE NUMBER (*Include area code*) (937) 255-8363 |

THIS PAGE IS INTENTIONALLY LEFT BLANK

## TABLE OF CONTENTS

## LIST OF FIGURES

## 1.0 SUMMARY

During the development of a software system, something is lost in translation of the Human-Computer Interface (HCI) between the human factors engineer's analysis and the software developer's implementation. Since the developer touches the product last, part of the human factors engineer's contribution is frequently lost. Graphical User Interface Design Patterns (UIDP) are templates representing commonly used graphical visualizations for addressing certain HCI issues. These patterns include substantial contributions from human factors professionals, and using these patterns as widgets within the context of a GUI builder helps to ensure that key human factors concepts are quickly and correctly implemented within the code of advanced visual user interfaces. This final report describes the concept of the UIDP and discusses how this concept can be implemented to benefit both the programmer and the end user by assisting in the fast generation of error-free code that integrates human factors principles to fully support the end-user's work environment.

## 2.0 INTRODUCTION

During the development of a software system, something is often lost in translation of the Human-Computer Interface (HCI) between the human factors engineer's analysis and the software developer's implementation. Since the developer touches the product last, part of the human factors engineer's contribution may not make it into the final product. This unfortunate circumstance can degrade a software system's ability to effectively support the work for which it was originally designed. In addition to formal specifications for software design, which are already used in practice, it would be beneficial if software designers and implementers could rely on the availability of a collection of prewritten graphical elements (design patterns) that meet human factors design constraints. This would eliminate much of the miscommunication that leads to poor implementation choices when designing specific software user interfaces.

This research concentrates on defining and implementing Graphical User Interface Design Patterns (UIDPs). These are templates representing commonly used graphical visualizations for addressing certain work requirements. HCI issues are addressed within the patterns, with substantial contributions from human factors professionals. Using these patterns as widgets within the context of a GUI builder helps to ensure that key human factors concepts are quickly and correctly implemented within the code of advanced visual user interfaces. More specifically, this research explores the concept of the UIDP and describes how this concept can be implemented to benefit both the programmer and the end user by assisting in the fast generation of error-free code that integrates human factors principles to fully support the end-user's work environment.

Although UIDPs would be useful across a wide variety of software domains, the need for UIDPs within the United States Air Force is drawn from the requirement to rapidly implement and deploy software systems supporting command and control (C2) and similar applications. Such applications may also include net-centric components and interfaces to intelligence and surveillance systems.

The UIDP research investigated within this effort has foundations within the work-centered support system concepts originally defined in AFRL's Cognitive Systems branch AFRL/HECS), and developed as a part of the WIDE program. The requirements and objectives of the UIDP research can be summarized as:

- Customer Requirements
  - Reduced resources required to develop user interfaces for work-centered support environments

- Objectives – Develop reusable, work-centered User Interface (UI) components for complex environments such as C4ISR and Homeland Defense, resulting in:
  - Improved collaboration among operators, designers, and other stakeholders during UI design process
  - Maturation of work-centered design philosophy into a design science
  - Increased adoption of work-centered design approach across DoD and Homeland Defense agencies

This final report describes UIDP concepts, discusses the steps that would be required to implement them, and identifies key resources used to investigate these ideas. Much of the description comes from the published conference paper, included as a product of this work unit.

## 3.0 DESCRIPTION

Human factors specialists (HFSs) focus on, among other things, the design of excellent user interfaces targeted towards the end user. The traditional approach to building computing systems is for the HFS to generate conceptual interfaces, and then pass these designs along to the programmers for implementation. Unfortunately, inadequate specification and gross miscommunication is the norm, so the well-designed interfaces are frequently generated in such a way that many of the elements intended to contribute to a well-designed interface are left out of the final implementation. The result is a new software system that does not optimally support the end-user's work environment.

When this occurs, not only is the resultant software sometimes tedious or difficult to use, but it may not support the work requirements. This may bring about a series of unfortunate and unintended effects: the retirement of the newly developed system in favor of the design and development of yet another solution. Clearly, this parade of events is a waste of time and money for the developers as well as the customers.

Another facet of software design is the tendency to use existing off-the-shelf tools to provide C2 and C4ISR solutions. While this approach may initially be low in cost and risk, it is rarely a good fit for the actual work. Even if such solutions begin as a useful tools, they tend to grow in size and complexity until they no longer meet the work requirements, and are neither easy to use or maintain.

Custom software designs allow good human factors and cognitive systems engineering to be applied, when available, which could result in an excellent fit to the work (when well-

implemented). The principal drawbacks to custom design are the large expense, in dollars and in short-term risk, of these designs, and the potential complexity of the design of the new system.

The UIDP approach encourages the software designer to reuse carefully crafted graphical patterns to rapidly generate and deploy software that is lower in risk and approach to custom software, yet still presents a tailored fit to the work. With this approach, the monetary cost, development time, and software complexity are all substantially reduced. These techniques are shown in Figure 1.

The most practical solution for using UIDPs is to have a UIDP library that goes beyond the mere description or specification of graphical user interface (GUI) patterns. With the exception of abstract data type (ADT) libraries, most design pattern work currently ends with the description of the patterns and small snippets of sample code. This leaves implementation details up to the programmer, introducing the strong potential for misrepresentation of the final GUI elements. A UIDP library would include templates that can be parameterized to generate a "90% solution" to specific GUI elements in the design. Each one of these templates would be vetted by HFS experts, and their use would guarantee that human factors components are correctly represented graphically to the end user.

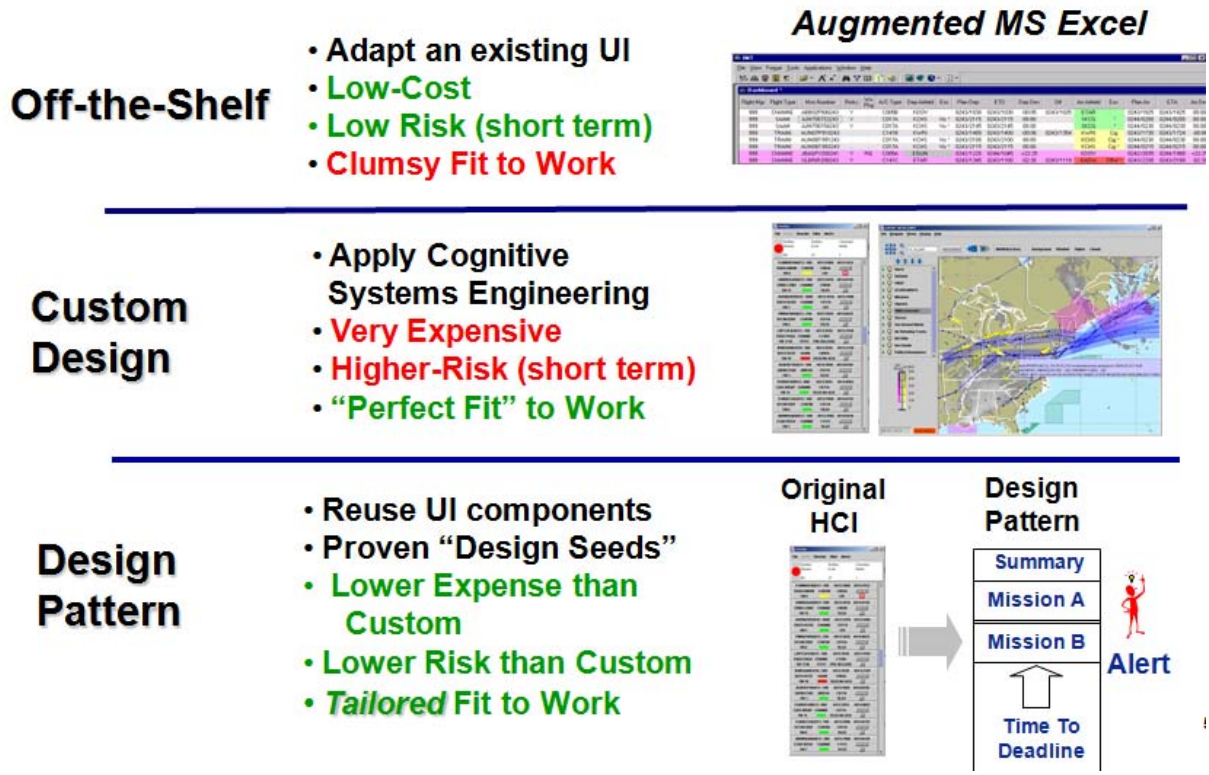


**Figure 1: Software Development Approaches**

Our contribution to this solution is to provide these library components as "widgets" in a popular drag-and-drop GUI builder. Programmers could drag these human-factored UIDP elements into the applications being built, and an integrated wizard would guide the programmer though a series of tailored queries to parameterize the template. The code generated by the GUI builder would include not only the code necessary to design the GUI's standard widgets (pull-down menus and button elements), but also code to generate the specialized UIDP elements. This will result in an application where human factors designs are not lost due to implementation miscommunications. Furthermore, the introduction of human factors into the standardization of design patterns encourages coders to use these pre-designed elements, since these elements will be more user-friendly and less prone to coding error. This also achieves a reasonable amount of software reuse.

## 4.0 ARCHITECTURE

The full UIDP concept includes the deployment of a collection of design patterns to be specified and documented in a design pattern library, along with one or more corresponding software components implementing the patterns. Software developers and human factors experts should continue to examine both new and existing applications in order to identify new patterns to add to the library, and to generalize or refactor existing pattern implementations. The intention is that the repository be fully accessible online, available as a source of components for software reuse by designers and developers. Figure 2 demonstrates these ideas, with the philosophy clearly depicted: developers use the library, but also feed the repository with new and updated patterns and pattern concepts throughout the life cycle of the products being developed. This cycle continuously feeds and grows the UIDP repository.

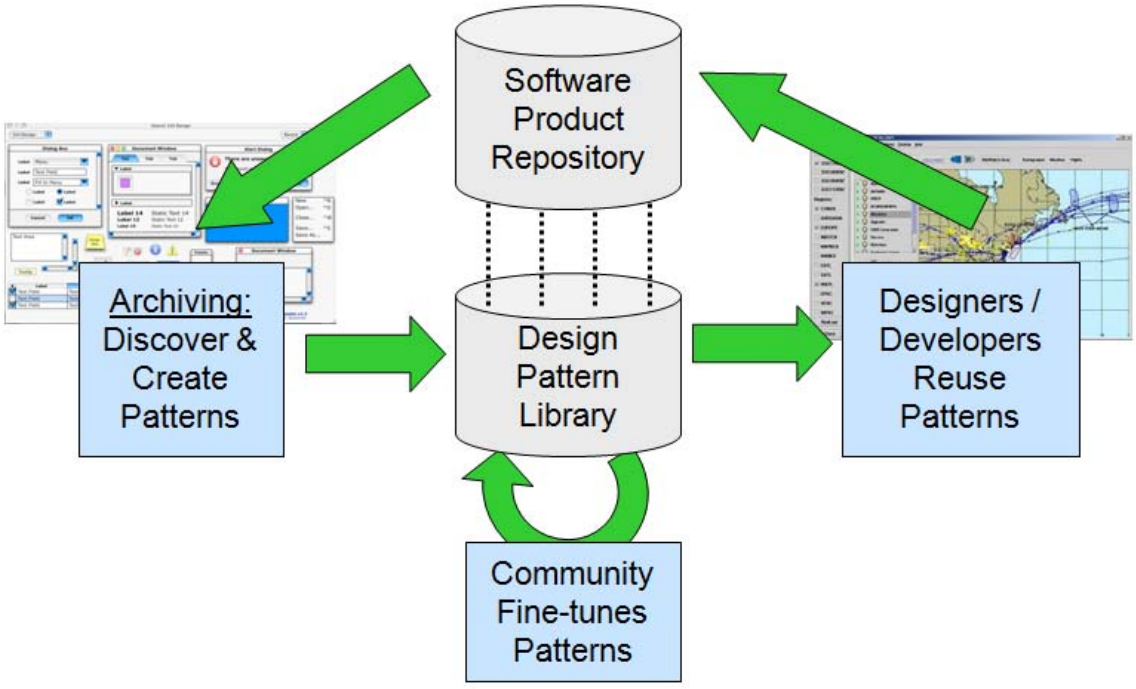


**Figure 2: User Interface Design Pattern Concept**

For this research, the details of the repository and pattern library deployment are left as a point of discussion as we concentrate on the implementation and usage of a very small collection of specific patterns. The remaining discussion in this section of the report outlines our thought process as we continue to develop and realize components of specific UIDPs for demonstration and proof-of-concept purposes.

Our goal is to demonstrate how pattern specifications might be implemented for use as UIDP components. In order to be interesting to the widest possible technical audience, the implementation platform decisions are largely guided by our selection of the target programming language (i.e. Java, C++, Python, Perl, etc.). The list of potential languages to be used is narrowed by our needs: we must be able to create visual objects that are easily manipulated by the typical programmer.

Our initial conceptual research efforts focused exclusively on the use of Java, since contemporary programmers are frequently familiar with Java, and because Java code is platform independent, meaning it can be used on any computer architecture with a JVM. This allowed us to have meaningful discussions at a detailed technical level, describing how specific portions of UIDPs would be implemented or used in an integrated software development environment. Ultimately, not much time was invested in the coding of an actual Java implementation, and as our philosophy progressed we changed the specific focus of our discussion and methods to use Python. (Python is a high-level powerful cross-platform scripting language with strong object-oriented focus, and is commonly used for rapid software development. See www.python.org.) The selection of Python for a reference implementation was based on its utility for rapid application development, and on the availability of accessible "local" Python software development expertise.

The target of the UIDP implementation is the GUI builder (or similar IDE). Our rationale is that GUI-based software is frequently built using GUI-builder tools, and it seems reasonable to extend these drag-and-drop interfaces to include a collection of "super widgets" that implement the User Interface Design Pattern concepts. The coder would use these UIDP elements within the GUI builder just like the common widget set. Figure 3 illustrates this concept.

An extensible GUI builder is required, since these are visual design patterns that we attempted to create. Extensibility is vital to iteratively build up new pattern widgets using existing widgets as "building blocks." As long as the GUI builder is technologically sound, low GUI builder cost should result in encouraging frequent use within the community at large. Cost is also a major consideration for implementers and end-users: when the development utility is free, the end-user can use the widgets without dipping into their wallets, or filing corporate purchase requisitions.

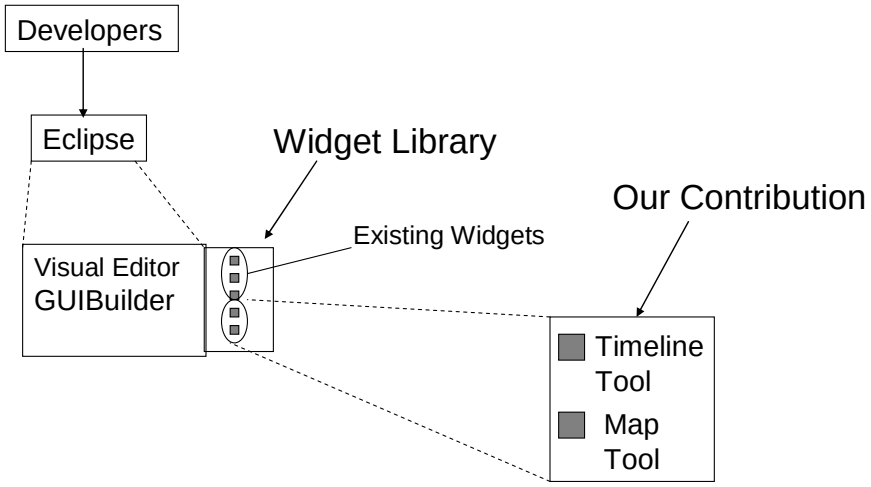


**Figure 3: UIDP Module Within GUI Builder**

A brief survey of GUI builders and IDEs meeting our development requirements (documented at the beginning of our research) led us to Glade (glade.gnome.org), a cross-platform GUI builder platform that saves its descriptive files in XML format. (Major consideration was initially given to Java Netbeans and Java Eclipse, and later extended to Glade.) The saved XML files fully describe Glade-designed user interfaces. Glade libraries are available for numerous programming languages on many computing platforms. This makes the choice of Glade ideal, since the UIDP concept is less constrained when the application developer is not required to work with a particular computer language, operating system, or hardware platform.

The reference implementation concept shows how to use Glade to create UIDP tools as widgets, collected into a loadable module. When the module is loaded, the UIDP elements are placed on a GUI palette along side of (or in the same capacity as) other GUI widgets. Individual UIDPs can be used in a drag and drop fashion, just like the IDE's pre-existing widgets.

The principal difference between using a "standard" widget and a UIDP widget is in the actions that occur when the UIDP is used. When the UIDP element is selected and placed on the display, a wizard will guide the programmer through a series of selections that instantiate the templates based on the desired characteristics. With the addition of a wizard to assist in the creation of new widgets, a fully customized high level pattern could be implemented in a matter of minutes.

The GUI builder will save the UIDP code along with all other generated code. Wizard selections should be saved (as XML, for example) so UIDP instantiation choices can be revised

if needed. This information could be saved as an additional file, or even as comments within the generated code.

Implementation of the UIDP "widget set" within Glade should be module-based, so one or more design patterns would be stored in a single loadable module without requiring a Glade recompile. All widgets within the UIDP module will already be vetted by human factors experts. The expectation is to heavily involve these experts in the design and evaluation of UIDP components. As coders who work closely with human factors engineers, we are somewhat uniquely qualified to produce the UIDP elements for this library in such a way that the instantiated and generated library code results in visualizations with built-in human factors considerations that support the end-user's work environment and needs.

## 5.0 SPECIFIC UIDP CONCEPTS

Our internal discussion of UIDP was based primarily on two (2) conceptual models: the Map pattern and the Timeline pattern. Each is reviewed briefly.

### 5.1 Map Pattern

The Map pattern is useful for displaying spatial information as layers of imagery, symbols, and corresponding text. As the name suggests, this pattern is found primarily in maps and mapping software. Since geographic paper and electronic maps are commonly used, most people are already familiar with the map concept.

In general, maps are frequently used for driving directions and road map scenarios. However, this pattern could also be used in a wide variety of other scenarios: creating a weather map for a new area, mapping underground tunnels, visualizing something as small as a microchip and schematic diagrams, or displaying something as large as the night sky.

The map pattern can be implemented as a widget that is a collection of layered images and corresponding text, along with extra control options and map legend data. The example shown in Figure 4 could easily be created with two layers. The first would be a base layer consisting of a picture of the area. The second would be a drawn image of roads that coincides with the base layer's picture.

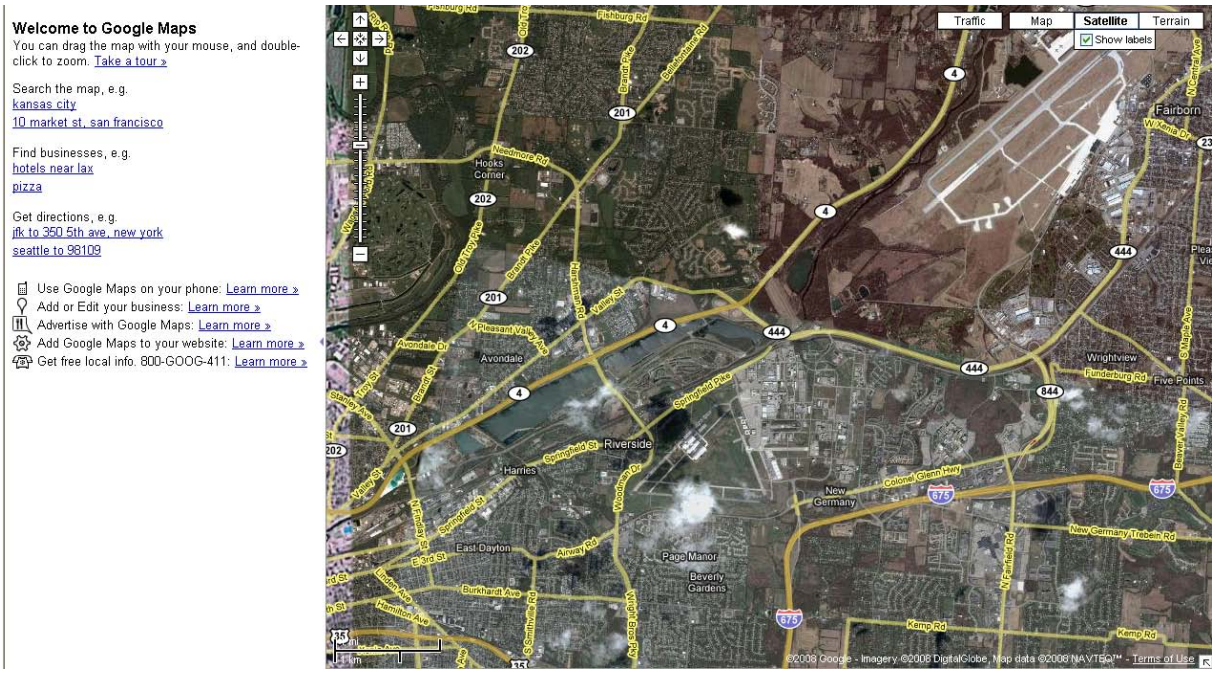


**Figure 4: Map Pattern Example**

Each layer could have a collection of related attributes and controls, such as:

- Image
- Opacity
- Vertical Pan
- Horizontal Pan
- Vertical Wrap
- Horizontal Wrap
- Zoom qualities
- Zoom rate
- Toggle on/off

These characteristics might be explicitly coded into the map implementation by the programmer, or they may be configurable in real time by the user at the user-interface level.

As an additional example, the map pattern could portray a floor plan for an entire multi-floor building (see Figure 5). The pattern is instantiated by adding a layer for each floor and

controlling layer opacity. Clearly, the map is a good pattern because of its ability to be reused in a multitude of situations.

The map pattern could be implemented as a template that is instantiated within a GUI builder. When the programmer adds the map widget to the display, a wizard guides the GUI designer through a series of decisions to fill out the code template. The map's layer concept is represented by providing the GUI designer an opportunity to continue to add new layers until all desired layers have been described. For each layer the user adds, the wizard provides a series of options and attributes that can be selected for this layer, and indicates how the layer relates to other layers. (For example, should opacity be defined by the programmer, or should the end-user be able to change opacity dynamically for this layer? Should this layer be registered to pan and zoom independently, or tied explicitly to the corresponding values of other specified layers?)

The wizard also allows the layers to be prioritized, stacked, and placed as a cohesive unit. While there are only a few attributes, adjusting them can lead to an optimized "map" for a vast number of different uses.

Suppose there is a need to display weather information on a map. By adding a layer containing a dynamic image of a Doppler readout, giving this new layer a high priority (or top location on a visible stack), and changing its opacity to 50%, clouds and precipitation would be shown along with the referenced geographic data. This layer could also be deselected so it is not displayed. Clearly, the layers need to be able to display both static and dynamic information, as well as textual and symbolic data. The map pattern template must provide a mechanism for these displays.

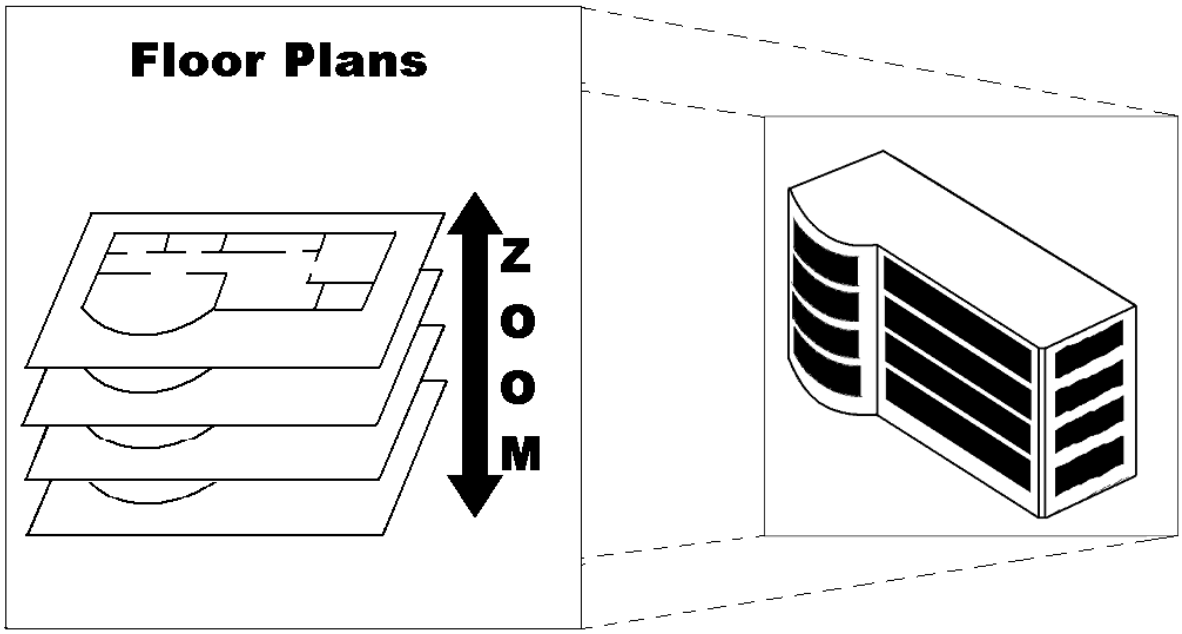


**Figure 5: Map Pattern Depicting Building Floor Plan**

Some characteristics certainly need to be determined by the programmer. The map pattern template should provide code for these decision points and implementing the capabilities required. Navigation and perspective are prime examples. Additional navigation options may be necessary when an image becomes too large for the screen. This is where the pan, zoom, and horizontal and vertical wrap attributes come into play. A programmer may add a slider for measurements that do not pan with the other layers (such as distance or angle overlays and other "heads-up display" information). Similarly, a programmer may require layers to pan at different rates (perhaps to approximate differences in distance between successive layers; a rotatable first person perspective and pan is similar to rotating the point of view, so closer objects will pan and rotate at a different rate than farther objects).

Another common map pattern usage is zoom capability. The programmer will need to decide if the user has a requirement to zoom into the image of layers, or zoom from one layer to another, or both. If the programmer makes interactive blueprints for an office building, then it may be desirable for the zoom function to go up or down one floor, whereas the Google Maps application zooms into the layer images until the image quality cannot be maintained before it switches to another layer.

The map pattern is common and versatile, and its many implementation characteristics make it a clear candidate for incorporation into a user interface design pattern library. While we discuss the map pattern as a two-dimensional pattern, it is unclear at this time if it is good idea to directly extend the map pattern into a 3-D (or other multi-dimensional) pattern, or if such usages should eventually be implemented independently.

### 5.2 Timeline Pattern

Timelines are an integral part of today's professional world, whether project management, corporate planning, military missions, or even just trying to reserve room C-3 down the hall for a conference next Tuesday. This pattern is hidden everywhere in our daily lives as well, and can be seen whenever we look at a calendar or make an entry in our PDAs. At its simplest, a timeline pattern is nothing more than a standard way of visualizing time along with constraints and obligations. Incorporation of human factors considerations into the implementation adds tremendous value to the timeline pattern as a software component.

The timeline pattern shows time linearly using a collection of simple numberline-like graphs: lines and points correspond to important moments or time spans. Dependent timelines can be used in conjunction with each other to account for objects, people, or events that mutually constrain the same time periods. Consider a scenario where you want to set up an office recognition party for Jim during business hours. The facility has three conference rooms suitable for the event, but they are intermittently reserved for meetings. The party is expected to last an hour and a half. Bob and Dave, Jim's best friends, must also be able to attend. Jim, Bob, and Dave all have to oversee the rebooting of their servers at 4 p.m., 1 p.m. and 3 p.m., respectively, on top of their various other daily commitments and meetings.

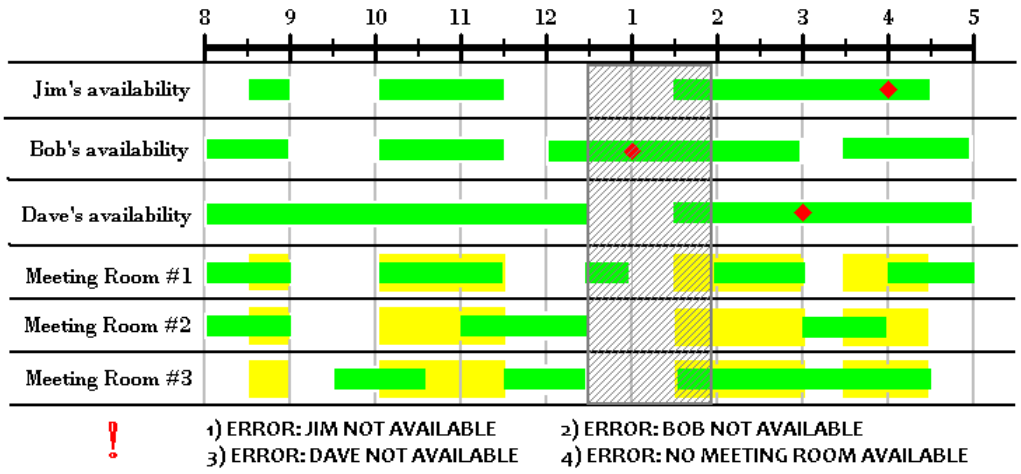


**Figure 6: Timeline Example Depicting Meeting Schedule**

There are several ways someone could deal with scheduling Jim's party with this information. They could try to "think it through" in their head, but this grows increasingly difficult with each new constraint or additional piece of information. The data could be placed in an electronic spreadsheet, but this format is often sloppy, complex, prone to error, and could take a long time to analyze when cross-referencing all the data. This type of data could be easily visualized with a timeline, such as in Figure 6.

The timeline formats all the data into an easy-to-comprehend view that doesn't require manually incorporating constraint cross-references as in an Excel spreadsheet.

The basis of this pattern is a simple linear reference timeline, shown by the notched line labeled from "8" to "5" in Figure 6. This base timeline will act as the point of reference for all related timelines. Each labeled row (shown below the reference timeline) contains a visual representation of the relationships and constraints of a specific activity to the reference timeline. These activities could also be related to (or constrained by) one-another.

Within these rows, there are three separate markers to identify important points or tracts of time:

- **Lines**: These generally signify the primary information of an informative line.
- **Points**: These signify an instantaneous action, restriction, or event.
- **Backgrounds**: These are similar to lines, however they are placed behind lines and point to better use them in conjunction with the other two markers. Backgrounds are generally used to identify an acceptable period of performance for a desired action.

Lines, points and backgrounds can have constraints placed upon or between them among the timeline's rows. In Figure 6, the constraints can easily be seen; all people must be available,

and at least one conference room must be available for an hour and a half period. As more informative lines are added and constraints grow more complicated, it will be necessary to let a computer keep track of the constraints. This is where a Timeline pattern becomes even more useful.

The example in Figure 6 uses Backgrounds, when all three employees are available, to highlight the meeting rooms and aid in the visualization of the constraints of this party. The slider bar can be moved to find the desired time. If the desired time breaks constraints, the slider bar will change color and an error will be given along the bottom of the timeline. A find algorithm could also be included to allow the computer to find acceptable times for the user.

Just as with the Map pattern, the Timeline pattern involves pre-implemented human factors design. By progressing through a wizard, a programmer can easily create a base timeline, informative corresponding rows, and constraints amongst them to match any scheduling or planning needs.

Further aspects of the Timeline allows for tooltips. This enables detailed, relevant, contextual information of a line or object to be displayed by hovering over the object with the mouse. Tooltips provide easy access to contextual information without confusing the user and cluttering the screen when the information is not needed. In addition to the look, feel, and functionality features, a simulation mode can also be easily added. The simulation mode allows the user to graphically move lines and points (time constraints) without changing the actual configuration of the timelines. This capability allows the user to rearrange obligations and experiment with timeline configurations without impacting the actual data. The programmer is responsible for specifying the lines, points, are variables that can be moved, as well defining any restriction upon moving these objects. The simulation mode also implements warning to indicate broken relationship constraints.

A Timeline pattern can be used to represent immense and intricate temporal relationships for extremely complicated scheduling tasks. The United States Air Force intends to use timeline patterns for important coordination of aircraft, flights, and refueling. The Timeline pattern described in this section is a parameterized template based on a custom-coded Timeline Tool application (see Figure 7) being developed within the Air Force Research Laboratory. Our goal is to implement the timeline pattern such that the Timeline Tool could be created using the Timeline pattern wizard.

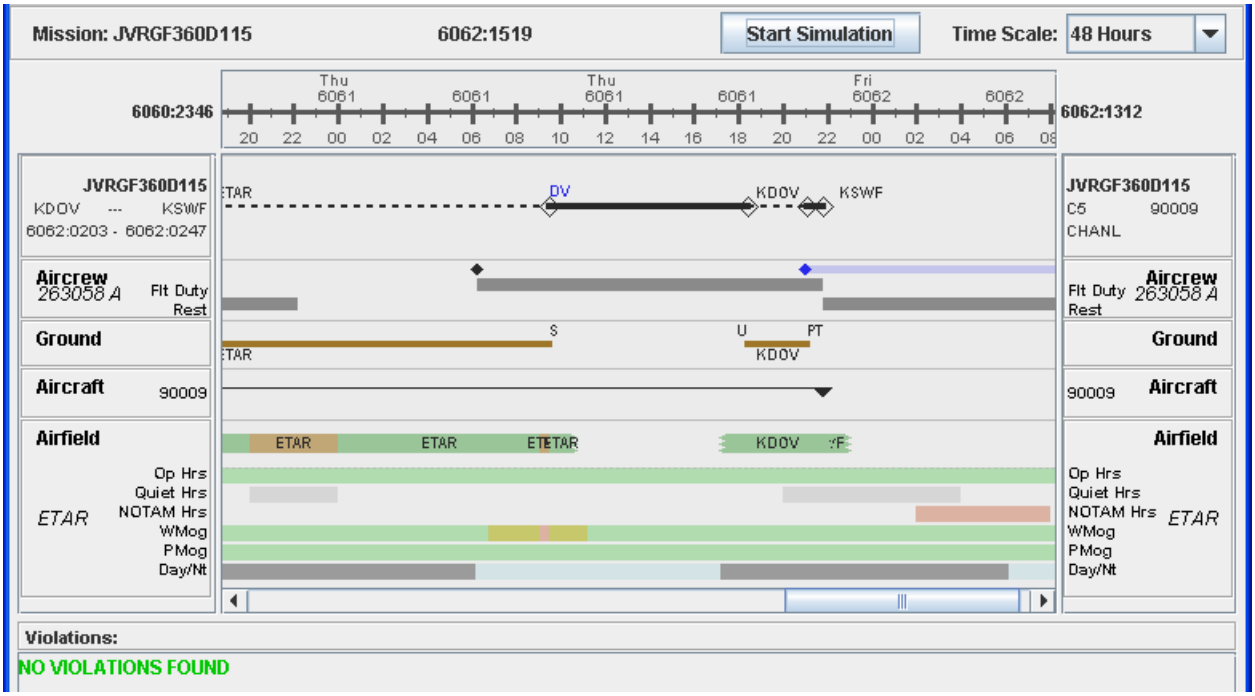


**Figure 7: Timeline Pattern Example**

## 6.0 IMPLEMENTATION

This section describes the fundamental approach to implementing UI Design Patterns as Glade widgets. The computing platform used in this procedure is an Ubuntu-based Linux system, version 9.10 or newer. Be sure to add development (-dev) packages for Glade (3.6.3), gtk (2.12), and glib (2.20). Also add Python 2.x (*not* 3.x, since many Python libraries had not been completely ported to the newly released 3.x versions during the course of this research).

Custom widgets are already supported in Glade, but sparse documentation can make these difficult to install and tedious to configure. For best results, follow the pyclock example from this site to add a fully custom python-based widget:

www.pygtk.org/articles/cairo-pygtk-widgets/cairo-pygtk-widgets.htm (page 1 of 2)
www.pygtk.org/articles/cairo-pygtk-widgets/cairo-pygtk-widgets2.htm (page 2 of 2)

These widgets are written using Cairo for Python (www.cairographics.org) for fine-grained control over the graphic layout. If successful, you should be able to initiate a new Glade session and integrate a Clock "widget" into the Glade display.

Note: you must add the line

__gtype_name__ = 'EggClockFace'

to start of class definition (consider adding it directly above the __gsignals__ line) for glade to recognize the "new" data type.

We recommend you create a small Glade GUI using the Clock widget, then use the UI with a small Python program to ensure your configuration is complete and properly configured. If a Glade GUI with a main window called "window1" is saved as "sample.glade" then a Python program similar to this should be an adequate, if trivial, configuration test:

```
import gtk, pygtk
import pyclock

builder = gtk.Builder()
builder.add_from_file("sample.glade")
win = builder.get_object("window1")
win.show()
gtk.main()
```

If any of the import statements fail, either gtk, pygtk, or the pyclock widget is not installed correctly.

The python widget example, linked above, indicates where to put the files in order for Glade to find and use the new widgets. The .xml file tells Glade about the widget definition, and should be placed in Glade's *catalog* directory. On our development system, this is:

/usr/local/share/glade3/catalogs

The python file (.py source code) is the actual widget definition for the new module, and must be placed in the module directory for Glade. On our development system, this is:

/usr/loca l/lib/glade3/modules

The *modules* directory must also include a library that enables Glade to execute Python code: libgladepython.so. This library should be available for installation though Glade's web site, or through typical Ubuntu package management if it is not already installed.

Although the pyclock example demonstrates how to add a custom widget to Glade, it does not address the UIDP approach. The pyclock widget can be used within Glade by selecting the widget and dropping it onto the palette, but no user-selectable options for the control or appearance of the widget are offered. The UIDP concepts suggest that dropping a "patterned" widget onto the palette (i.e., the map or timeline) would instantiate a construction wizard to guide the user through relevant decisions regarding how the widget would be generated and displayed.

The source code produced under this work unit is incomplete, but work was started on the design of a series of wizard panels for the timeline pattern. An explanation of the intended usage and rough implementation follows.

When the user starts the Glade builder, a palette displays all available widgets in a common area (at the left of the screen in Glade 3.6), including the new UIDP widgets loaded

from the Glade catalog and modules directories. The GUI can be designed via drag and drop operations as desired.

When a UIDP widget is initially dropped onto the GUI, a wizard guides the user though a series of stages to configure the precise operating characteristics of the UIDP widget. These configuration parameters are potentially much more complex than the simple "properties" characteristic of standard widgets, and are used to define both the visible and dynamic components of the UIDP widget, (i.e., for a map UIDP), the decisions may include the number of map layers, layer titles, layer image sources, panning and zooming options, and information regarding how an application user can interact with the widget. A graphical representation of the configured UIDP is displayed (in place) on the screen, and will even be updated dynamically if the coded UIDP supports such updates. (Coding for this process was partially completed as a proof of concept, but is not fully functional.)

The intent of the wizard is that the UIDP widgets are carefully human-factored designs, and a guiding wizard will capture a considerable portion of the application designer's intent while preserving the human factors design of the widget. This also promotes code reuse.

When the designer saves the Glade GUI, all aspects of each UIDP used in the design are saved as part of the Glade GUI's XML file. This same file is reloaded and reprocessed if additional Glade GUI editing operations are performed. (Sample XML code was partially completed for the proof of concept.)

An application designed to use the Glade XML file will also, necessarily, include the UIDP widget. The same (Python) implementation code that displays the UIDP in Glade during the application design phase is also used to actually execute the widget within the application. This means that the UIDP widget module(s) must be included on the end-user's system, along with relevant libraries. (No portion of this process was coded for the proof of concept.)

The original research intent was to code a small collection of these patterns into a single cross-platform library, and distribute this library as an example of UIDP widgets with full human factors support. However, a variety of technical issues prevented the UIDP widgets from being fully coded. Late Glade library developments and improved external documentation significantly increased the ability for these sorts of designs to be successfully integrated into Glade, but time constraints prevented real progress from being made.

Despite our own setbacks, we believe that a small and active programming and human factors staff of 3 to 5 people could quickly develop and integrate one or two example patterns within a matter of a few months, given the references we generated, our own examples, and newly published work in this area. After this, only moments of work would be necessary in order for application developers to fully utilize patterns from a Glade UIDP library module.

## 7.0 CONCLUSIONS

Efficient and effective software design is the grail sought by software professionals and technologists worldwide. Incremental steps are continually being made toward this objective: more accurate computing paradigms and information theory, specialized computing languages, and more capable and more complex software libraries. Together, these enable programmers to generate and maintain software systems faster, with likelihood of fewer bugs, and with greater flexibility than ever before.

A user-interface design pattern library is the next evolutionary step in software libraries. With design pattern tools such as the map tool and timeline tool in place of small descriptions of patterns with a snippet of code, an easier method of implementing these documented patterns is formed. Instead, an interactive template will guide the creator to produce the desired user-interface with minimal programming errors, and a human factors engineering component is already included in the code templates.

The implementation architecture is simply an effort to create a proof of concept for UIDPs, and these preliminary implementation efforts are subject to change as the technology continues to advance. Meanwhile, these venues may prove valuable to those desiring to continue this research focus:

- Continued engagement with DoD Human Factors Engineering Technical Advisory Group; UI Design Patterns are widely recognized as important.

- Engagement with the computing community through WorldComp (major international computing conference)

The development and documentation of UIDPs has the potential to decrease build time, increase product quality, reduce programming errors (development and cost risk), and improve coding efficiency. Most of all, use of these patterns at the coding and design levels will improve the user experience, since these design patterns include a human factors component that supports the ability of the end-users to better complete their work.

## 8.0 SELECTED REFERENCES

### User Interface Design Patterns

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. 1st ed. Reading, MA: Addison-Wesley, 1994

Alexander, Christopher. The Timeless Way of Builder. 1st ed. Oxford: Oxford University Press, 1979.

Tidwell, Jenifer. Designing Interfaces: Patters for effective Interaction Design. 1st ed. Sebastopol, CA: O'Reilly Media Inc., 2005.

Tidwell, Jenifer. "Common Ground." Common Ground. 17 May 1999. 7 Feb 2008 http://www.mit.edu/~jtidwell/common_ground.html

Conrad, K., Stanard, T. (2007). Advanced Design Patterns: Enabling Designers of Complex Systems. *Proceedings of the 2007 International Conference on Software Engineering Research and Practice (SERP).* June, 25-28, 2007, Las Vegas, NV, USA.

Knapp, J.R. (2006). Specification for Visual Requirements of Work-Centered Software Systems. Master's thesis, Wright State University, Dayton, Ohio.
"GNOME Human Interface Guidelines 2.2" GNOME Documentation Library http://library.gnome.org/devel/hig-book/stable/.

### XML

Harold, Elliotte, and W. Scott Means. XML in a Nutshell. 1st ed. Sebastopol, CA: O'Reilly & Associates Inc., 2001

Ray, Erick. Learning XML. 2nd ed. Sebastopol, CA: O'Reilly & Associates Inc., 2003.

### Eclipse

D'Anjou, Jim, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. The Java Developer's Guider to Eclipse. 2nd ed. Boston: Addison-Wesley 2004.

Eclipse, www.eclipse.org.

Orme, Dave. "Extending the Visual Editor: Enabling support for custom widget" eclipse.org. 20 Jun 2005, 7 Feb 2008 http://www.eclipse.org/articles/Article-VE-Custom-Widget/customwidget.html.

Klinger, Doina. "Creating JFace Wizards" eclipse.org 16 Dec 2002, 7 Feb 2008 http://www.eclipse.org/articles/article.php?file=Article-JFaceWizards/index.html.

"JFace" eclipse wiki 15 Aug 2008, Aug 2008 http://wiki.eclipse.org/index.php/JFace.
Proulx, Emmanuel. "Eclipse Plugins Exposed, Part 1: A First Glimpse" O'REILLY on Java 09 Feb 2005, 7 Feb 2008 http://www.onjava.com/pub/a/onjava/2005/02/09/eclipse.html.

Good, Nathan A., "Build extensions for Eclipse one snippet at a time" ibm.com 26 Jun 2007, 7 Feb 2008 http://www.ibm.com/developerworks/opensource/library/os-eclipse-snippet/index.html#understand.

"VE Extension Points" eclipse wiki 18 Oct 2007, 7 Feb 2008 http://wiki.eclipse.org/VE_Extension_Points.

## GUI Builders

Stuart, Mitch. "Java GUI Builders." fullspan. 06 April 2005. fullspan. 7 Feb 2008 http://www.fullspan.com/articles/java-gui-builders.html.

## Python

Christopher, Thomas W. "Objects and Classes in Python" informIT 16 Feb 2002, 7 Feb 2008 http://www.informit.com/articles/article.aspx?p=28672&seqNum=1.

Martelli, Alex. Python in a Nutshell. 1st ed. Sebastopol, CA: O'Reilly & Associates Inc., 2003

Python Imaging Library (PIL). 03 Dec 2006, 7 Feb 2008 http://www.pythonware.com/products/pil.
"convert a PIL image to a GTK pixbuf" snippets. 07 Sep 2005, 7 Feb 2008 http://snippets.dzone.com/tab/pil.

## Glade

Glade – a User Interface Designer for GTK+ and GNOME. 16 Mar 2009 http://glade.gnome.org.

Afshar, Ali. "Custom PyGTK Widgets in Glade3" PyGtk. 21 Mar 2007, Mar 2009 http://www.pygtk.org/articles/custom-widgets-glade/Custom_PyGTK_Widgets_in_Glade3-part-2.html.

Carrick, Micah. "GTK+ and Glade3 GUI Programming Tutorial" Micah Carrick 24 Dec 2007, 7 Feb 2008 http://www.micahcarrick.com/12-24-2007/gtk-glade-tutorial-part-1.html.

## Other References

"Google Maps." Google. 07 February 2008. Google. 7 Feb 2008 <http://maps.google.com/>.

## 9.0 LIST OF ACRONYMS

| | |
|---|---|
| ADT | Abstract Data Type |
| C2 | Command and Control |
| DoD | Department of Defense |
| GUI | Graphical User Interface |
| HCI | Human-Computer Integration |
| HFS | Human Factors Specialist |
| IDE | Integrated Development Environment |
| UIDP | User Interface Design Patterns |
| XML | Extensible Markup Language |