PYSAFE: AN INTERDISCIPLINARY APPROACH TO INTERFACE DESIGN

A Project

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Arts

in

Interdisciplinary Studies:

Interface Design

---

by

Matthew Adam Richardson

December 2008

PYSAFE: AN INTERDISCIPLINARY APPROACH TO INTERFACE DESIGN

A Project

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Matthew Adam Richardson

December 2008

Approved by:

Richard Botting, Chair, Computer Science and Engineering	Date
David Turner, Computer Science and Engineering	
Kurt Collins, Art	

ABSTRACT

The need for designing software to work congruently with people has been recognized for over forty years; yet the field of Human-Computer Interaction is relatively new, coming of age in the last twenty-five years or so. A rapid maturation occurred, as designers and developers progressed from basing interfaces around workflow analysis to utilizing cognitive psychology to better account for the needs of people. However, the quickening pace of globalization has created a large hole in the design process: making interfaces accessible to a worldwide audience. The inclusion of cultural understanding and the field of semiotics offer potential solutions to the problem of creating better interfaces for all users, local and global.

PySafe is the result of the development of an application from the interface down, rather than from the system up. Once a purpose was settled upon, an inventory tracking system, design software was used to quickly explore ideas and create a mock up. The mock up served as a guide for the development of the user interface and application logic.

ACKNOWLEDGMENTS

I would like to thank Dr. Richard Botting for his support, guidance, and sharing his wealth of knowledge; Dr. David Turner for getting me up to speed in a field mostly foreign to me; and Professor Kurt Collins for teaching me how to see critically. Thanks also go to my family and friends for their encouragement.

To Natasha, Caitlyn, Lauren, and Georgia.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

Purpose

As interactive devices become more integrated with daily life, the importance of effective design grows. In order to produce well-designed applications, it is necessary to know what is happening in interaction design now, how it came to be, and where it may be going. The field of human-computer interaction has seen many changes in its brief history. From the initial encouragements to make applications work like people work to the application of semiotic theory to graphic elements, interaction designers have sought to improve people's experience. Because the design of an interface is very much an art, a wide-array of opinions exists on what needs to be considered in order to produce well-designed applications.

The purpose of this project is two-fold: examine the prevailing wisdom of the interaction design field over the past forty years and look at emerging perspectives, then apply the established principles along with new trends in a practical application.

Scope

To meet the goals of the purpose, the project consists of a literature review and a working application. The literature review covers approximately forty years of material pertaining to the human-computer interaction field, as well as exploring the effects of globalization on future directions of interaction design. The application is presented from the design of the graphical user interface through software design and testing. PySafe is a computer tracking system for inventory purposes, designed with portability, extensibility, and internationalization in mind.

Definition of Terms

Affordance: A cue that indicates how something is to be used.

Activity-centered design (ACD): Design focused on the collection of tasks used to produce a result.

Application Programming Interface (API): A set of functions provided by some piece of software to support requests made by other software.

Flash: Adobe Flash, animation software (http://www.adobe.com).

Graphical User Interface (GUI): An interface composed of windows, buttons, icons, and other visual metaphors.

HTTP: Hypertext Transfer Protocol, used for transferring data over the Internet.

Human-computer interaction (HCI): Field of study interested in how people interact with computers.

Inkscape: An open source vector drawing program (http://www.inkscape.org).

Interaction Design: In the broadest sense, the design of anything that a person has to give input to and receive output from.

Internationalization (i18n): The process of making software adaptable for use worldwide. The abbreviation is derived from the number of letters between the first and last letters.

Localization (l10n): The process of adding locale specific details, such as language.

Media Access Control (MAC) address: A unique address assigned to a network interface card. It provides a convenient way of identifying a particular computer as no two are supposed to be alike.

Persona: A fictional person representing a class of people in the target audience, used as a design tool.

Profile: Broader than a persona, it describes the audience as a whole.

Python: Interpreted, object-oriented programming language.

Scenario: A description of how a product will be used.

SSL: Secure Sockets Layer, a protocol used to establish secure communication across the Internet.

Traceroute: A program that finds routers between the client computer and a specified host.

User-centered design (UCD): Design focused on people (users) and the tasks used to achieve a goal.

XML: Extensible Markup Language is a specification for creating user-defined encodings of data.

Document Organization

A literature review is presented to provide historical background and emerging ideas to be incorporated in the project. Following that, an overview of the design process is presented, beginning with the development of the interface through graphic design software. Then the software design and testing process is detailed. All of the source code for the application is included in the appendices.

CHAPTER TWO

LITERATURE REVIEW

Introduction

Human-computer interaction (HCI) is concerned with that part of computing with which most people are familiar: the stuff they need to use on their computer screen. In the early 1970s emerged proponents of the importance of designing an interface so that it presents a model that matches the user's own model of how things should work; however, the field is still coming into its own. In the early to mid 1980s, developers and researchers began to look at workflows and make observations, in order to make improvements to the designs. By the late 1980s and early 1990s, researchers were incorporating cognitive psychology to account for human limitations in their design processes. Over the last ten years or so, the ideas of user-centered and activity-centered design have become prominent.

In roughly those same ten years, the pace of globalization has jumped tremendously. The massive deployment of high-speed communications cable and the economic pressure to find talent at a lower cost have brought many disparate areas of the globe into closer

contact than ever before. Of course, this means that the diversity of people using software has grown much faster than designers have been able to keep up. It has only been in the last four years or so that designers have been applying cultural studies theories and semiotics as possible strategies to close this gap in the human-computer interaction field.

As businesses grow and expand into the emerging foreign markets, the need to develop interfaces that require zero to little customization for each locale will become increasingly urgent. No one wants to spend time re-implementing the same graphical interface for each market, especially when that costs money to pay for developer time. Time that may be better spent on improvements for the next version to be shipped. The upside of market expansion is that interfaces for all users should improve as developers take into account cultural issues as well as the cognitive- and activity-related ones.

However, the decisions made by developers raise a lot of questions. Where has HCI come from and where is it now? What are the effects of globalization and how do they affect HCI? What new directions are being explored to

create better interfaces? Is it possible to design once, run everywhere?

A Brief History of Interface Design

To understand where the field is going, it is important to know where it has come from. Advocacy of user-centered design began early on, with research in how to deal with common problems such as error handling. Afterward came the integration of cognitive psychology. User-centered design is the most recent addition to the field.

Ted Nelson released the self-published Computer Lib/Dream Machines, in June 1974. Actually two books in one (Dream Machines being the second half, reversed to Computer Lib), it outlined in a rather anarchic way Nelson's thoughts on the computer industry, programming languages, networks for sharing knowledge, hypertext, payment of content producers, multimedia, and interfaces. At a time when computing meant time-sharing and punch cards for many people, his ideas on making interfaces usable by people were truly remarkable.

Nelson defined interactive systems as anything that responds to input (1987, 67). This definition broadly

covers computing in general, at least the way in which most people will use a computer. Recognizing this, lists of things and aphorisms to keep in mind while designing software are presented throughout. The major themes are conceptual clarity (Nelson, 1987, 25) and ease of use (Nelson, 1987, 12).

Along with the ‘thirty thousand foot view’ and research-based works, is the practical application genre of work. Drawing on research referenced from other sources, this work provides the software developer with a quick way of getting to the heart of the matter: applying the principles derived from the study of human-computer interaction. Mehlmann (1981) made a number of points regarding the production of terminal-based business software with an eye on making the workflow similar to the non-computer based workflow.

Written for a programming audience, Mehlmann points out that there are consequences for poor work (1981, 23). Suggestions on how to work with limitations in mind include requiring as little manual entry as possible, eliminating as many possible points of potential error as possible. To create better software, the complexity of the system should

be confined to the background, away from anything the person has to deal with (Mehlmann, 1981, 45).

In a case study of application development with a focus on interface issues, Ledgard, Singer, and Whiteside (1981) documented the process as completely as possible. From discussions of design decisions to what worked and why, and even what didn't work and why, the shift toward accounting for the idiosyncrasies of people is illustrated from the inside. Noting that a lack of consideration for HCI results in fewer people using computers, the authors considered everything from prompts to grammar and long-term feedback to make improvements. An important insight that seems to have informed the project they detailed is that previously designed systems are not perfect (Ledgard, Singer, and Whiteside 1981, 141). This immediately suggests that design is a process, not a product, and therefore is always short of the ideal. Some aspect can always be improved.

Gilb and Weinberg (1984) dealt with issues of keyed input and conducted research on how to reduce errors, increase productivity, and improve working conditions. While the specifics of their research may not be as important today, keyed input being replaced by automatic

input whenever possible, many of the general concepts they drew from their data are very much relevant. Steps to take in error avoidance, such as defaults, implicit and explicit data entry and input checking are explored in great detail. As for improving working conditions, it is pointed out that forcing people into strict work flows does not do anyone any good, neither for programmer nor user (Gilb and Weinberg, 1984, 180), and that people like to think, given the chance (Gilb and Weinberg, 1984, 130).

By the late 1980s, the cognitive psychologists had entered the field. The constraints, mental and physical, that people have to deal with are addressed and design practices to take them into account are detailed (Shneiderman, 1987). Ways of making things easer for people to figure out, perceived affordances, and taking advantage of knowledge in the head as well as putting some knowledge in the world are explored as means of making more effective interfaces.

Schneiderman notes that a model for software design outlining four levels-conceptual model, semantic level (where meanings are conveyed by input and output), syntax level (specific commands), and the lexical level (device dependencies)-compares with Norman’s model for user

interaction-forming an intention, selecting an action, executing the action, and evaluating the outcome (Schneiderman, 1987, 46-47). To address the semantic level of the model, designers employed positive and negative examples of use as well as tied together new concepts to previously held knowledge (Schneiderman, 1987, 49). Furthermore, tying structured semantic knowledge of computer paradigms to things users already knew resulted in better memory retention (Schneiderman, 1987, 50).

On the topic of preventing errors, Schneiderman advocated the use of organization by function, providing distinctive choices when presenting options, and making actions reversible (1987, 69). The methods he presented are the same as earlier researchers had drawn. Those methods were also applied by future designers, as discussed below. What is different about Schneiderman's analysis is that his focus is slightly different from the earlier perspective. That, basically, people make mistakes and the software should take this into account. Instead, his view is that when designers know users physical limitations, such as visual perception fields, and their mental model allows them to tailor the interface to meet the basic elements of a good design. The criteria he provides for

evaluating an interface-and ditching the term ‘user-friendliness'-are:

- Time to learn;
- Speed of performance;
- Rate of errors by user;
- Subjective satisfaction;
- Retention over time (Schneiderman, 1987, 73).

To underscore the importance of thinking outside of the engineering box when designing for people, he quotes Heckel as saying that thinking logically rather than visually is not conducive to good design (Schneiderman, 1987, 198).

It is here that Norman picks up the study, not of computer interfaces in particular, but of interactive objects in general. Visual cues create the relationship between what the user means to do and what can be done within the interface (Norman, 2002, 8). Too few cues make a device difficult to figure out, too many make it confusing. A balance has to be struck that provides the right amount of cues in order to facilitate the use of the device. Norman called these cues affordances, real and perceived properties of an object that indicate its use (Norman, 2002, 9).

Providing a clear conceptual model is just as important as affordances to the success of a design. The principles of affordances, constraints, and mappings are illustrated by contrasting scissors with a digital watch: scissors have two holes, with a size constraint that makes them suitable for fingers, whereas a digital watch may have buttons on the side that do not map clearly to any particular function. The conceptual model formed of the scissors is clear, whereas the wearer of the watch is going to find themselves mashing buttons (Norman, 2002, 12-13).

The conceptual model can be broken in to three parts; the designer's model, the system image, and the user's model. Ideally, the designer's model and the user's model will match and everyone is happy. However, since the two do not communicate directly, the system image (which is the system, documentation, instructions and everything else accompanying the product) must be formed in a way that properly informs the user of the designer's concept or the user may form an incorrect conceptual model.

The possible arrangements of a particular interface, a good one anyway, are reduced by external and internal constraints. Norman denotes the reasons for this as: information being present in the world that can be combined

with previous knowledge to complete a task; a high degree of precision is not necessary, given enough knowledge to make the correct choice; natural constraints exist that restrict possible actions, like a knob which must be turned, not pulled; and cultural constraints exist that govern acceptable behaviors (Norman, 2002, 55). With these four conditions, people can figure out fairly quickly how something is to be used without having prior knowledge of the system, simply by taking advantage of what is presented to them and what they bring to it.

Being human, designers of course experience pitfalls in achieving their goals. The two major categories of errors are mistakes, where the goal is incorrect but the actions are correct for the goal, and 'slips' when the goal is correct but an automatic or subconscious action is incorrect for achieving the goal (Norman, 2002, 105). Error detection built in to an interface can help alleviate problems caused by taking the wrong action, but feedback has to be given to the person so that the problem is recognized. Care has to be taken with error detection and correction, as people will come to rely on it (Norman, 2002, 114).

One of the most important things that Norman has to say in designing for people is that designers are not normal people and that there is no substitute for interacting with the intended users. The complexity and variability of human thought, emotion, culture, belief, etc., are really beyond the grasp of one person (Norman, 2002, 155). To do so would be to deny one's own biases and perceptions and assert that the role of the user can be assumed. Later, Norman will discuss this more and clarify his position, which is not that all of a designer's design has to be derived from analyzing the user, but that failing to account for the audience's specificities is going to result in a missed goal.

It is pretty apparent that the study of HCI, from the idealist (Nelson) through the developers (Ledgard) to the cognitive psychologists (Schneiderman and Norman) was evolutionary. The importance of this conclusion is that HCI is truly an interdisciplinary field, with important contributions coming from many areas. However, the current paradigm, arising in the mid-1990s, has taken too much to heart the study of users and analyzing the task domain, at the expense of making interfaces too focused.

The Current State of Interface Design

So, with the progression of the first half of the development of HCI behind us, it's time to take a look at the second half. Broadly, there are four approaches to interaction design: user-centered, activity-centered, systems, and genius (Saffer, 2007,30).

User-centered design is focused on the needs and goals of the users of the proposed software. 'Personas' are detailed representations of people in the target audience. They are developed through interviewing a cross-section of audience to determine skill-levels, tasks that need to be accomplished, and work flows (Tidwell, 2006, 7). While not representing any one particular real person a persona is a highly detailed fictional person, complete with some biographical detail, for example, children, hobbies, etc. The other components of defining the user base are profiles and scenarios: a description of a range of attributes, such as position or age, and how the person will use the product, respectively (Courage and Baxter, 2005, 41). The process involves determining user requirements, testing effectiveness, and using an iterative design cycle (Courage and Baxter, 2005, 4).

Users can often be considered co-designers, being incorporated in the process at each step of the way. A benefit, and no small one, is a huge investment in the success of the project on the part of one of the major stakeholders: the user. Important insights can be had by utilizing the institutional knowledge of people familiar with workflows, business rules and peculiarities of the environment.

However, there is also a large risk in designing the system to suit the biases of a subset of the total users. Great care has to be taken in the selection of collaborators to avoid this trap. It also shifts some of the responsibility of the design to people who are not trained in the field and may result in inappropriate concessions to please the testers.

User-centered design has been around for some time and has many adherents. Either through use of abstractions like personas or through the use of a subset of the target audience as participants in the design process, it is certainly an established means of designing with the person in mind.

Activity-centered design, on the other hand, seems to be the upstart (Norman, 2005). Whereas UCD focuses on

users and the tasks they undertake to achieve some goal, activity-centered design focuses more on the collection of tasks that are undertaken to produce a result. Since it is a newer development, it is discussed further below.

Systems design is the least humanized method of interaction design. It is an analytical method of approaching a design problem, with distinct components that the designer is responsible for producing. The goals of people are not discounted but are taken into consideration in the context of the system: the goal in this model is the system goal, which may be taken from user goals (Saffer, 2007, 36). Design in general is fraught with uncertainties and this method provides a rigid framework as a means of eliminating that. However, it would appear to be a throwback to the days when systems were engineered and people had to accommodate the system.

The fourth method is genius design (Saffer, 2007, 41). One designer is responsible for producing the object, from concept to completion. From the designer's perspective, this approach has some real advantages, namely artistic freedom and an easier workflow (no committees or teams to deal with). An experienced designer may be able to develop a product that is revolutionary, drawing upon years

of experience on other projects. Nevertheless, the flip side is also true: design without consultation may result in a huge flop. Apple, Inc., practices this method with many of its consumer items and thus has seen both huge successes and huge failures. The aversion to design by committee is not new, Nelson advocated a chief designer with absolute power (1987, 72) as a way of ensuring conceptual integrity. Interestingly, Apple also follows a user-centered approach in its Human Interface Guidelines, suggesting that the company stays at the fore of the field by hiding their designs until ready to reveal, while at the same time enforcing standards on third party vendors.

There are, of course, other methods of interaction design. One is simply to make a product look pretty after everything else has been said and done. Consumer electronics, sadly, often fall into this category: the DVD/VHS player, remote controls, and alarm clock that are used daily often leave much to be desired. Strictly speaking, this is window dressing and not interaction design. There are so many constraints on the product at the final stages of completion that the designer has very little influence over important aspects such as features, input methods, work flows and so on.

Another method is to skip the designer and let the programmers take care of it. Spolsky suggests that the only thing usability testing is good for is reminding the programmers that they are not writing for themselves (2001, 100). What is really fortunate about his usability testing approach is the fact that he has compiled a good set of axioms from previous work in the field: make the system image fit the user's mental model (2001, 6); options mean decisions (2001, 18); know that affordances help, bad metaphors hinder; and make things work acceptably under the worst conditions and better under normal conditions (2001, 59). Spolsky's book is really a fantastic primer on design for programmers; however, toward the end of the work he asserts that, given the rules and axioms he has introduced, using professional designers is a waste of money (2001, 101). The unfortunate message delivered by an otherwise good book is that a set of recipes is all that is required to produce a good interface.

In what can be viewed as a counter to Spolsky's assertion, Schneiderman (1987, 391) set forth the following:

- Design is a process, it is not a state and cannot be adequately represented statically.

- The design process is non-hierarchical; it is neither strictly bottom-up nor strictly top-down.
- The process is radically transformational; it involves the development of partial and interim solutions that may ultimately play no role in the final design.
- Design intrinsically involves the discovery of new goals.

Despite the copious work being done in the HCI field, it would seem that the tradition begun in the mid-1950s with industrial design has remained out of scope for software engineers (Saffer, 2007, 31). The disconnect between engineers and designers may lie in the notion of design as a problem solving process, with an implication that the problem has a right or wrong solution (Lowgren and Stolterman, 2004, 9), whereas Schneiderman argues that it is a process in which the results are more successful or less successful, but without a single ‘correct’ solution. Lowgren and Stolterman state that the process is not predictable, there are too many variables involved-people, resources, conditions-to be able to determine an outcome (Lowgren and Stolterman, 2004, 9).

The design activity encompasses many aspects. It is an ethical activity in that the decisions made in the design affect people and their actions. It is an aesthetic activity in the sense that the forms created add to our constructed world and how we experience it. What may become most important are the political and ideological aspects. Each designer brings their own perspective on humanity and society, and the influence of design on people's lives makes these political and ideological ideas tangible (Lowgren and Stolterman, 2004, 10).

Globalization

To some, the term ’globalization’ denotes a growing global economy, with countries exchanging goods and services at a level not seen before. To others, it is the new word for imperialism, where the richest nations get richer by taking advantage of the less privileged. For this discussion, the definition provided by www.globalization101.org will be used:

> Globalization is a process of interaction and integration among the people, companies, and governments of different nations, a process driven by international trade and investment and

> aided by information technology. This process has effects on the environment, on culture, on political systems, on economic development and prosperity, and on human physical well being in societies around the world.

Clearly, there are many issues and many opinions revolving around this topic. The aspect of globalization important to our discussion is that the interaction among diverse cultures is becoming more commonplace, through personal and economic avenues, and this is going to have a profound effect on interpersonal and international relations, business, and the growth of technology.

Friedman (2009, 9-10) maintains that there have been three eras of globalization. The first is demarcated by the discovery of the Americas, lasting until 1800. This period is characterized as being one of power and force, with nations expanding and seeking to build empires. The second period starts in 1800 and ends in 2000, a time of multinational corporations seeking power and fortune. The third, the current era, begins around 2000 and is when, Friedman argues, the individual has the ability to compete on a global scale. The convergence of wide spread use of personal computers, high-speed data communications

networks, and work flow software has enabled a much wider spectrum of people to engage each other and markets than in the previous eras. Of course, this means that as more people get access to this framework, the push will come increasingly from parts of the world other than the West (Friedman, 2007, 11).

The emergence of the World Wide Web (WWW) in 1991 kicked off the newest era. The invention of Tim Berners-Lee, it was meant to be an easy way for scientists to share documents and collaborate. However, within five years the number of web users jumped from 600,000 to 40 million (Friedman, 2007, 62). The Internet existed before the WWW came along, but it took an easy method for non-technical people to take advantage of the massive communications network. Where these people originated from is not entirely clear, but considering that the Internet started as the Arpanet, a Department of Defense project, and Berners-Lee was working at CERN, the European Organization for Nuclear Research, it is probably safe to say that most web users during the early 1990s were from Western countries.

Statistics gathered by Dr. Richard Botting support this assertion. In his collection of new domains announced

on comp.infosystems.www.announce, a Usenet group, the top ten domains are dominated by .com, .uk, and .au in 1995. In 1996, the top-level domains for Germany, France and the Netherlands broke into the top ten.

The tech boom of the late 1990s saw a large number of web-based start up companies and the accompanying infrastructure companies frantically trying to capture market share, with no end in sight. There was an end, and it hit in the early 2000s. By this stage, many e-commerce businesses had fallen by the wayside and infrastructure companies had laid so much fiber optic cable that data transmission prices fell to near zero (Friedman, 2007, 74).

Web standards such as HTML (hypertext markup language), HTTP (hypertext transfer protocol), SOAP (simple object access protocol), and XML (extensible markup language) made it easy for machines to talk to one another and exchange data independently of the platform or application (Friedman, 2007, 84). For example, two different programs in two different locations could send data back and forth to one another as long as they both understood XML and could communicate across the Internet, often done using a web standard due to the robustness of the protocols and server software. Coupled with the huge

(and cheap) data links, business could now be global and function in real time.

With the low threshold to the market and an abundance of talent in information technology and business administration, India became a leader in providing services to businesses in the West. Other countries, such as China, Russia, and former Soviet satellites, were able to take advantage of this niche as well (Friedman, 2007, 126).

The growing response in the software community has been through internationalization (i18n) and localization (L10n) of products. The first, i18n, is the process of making software adaptable for use worldwide, whereas L10n is the process of actually putting in the locale specific details such as language, currency, date formats, etc. (Wikipedia, 2008). While this is a critical step in making software that can potentially be used anywhere, it is only providing one piece of the puzzle. The next step is creating interfaces that meet the needs of each locale. So, the question becomes, is this possible and if so, to what extent?

New Directions

If it really takes a generation for changes in technology to result in changes in productivity and work flow (Schwartz and Leyden, 1997), we should now start seeing the effects of the interaction design community on software in general and the effects of globalization on the interaction design community about now. The rise in amount and extent of intercultural relations is bringing about wider awareness of the need to account for cultural differences when designing for global audiences. With the incorporation of HCI principles in software design, designers are reflecting on whether their assumptions derived from user-centered design are adequate. And just to make things more interesting, a semiotic engineering approach to HCI is emerging in which the influence of culture on interpretation of signs is being explored.

Hofstede's work in examining cultures and assessing their nature according to five dimensions has become a key reference for many researchers interested in intercultural aspects of computer-mediated communication. The data source was a collection of surveys administered by IBM to employees at its offices across the world. The results of his analysis of this material and later, similar inquiries,

are his definitions of five aspects (dimensions) in which cultures can be qualitatively compared to one another.

The first dimension describes the distance the least powerful in society have to the most powerful, the Power Distance Index. The PDI indicates the amount of dependence between subordinates and superiors, a high score indicating a high dependence of subordinates to their bosses and a low score indicating little dependence in the relationship (Hofstede and Hofstede, 2004, 45). Based on this, power distance is defined as the level of acceptance of the unequal distribution of power within an organization, from the perspective of the least powerful members of the organization (Hofstede and Hofstede, 2004, 46).

The second dimension is labeled as individualism versus collectivism and indicates the level to which individuals are part of groups. Individualistic societies have loose ties between their members, with the expectation that each is to look after themselves and perhaps their immediate family, whereas members of collectivist societies are part of strong groups from birth that provide protection throughout life in exchange for loyalty (Hofstede and Hofstede, 2004, 76). While there are many differences between the two types of societies, the goals

of each are documented in the study and relate to personal growth for the individualist and contributing toward the group in the collectivist.

The masculinity versus femininity dimension is based on responses to questions relating to the work goals of earnings, recognition, advancement, challenge, good relationship with management, cooperation, living area, and employment security. Hofstede points out that he named this dimension because it was the only one on which men and women consistently scored differently (2004, 119). The men rated earnings and advancement highly; the women rated management relations and cooperation highly.

Uncertainty avoidance is the fourth dimension, and it is the extent to which people in a particular culture are threatened by ambiguous or uncertain situations (Hofstede and Hofstede, 2004, 167). Those cultures with a high uncertainty avoidance index would be more anxious than those of a low index. An interesting observation made regarding high anxiety countries is that the people tend to be more expressive and that low anxiety cultures tend to have members more prone to coronary disease, probably due to bottling up their emotions (Hofstede and Hofstede, 2004, 171).

The fifth dimension is that of long term versus short-term orientation. This one was added later after research was conducted in Asian countries based on questions coming from a Chinese perspective in order to test for a Western bias in the original IBM survey. Long term oriented cultures value perseverance and thrift (future rewards), where short-term cultures are oriented toward tradition, maintaining face, and fulfilling obligations (Hofstede and Hofstede, 2004, 210).

Having a dimensional frame of reference for the practices and expectations, even on an abstract level, can foster understanding and avoid embarrassments when HCI designers deal with other cultures. As discussed above, communication technology and global travel have seen the rate of intercultural meetings grow tremendously (Hofstede and Hofstede, 2004, 321). Being aware of how issues play out in these dimensions and cultivating intercultural communication skills to resolve conflicts and foster understanding will improve the chances of succeeding with international projects.

Debunking the notion that the global communication links will unite the world into a single village, Hofstede asserts that the flood of information does not affect

people's ability to absorb the information nor does it change their values (2004, 330). People will continue to gravitate towards those things that reinforce their values and predispositions and exchanges with other cultures will illustrate the differences rather than erase them (Hofstede and Hofstede, 2004, 330). The way to successful intercultural communication is through:

- Awareness: people are brought up in different environments and consequently have different mental software for equally valid reasons.
- Knowledge: interacting with a culture will require learning about it.
- Skills: the application of knowledge in practicing the various aspects of the culture (Hofstede and Hofstede, 2004, 359).

On the basis of Hofstede's work and others researching in the area of cultural aspects of international communication, a development framework can be established for particular regions.

As mentioned previously, standardized protocols and software has had a huge effect on globalization. However, evidence has arisen that a standardized interface poses problems for certain cultures due to the differences in

signs and visual cues (Li, Sun, and Zhang, 2006). The desire for compatibility creates a condition in which cultural considerations are ignored in favor of universality. A better understanding of cultural differences is necessary to better understand values and perceptions in order to provide a higher quality product that meets people's needs (Li, Sun, and Zhang, 2006).

Western and Eastern cultures have fundamentally different ways of seeing the world. Western perception tends to be analytic, categorizing objects based on attributes, while Eastern perceptions tend to be holistic, where the object is seen as part of a context and behavior is a function of relationships (Li, Sun, and Zhang, 2006). Because of these differences, a design effective in one culture will probably not be in another. Even between two Western cultures, the same may be found to be true.

Internationalization and localization fail to address the problem because they are geared toward recognizing differences in culture and have to be customized for a particular culture, leaving out those that have not been accounted for (Li, Sun, and Zhang, 2006). The commonality of culturally related problems in interface design has to do with semiotics and the interpretation of representations

within a particular context (Li, Sun, and Zhang, 2006). For successful interpretation, the context of the designer must be understood by the user. Conversely, the designer must understand the context of the user and design accordingly.

The degree to which a design agrees with a particular culture can be used as a factor in determining the amount of cognitive effort required by the user (Kralisch, Yeo, and Jali, 2006). The implication of this system science proposition is that cultural preferences have to be determined and accounted for, as those things will affect a user's perceptions of an interface.

Kralisch, Yeo, and Jali define culture from a system science perspective as a collection of valuation, thought and behavioral patterns (Kralisch, Yeo, and Jali, 2006). They make a number of hypotheses regarding web design and the communication of information to a wide audience that can be summarized as people of different backgrounds will prefer different methods of information organization and the more in line with cultural expectations, the better the experience will be. Their findings suggest that people with a low level of domain knowledge benefit from visual cues and that presenting information in the user's native

language provided more options for searching for desired information (Kralisch, Yeo, and Jali, 2006).

There is a strong argument made by Norman that task-oriented design has made things better in the user interface world, but that this sole focus is not good enough (Human-Centered, 2005). He argues that people do not perform individual tasks in performing their work; they perform activities that are composed of many tasks. Focusing on the tasks rather than the higher-order activity places the focus on specifics and not on the overall nature of the activity, leading to a lot of time spent learning about particular users and therefore excluding many others. Norman cites the use of automobiles and household items as two instances in which products were made without the benefit of user-centered design principles and yet people worldwide have managed to use them. Furthermore, the assertion of the UCD community that the tool should adapt to the user is baseless, since many other successful technologies have required much effort on the part of the user to become adept at, such as clocks, musical instruments, and even written language (Norman, 2005). As technology changes, people adapt and conversely as people change, the technology is adapted.

Activity-centered design (ACD) is the superset of UCD. ACD is broader in that it requires an understanding of the technology and the reasoning behind the activity. Norman states, 'to the Human-Centered Design community, the tool should be invisible, it should not get in the way.' With Activity-Centered Design, the tool is the way' (2005). In the hierarchical view of ACD, there is the activity, which is composed of tasks, which are made up of actions, which in turn are based on operations. In this manner, support for the user is implicit as the activity will ultimately be performed by someone, but the design focus is on the activity.

The main concerns with UCD are that developing toward individuals or even a group will make the product better for them at the expense of others, focusing on individuals today risks being obsolete tomorrow, and focusing on people detracts from understanding the activities they are attempting to perform. There are many valid lessons to be had from UCD, so there is not a push to discard the previous work and strike out for some brave new discipline. Norman does suggest that it is time to reconsider the lessons of UCD and incorporate knowledge from other fields, such as industrial design and ergonomics (2005).

In an earlier essay, Norman also discusses the perspective of design as a means of communication. In his earlier works, he proposed the three part conceptual model of a system, where the designer created a system image, ideally matching the user's model, and the user interacted with the system image. The new perspective of HCI today sees the designer conversing with the user through the system, putting affordances in the interface and explaining why they are there (Norman, 2004). He attributes this new model to de Souza and her work on semiotics and HCI.

De Souza approaches HCI with semiotics in an attempt to help users understand meanings in software and methods of use. Semiotics is the study of signs, signals, and how they are used in communication (de Souza, 2005, 3). As signs are produced within cultural contexts, there are some in the semiotics field that view semiotics as a theory of culture and it is this aspect of it that interests those seeking to understand computer-mediated communication between various groups (de Souza, 2005, 3). As graphical interfaces are full of signs, symbols, and icons meant to communicate an idea or indicate an action, designers need to be acutely aware of the meanings that they are encoding and that the users will be able to decode them as intended.

If this exchange happens and both sides of the interface understand each other, users will be happy. As this process involves a study of not only the user's frame of reference, but also the designer's, semiotic engineering is a reflective theory (de Souza, 2005, 7).

If an intellectual artifact, as opposed to a material artifact like a chair, has the following properties:

- Encodes a particular understanding of a problem;
- Encodes a set of solutions for the given problem;
- The encoding of the above is symbol-based (linguistic);
- use of the artifact is dependent upon being able to place it in the users linguistic context;

then what has been described as perceived affordances in an interface are better understood as metaphors or symbols of intention and meaning (de Souza, 2005, 10). This insight leads to the homogeneous model, uniting users and designers in HCI through the interface medium by illustrating that, as Norman summarized above, the encoding and decoding of messages constitutes a communication. The message encoded by the designer becomes the deputy, which presents all of the rationale, principles, and processes used in creating the artifact (de Souza, 2005, 24).

This communication comes from a process in which the designer analyzes the user's situation in terms of environment and activities, drawing on cognitive, cultural and other influencing factors. Then designers express their perception of how the user will need to work through the interface. Users then decode and finally make sense of the design and respond to it (de Souza, 2005, 25).

Signs can be thought of as having two components, a representation and the object represented, or having three components, a representation, what it refers to, and a meaning. The latter provides an explicit inclusion of meaning, and the theory behind it goes on to say that the meaning is really another sign, making the description recursive and infinite. Human limitations obviously place a limit on this recursion, but it still stands that the variety of interpretations to be had make it impossible to predict exactly how a sign will be decoded (de Souza, 2005, 41-2). Within a given cultural context, the number of possibilities is reduced further, but the potential for unintended interpretations still exists.

HCI is at an interesting junction. The adherents of UCD are still gaining acceptance amongst software developers, while at the same time researchers in various

fields are exposing its shortcomings. Activity-centered design argues for a broader view of what users want to accomplish, saying that UCD has done good, but is really too granular and fails to see the big picture. Research into cultural aspects of interface design are becoming more popular, especially by those outside of the United States, as more and more of the world is affected by global communications. Semiotics is poised to become another extremely useful tool for the development of interfaces by introducing concepts of mediated communication, with implications for intra and intercultural relations.

Conclusion

The history of HCI shows that the field has received contributions from many different disciplines. Cognitive studies have contributed greatly to understanding the mechanics of how users experience interfaces and cognitive psychologists have contributed methods to facilitate ease of use. User-centered design, born out of these studies, is probably the most influential contribution as it shifted focus from the machine to the user. While this has been a huge step forward in creating useful artifacts, it is certainly not the apex of the field.

Rapid expansion of high-speed global data and communication networks has increased the growth of globalization, creating relations between more people from more diverse areas than at any other time. This environment has brought the importance of intercultural relations to the fore, as individuals and businesses expand beyond their local domains and in to those of others. Among other things, the need to be aware of differences between cultures and become knowledgeable about those cultures that are engaged has been highlighted, especially in computer-mediated communication.

Research conducted across cultural boundaries indicates that different cultures prefer different methods of having information presented to them. This undermines any notions of having a single silver bullet design style that can work equally well across the globe. Instead, it confirms the idea that each culture will have to be approached on terms that are understood by that group through the use of appropriate language, symbols, color, and organization.

Globalization and intercultural relations have exposed some of the shortcomings of user-centered design and new directions are being explored to better meet the needs of

users. Activity-centered design takes a holistic view of goals that people want to achieve in order to better understand what it is that needs to be done to accomplish those goals with better efficiency and a more enjoyable experience. The study of semiotics is being introduced on a more formal level than has been used previously. Advocating a view of design as being a communication process between the designer and the user, the conceptual model of UCD that separates the two is rejected in favor of a model that places designer and user on the same level, using the interface as the medium.

If it is to be successful, a design must be able to evolve. There are many aspects of an application to be considered when planning for the possibility of future changes. The interface should be adaptable to new audiences. The software should be modular, keeping the logic separate from the interface allows for alternative methods of interacting with the application to be developed easily.

Design is an art, not a science. Good design decisions are those informed by knowledge gleaned from many different sources, done in the best interest of the user. As conditions change, design processes must evolve to meet

them through the incorporation of new knowledge or taking a broader look at existing information. When design is accepted as a process, one in which new goals are discovered, one in which change is accepted as people, conditions, and environments move and shift, then issues such as intercultural communication cease to be problems and become exciting challenges instead.

CHAPTER THREE

INTERFACE DESIGN

Introduction

PySafe was originally conceived as a simple client/server program with a command line interface. The goal was to learn low-level socket programming and apply it to a real need, inventory control in this case. For a learning tool and proof of concept, a purely text interface was fine. However, there were two big problems with using it as a production application. The first problem was with the server that was implemented, which will be discussed in the next chapter. The second problem, and perhaps more important from a user's perspective, was the interface.

The text-based interface provided a menu and some guidance on how to use it, but the experience was not nice. That alone made it almost useless, not because it didn't do what it was supposed to, but because no one would want to use it. Since the basic flow and structure of the application would remain the same, the interface would become the focus of the new version and drive decisions about the rest of the application.

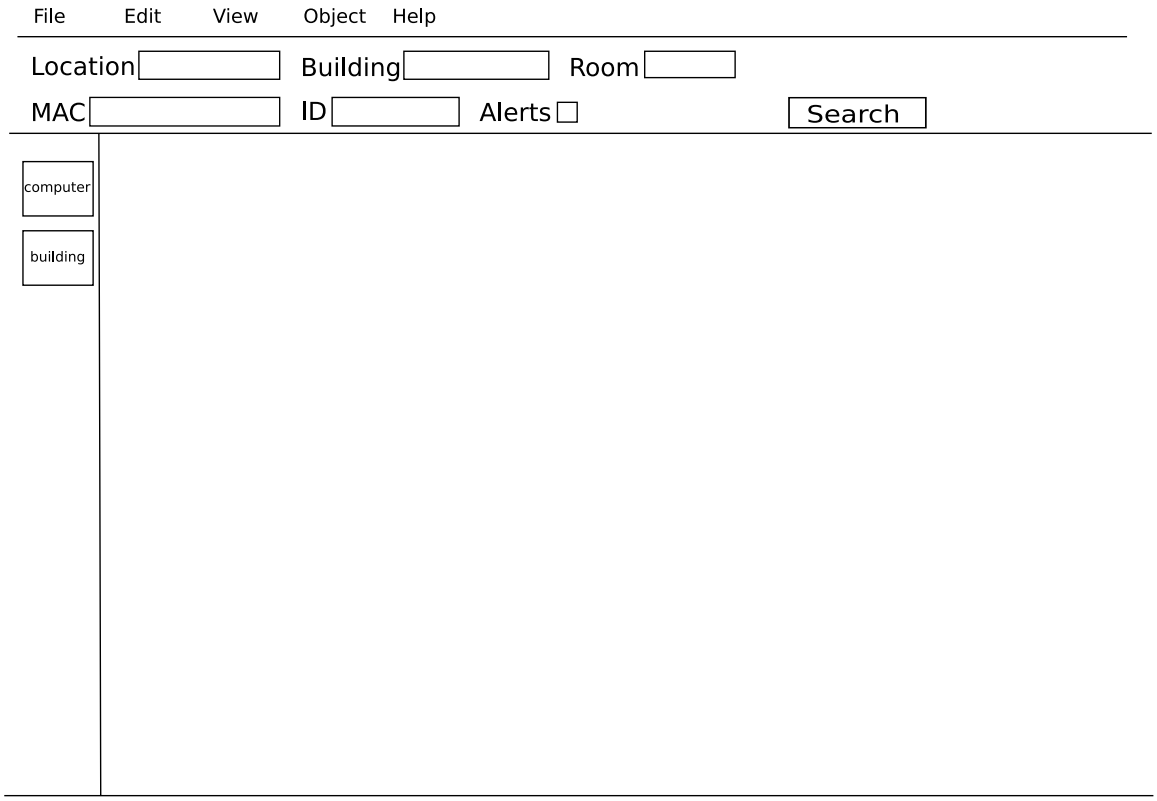

Fig. 1: Basic layout

Mock-Ups

Early on, a decision was made to do the initial design work using design software. This would allow for quick sketching of ideas and an interactive way of testing out different layouts and workflows. Inkscape, a vector-based drawing program, was used to draw the various components. Once the parts had been more or less fixed, they were imported into Flash to create an interactive application.

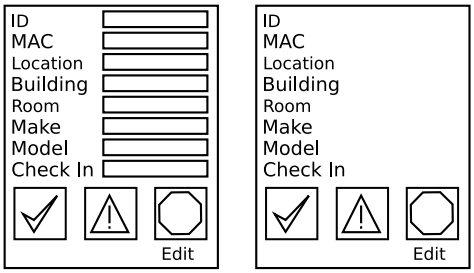

Fig. 2: Dialog boxes

Wireframes

A wireframe is a line drawing of what a product will look like. It focuses on composition and placement of elements. The vector drawings of PySafe were kept to wireframes to keep the design process moving quickly, see Fig. 1. Later, it would turn out that dealing with color and other decorations would have been largely pointless anyway.

The main window, search area, edit buttons, and map/list panel were laid out in a single composition. The dialog boxes (Fig. 2) and any other elements not directly part of the main window were rendered separately.

Using Animation

Using an animation tool to simulate using a program turns out to be a very fast way to visualize and experience how

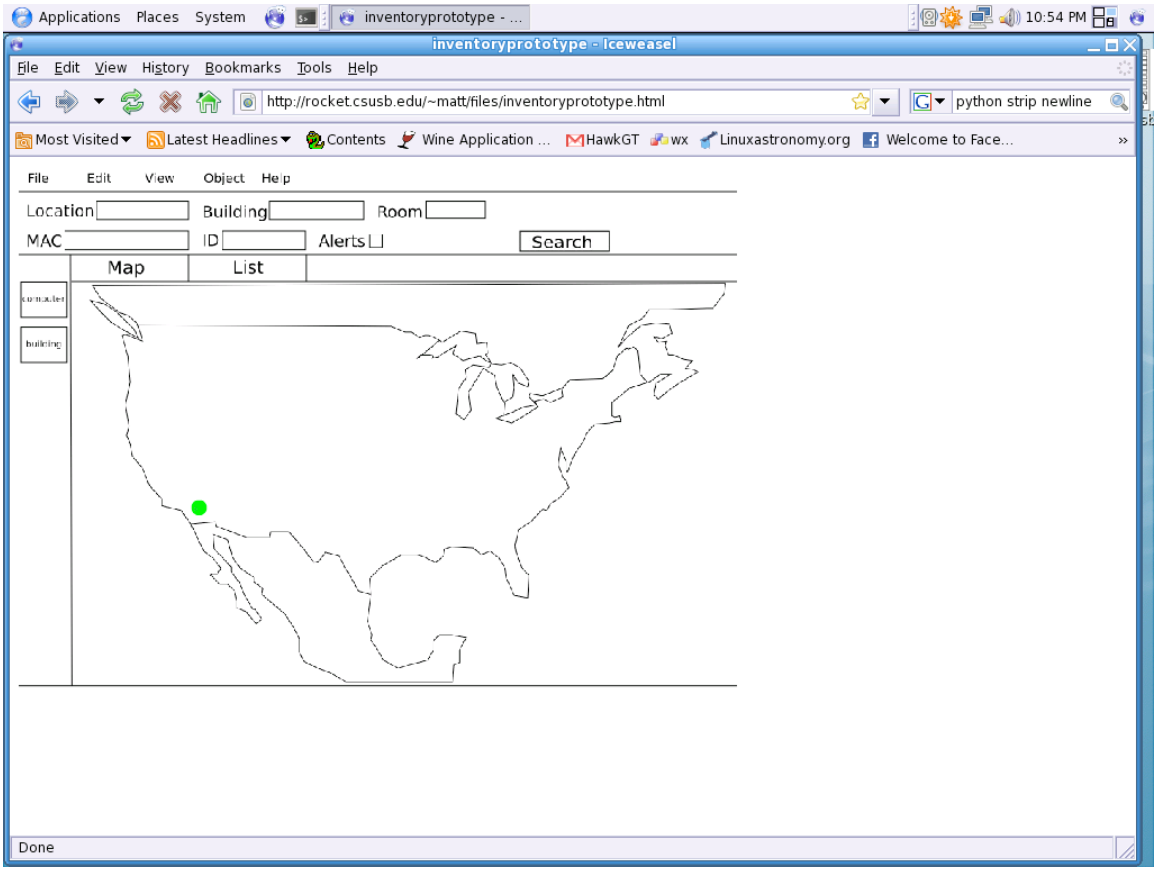

Fig. 3: Initial animation screen

the software design is working, see Fig. 3. The concept of event-driven programming is applied to an animation, so that a button press causes a jump to a point in the timeline where the result of the press is displayed. Testing the workflow becomes fairly easy.

Once the static images were set, they were imported into Flash and arranged in layers on the timeline. Invisible buttons were defined, placed over the previously

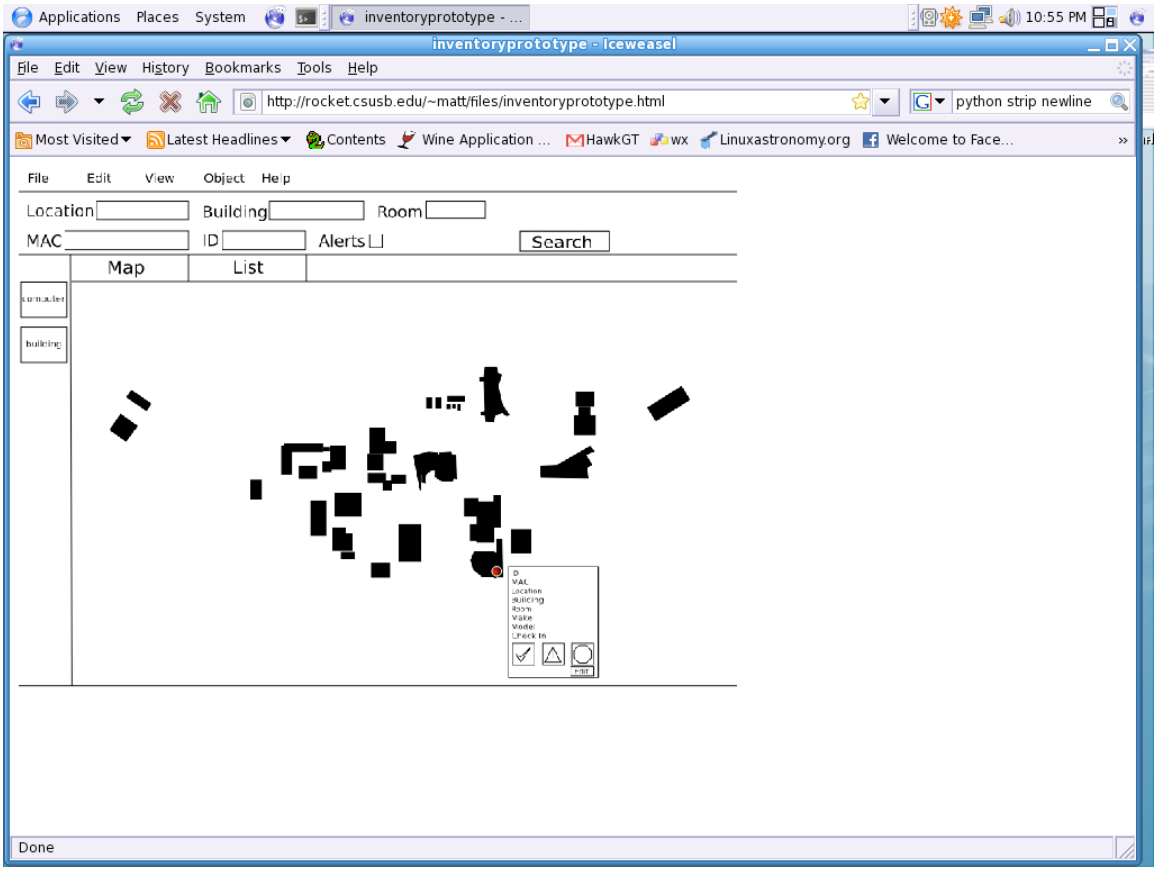

Fig. 4: The simulated search function

drawn ones. Events were predetermined to cut out the complexity of programming for real responses. For example, the text fields in the search area were just static boxes, but pressing the 'Search' button would result in the map panel being displayed, indicating where the computer was on the map and a dialog box populated with information relating to that search, see Fig. 4.

Development Tools

One of the goals of the project was that it be cross-platform, so that a user could run any of the pieces on Microsoft Windows, Apple OSX, or some other Unix variant like Linux. To this end, Python was chosen as the language in which all pieces would be written as it runs on all of the target platforms. The choice for an interface framework came down to PyXPCOM, Python bindings to the XPCOM framework used in Mozilla Firefox, and wxPython, Python bindings to the C++ wxWidgets framework. PyXPCOM offered a huge amount of features, but at the cost of a very steep learning curve and not much community support outside of the Mozilla project. wxPython 2.6 was chosen for providing a good feature set and quick development due to a relative lack of complexity. The software packages and versions used in this application are as follows:

- Python 2.5.2: interpreted programming language.
- CherryPy 3.0.2: web application framework.
- SQLObject 0.10.2: object-relational mapper.
- MySQL Server 5.0.51a: database engine.

Nuts and Bolts

Interface programming has a language all its own. The main window, or more correctly container, is a frame in

this case. A frame is the object that contains decorations such as a title bar, minimize and maximize buttons, a close button, and status bar. Within the frame are sizers that contain elements and take care of placing the elements within the frame. The alternative to sizers is the use of fixed coordinate positions, where each element is placed at a defined point. The main drawbacks to that method are that each element has to be placed individually and the positions stay fixed. Sizers adjust positions of elements automatically and can be told how to deal with a growing window, taking a lot of tedious coding away.

Each of the elements are referred to as widgets. Widgets can be acted upon in various ways, such as drawing on top of them, setting colors, and things of that nature. The main widget of the lower portion of the frame is a notebook. The notebook provides tabbed pages, where each page contains its own panel (a section within a frame.) The map tab panel loads a Portable Network Graphics image with a transparent background. The panel itself is colored white and set to grow as the window expands so that the default gray doesn't fill the remainder of the panel. The list tab contains a virtual list control that displays the results returned by a search. Initially, it displays the

entire database. As new searches are performed, the list display is updated to show the new results.

To the left of the notebook, a small button is placed in a panel. Clicking it pops up a dialog box for text entry. The dialog allows for a new entry or an update to an existing record. Each text box has a validation control associated with it to ensure that proper values are placed in the database.

Design Decisions

The layout has a left to right, top to bottom bias, much like every other interface. As Western written language is presented in this format, it is how most information is expected to be presented. However, this could be reversed with the addition of a method that reads a preference declaring a right to left layout. Then the elements-text, text boxes, buttons, and spacers-can be inserted in reverse order. The same can be done for the lower portion, with only a little more difficulty.

The sizer is based on columns, with the notebook taking four columns and the button panel one. Again, reversing the order of insertion would reverse the position of the elements. The additional difficulty would be in

ensuring that the proper column was set to grow as the window size grows.

Space was left in the button panel for the addition of map-making tools. Two different methods of loading maps are under consideration. The first is loading a pre-existing image file. This file would have to have certain properties, like a particular size and file type, i.e. 500 by 400 pixels in size and be a Portable Network Graphics (PNG) file type. The other method is to draw a map within the application. This would be saved to a file and loaded just as the pre-existing file would be.

For both methods, the placement of alert circles on the map would be determined by a mouse click on the building, which would capture the x and y coordinates and pop up a dialog where the user would enter the building name. This information would be written out to a file or to an additional table in the database for storage.

Time constraints have prevented the addition of this functionality. However, to make the application as portable as possible it would have to be added. For this iteration, the map was created by opening an official campus map in the Gnu Image Manipulation Program, selecting the buildings by color, and filling the selections with

black on a new layer. Then, the coordinates of each building were determined by mousing over the building and reading the details from the status bar. It was then saved as a PNG with a transparent background. The background color of the notebook panel it gets loaded into was set to white, so that the panel remains white as the window grows without having to create an arbitrarily large image file.

Because the coordinates are specific to the panel the object is in-(0,0) is at the top left of the particular panel-collecting coordinates from the image file works fine, if not a little tedious. However, if scaling is required to fit the image to the panel, the objects will move accordingly and the coordinates gathered rendered invalid.

For this iteration, a dictionary uses building names as keys and the coordinates as values. When the alert toggle button is depressed in a search, the results are parsed for building names. For each name found, a device container object is created from which a circle can be drawn on the map at the appropriate location. If a building turns up more than once, multiple circles get drawn at the same point, but in the exact same place, which doesn't matter as far as the user is concerned. The

circles are not persistent and disappear as soon as the screen is redrawn in that area, whether through switching to the list tab or even dragging a completely different window over the area. This is a feature of the device context object and a behavior that seemed desirable as the alerts are only meant to give a quick visual reference that a problem exists in a particular location.

The list that is displayed in the list tab is colored red when alerts are returned in a search. This should not present a problem for colorblind users as there is still sufficient contrast with the background and it is an additional cue, not the sole means of communicating a status. For localization, changing the color for both the list and the alert circle could be done through a preference, but for the moment is coded in.

As for color, system defaults are used for everything not in the notebook. This keeps the look consistent with the rest of the desktop environment the user has set up. Additionally, it saves time in determining appropriate color schemes for different locales.

Conclusion

Rendering the GUI in a dedicated drawing program and then testing the workflow by simulating the software in an animation makes for a rapid prototyping process. Ideas can be quickly tested and rejected, sent to others for comments, posted on the web for testing, etc. So in that sense, the process results in a useful method of communicating ideas and a vision of how the final product will look.

There were a few caveats. Because it is so easy to knock up an idea, it is easy to include features that may be difficult or impossible to implement. For example, an early idea was to incorporate the Google Maps API into the map panel. The GUI toolkit used in the project, wxPython, supports HTML but does not support more advanced features found in web browsers needed to run Google Maps. The other problem encountered was the addition of features that would later prove to be outside of the scope of the application, i.e., the 'Building' button that was intended for the addition of properties to a building object that never materialized. The creation of custom maps within the application was an early desire, but would have involved

the implementation of a drawing program within the overall application.

CHAPTER FOUR

SOFTWARE DESIGN

Introduction

The first version of PySafe followed a very basic structure: a client sent strings of data to a server; the server stored the data in a database; data was retrieved through a rudimentary interface to the database. The server played the Controller role in the MVC pattern, but the View role was not implemented properly. Of course, the huge problem with this structure is the protocol used to send data from the client to the server. In effect, a custom protocol was implemented that sent strings, generated from converting an object into a string describing its attributes (called 'pickling'), of a certain length from the client to a socket on the server, which then unpickled the object and stored the attributes in the database.

The list of problems this simple set up presents is quite long. The potential for a security compromise was huge. While the amount of connections at any one time would probably be small, the socket server would not scale well. The lack of standard protocols would make extending

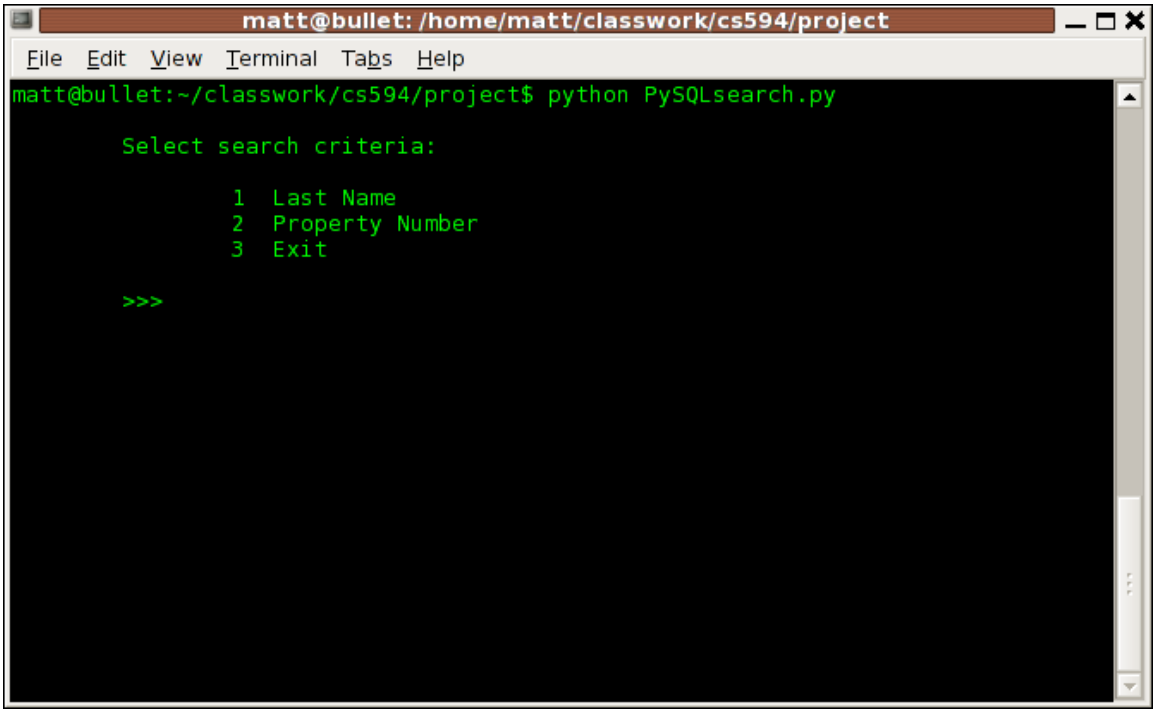

Fig. 5: An early prototype

and maintaining the application troublesome. It looked ugly (see Fig. 5), which may seem trivial but having a certain elegance makes working with an application more appealing.

With that as a basis, the original pattern was retained and the code rewritten. All components were written in Python, as simplicity trumped speed and a large number of useful libraries were available. Python also has the benefit of good Unicode support, allowing for localization of strings by declaring the encoding in the

file. The code is split into separate files based upon roles and responsibilities as determined by the application of the Model-View-Controller (MVC) design pattern. The MVC pattern separates the interface (the View) from the persistent storage of data, using an intermediary that handles requests and responses (the Controller.) The first version of PySafe partially implemented this with a simple server receiving data from a remote computer and storing it in a database; however, the interface communicated directly with the database.

A search initiated in the user interface would look like this:

- Search terms entered and Search button pressed.
- Parameters sent to mcp.py.
- mcp.py sends the parameters via HTTP POST to a specific URL in inventory.py.
- inventory.py sends the parameters to the database via store_data.py.
- store_data.py returns the SQL results to inventory.py.
- inventory.py sends an XML formatted response to mcp.py.

- mcp.py returns a list generated from parsing the Document Object Model of the XML to the interface.
- The Virtual List Control updates the contents of the list with the results.

Pattern

The MVC pattern divides code into three areas of responsibility. The Model is the business model reflected in the database design. What and how data gets stored is determined by this persistence layer of the application. The View is responsible for presentation of the data sent by the Controller and communicating user inputs back to the Controller. The Controller acts as a gateway between the Model and View, handling requests and submissions from the View and then sending appropriate commands to the Model.

PySafe is composed of five elements in implementing this pattern: the client for the reporting computer; a web application server; an object relational manager and database; an administrator-side client; and the graphical user interface, see Fig. 6.

The Client

On the reporting computer is a script, computer.py, that gathers the Media Access Control (MAC) address and a

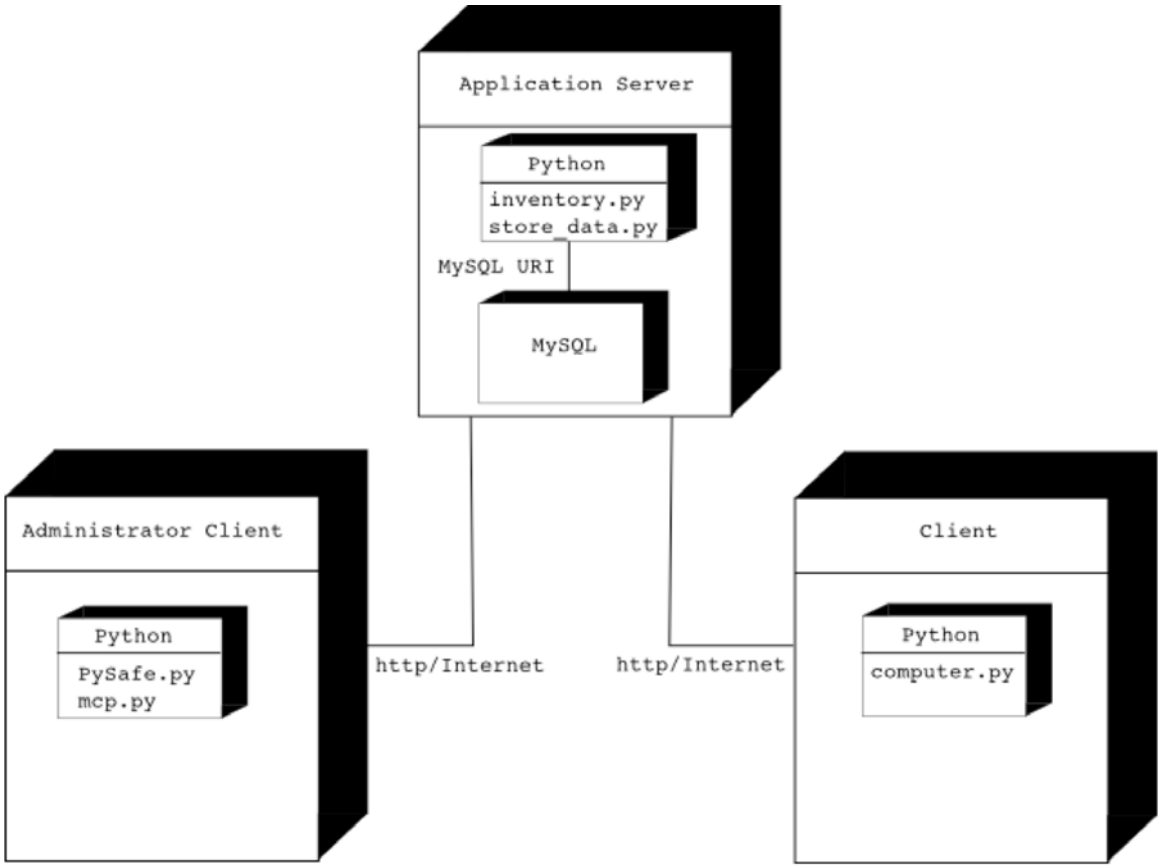

Fig. 6: Deployment diagram

traceroute dump, and then sends this data via an HTTP POST to the web application server. It does this by first determining the platform, Apple, Windows or Linux, then calling the appropriate system commands and storing the responses. The urllib library is used to emulate a browser and submit the information. This can be set up to run at system start up or at a scheduled time, whichever the system administrator prefers.

The operating system (OS) is determined by a class, MyOS, that is used in the Information Expert pattern. The single attribute of the class is set to an instance of one of the three classes implemented to represent the three operating systems. Each of the OS classes have the same methods and attributes, which takes advantage of Python's duck typing, allowing for polymorphism without inheritance. Duck typing is a means of determining an object's type by seeing if the method being called on it exists. If the method is implemented, then it is assumed to be of the same type and the procedure continues, otherwise a type error is raised. In this manner, a parent class and inheritance hierarchy is not necessary, as new classes only have to implement the same methods and attributes.

The Web Application Server

As HTTP is a robust protocol, it was chosen as the means of communication between both clients. The CherryPy framework provided the basis for the web application server as it not only provides a standalone web server, but also can be proxied behind a more robust HTTP server such as Apache. That has the advantage of providing secure communications if desired by taking advantage of the SSL capabilities of Apache.

The application server, executed by inventory.py, takes the data submitted by the client computer and calls the storage module, which places it in the database. From the administrator-side client, it takes requests for information and calls the storage module. It also submits new data and updates to existing data. To return data to the administrator component, database results are formatted as XML. The data extracted from the XML string is in Unicode.

Persistent Storage

The database engine is MySQL, chosen mostly for its popularity, which translates to lots of community support, and familiarity. SQLite may be even more appropriate, as the database consists of a single table with an expected maximum number of roughly 500 records. To manage and interact with the database, the SQLObject library was chosen as it provides an object-oriented way of dealing with data and a common interface to many different database engines. This layer of abstraction makes it easy to switch database engines in the future if necessary. The store_data.py module contains the connection information required to sign into the database and all methods and functions needed to interact with it.

Fig. 7: Initial PySafe window

The Administrator Component

This module keeps the application logic out of the GUI code. All of the administrator-side queries and submissions are called from this component by the GUI. This portion, mcp.py, sends requests and data via HTTP POST

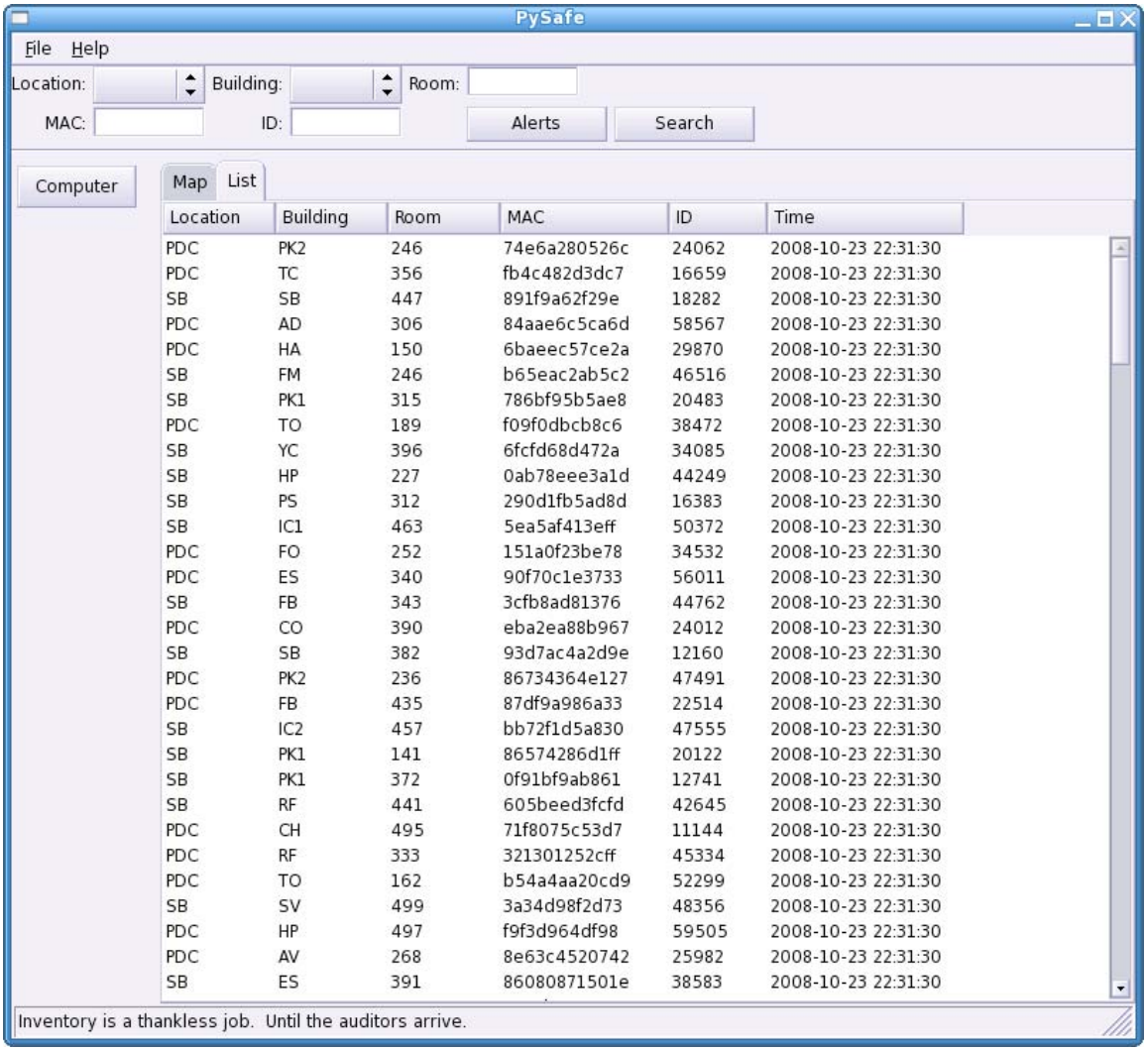

Fig. 8: PySafe list view

and receives data as XML, which then gets converted to lists and sent to the GUI.

All of the code used to generate the window, frame, panels, buttons, text entry boxes, lists and other graphical elements are in pysafe.py (Fig. 7).

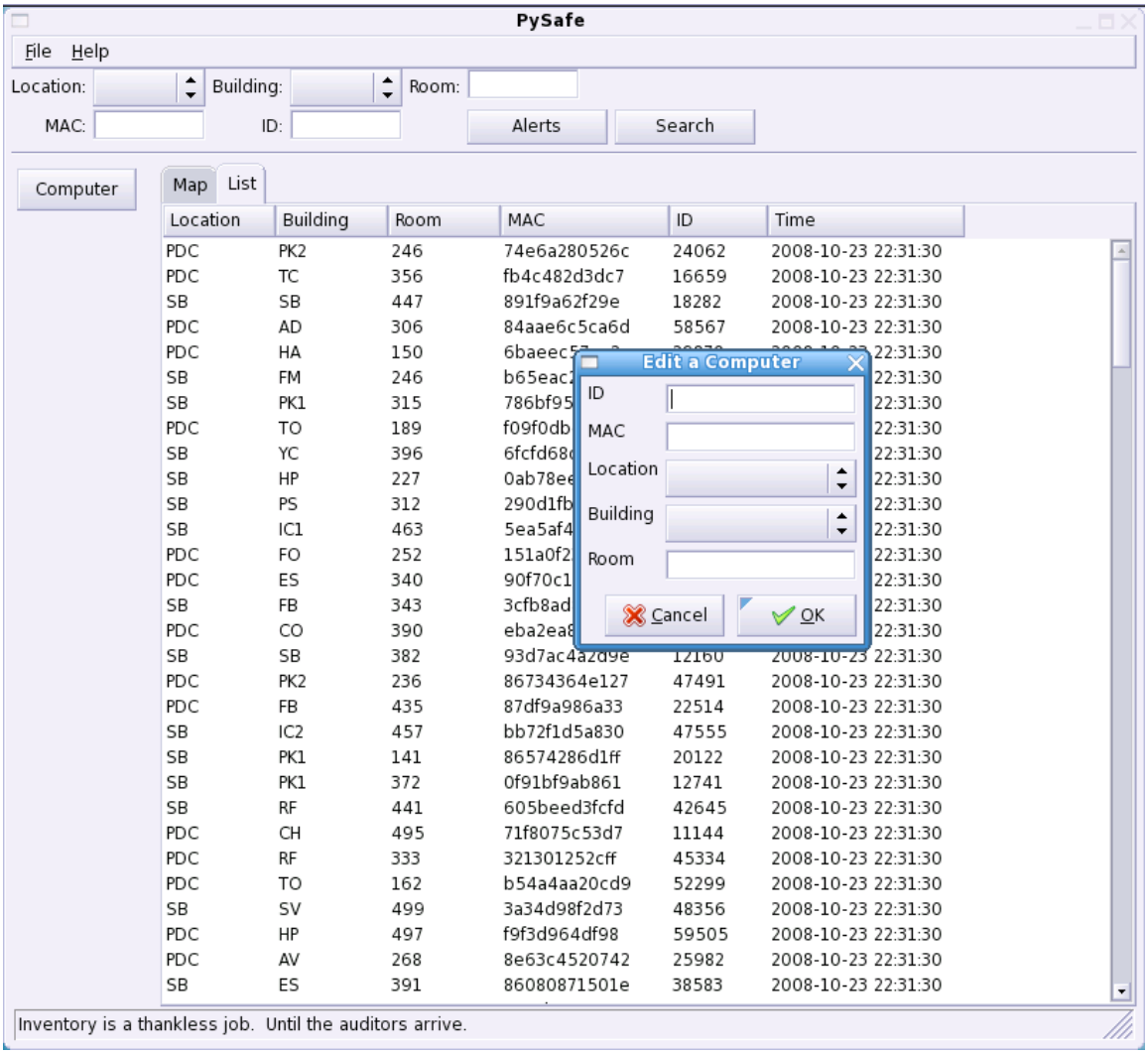

Fig. 9: PySafe edit view

The list tab of the notebook displays search results (Fig. 8). If alerts are returned, the items in the list are colored red. Upon new searches, the list is updated with the new results and clears other items no longer valid.

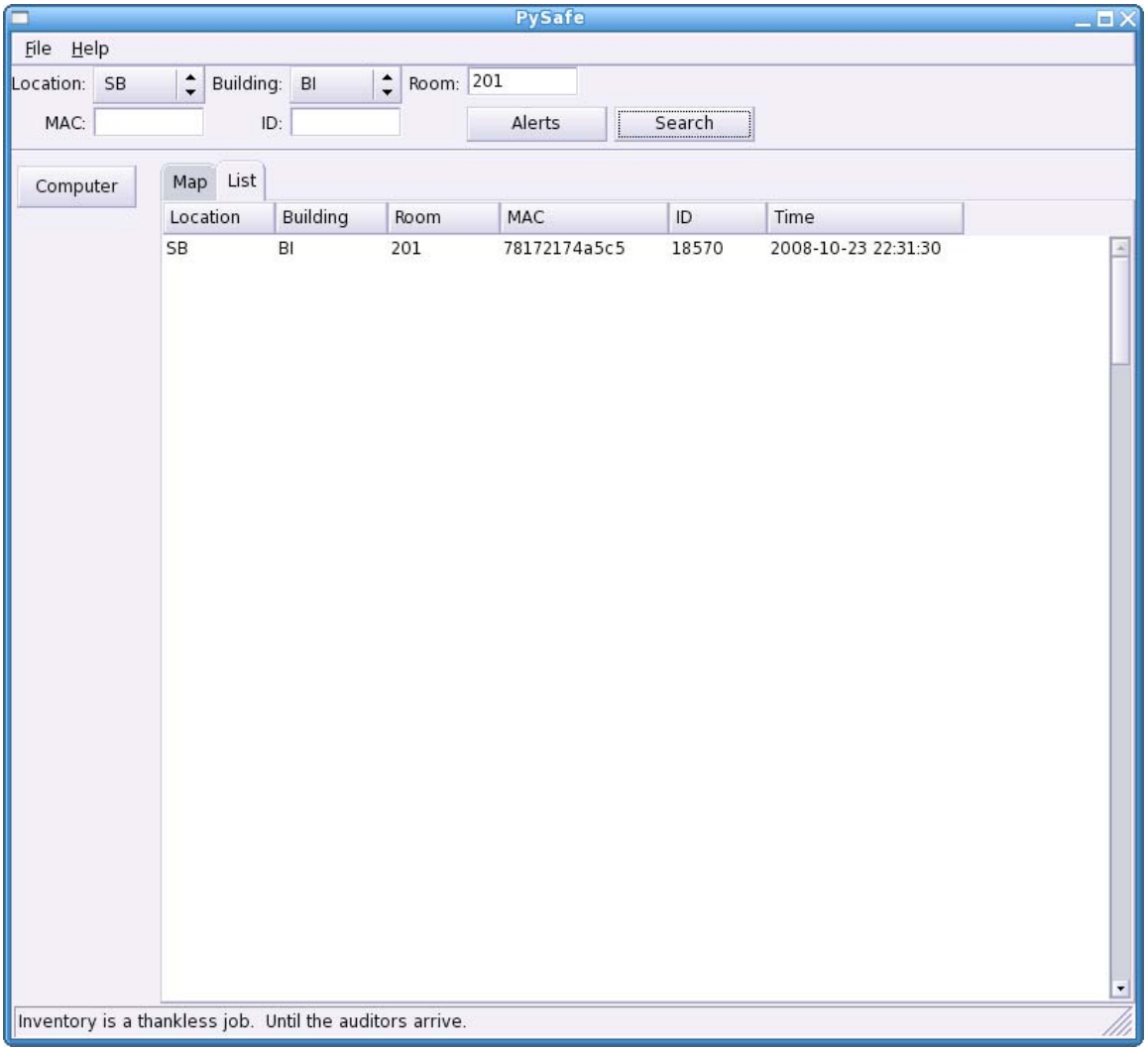

Fig. 10: PySafe search view

Validation checks on data submitted via dialogs is performed within the GUI code (Fig. 9). Each box is checked for appropriate values, with an error message and visual cue (the text box is colored pink) to indicate a problem and what needs to be done to fix it. Drop down

Fig. 11: PySafe alert view

boxes are used to ensure that only valid choices can be entered for items that need to be encoded, such as the "Location" and "Building" boxes.

The search bar takes on much of the same behavior as the edit dialog, with the exception of validation checks

(Fig. 10). If invalid data is entered, no results are displayed in the list tab. The database cannot be fed invalid values from this function, so no harm can come from a lack of checks. A tradeoff between potentially irritating validation errors and the implicit warning of zero results was made, in favor of less irritation.

The alert toggle button, when depressed, sets a flag in the parameters dictionary indicating that the SQL statement that compares the dates in the table to the current time should be executed. If there are results, the list returned to the interface for display is parsed for building names. For each name found, a Device Context object draws a red circle on the map at the coordinates set for that building (Fig. 11).

Installation

The computers used with the application must have the Python interpreter installed. It is included in Apple's OS X and most Linux distributions. For Windows, the interpreter can be downloaded from http://python.org. For all systems, install the Python interpreter and then the additional Python packages (CherryPy, SQLObject, and wxPython).

The application server will need in addition to the interpreter the MySQL database engine, available from http://www.mysql.com. It will also need CherryPy (http://www.cherrypy.org) and SQLObject (http://www.sqlobject.org/). Once the packages are installed, the database needs to be prepared.

Create a new database called ‘tracker.’ The connection details are contained in store_data.py, in the connect_db() function. If a different name for the database is to be used, specify it in the ‘resource’ string. The database user name and password are also in the string, whatever values used here must match with the values used in the creation of the database.

For example, the following lines would create a root password for the database, sign the root user into the database, create the database needed for PySafe, and grant all rights to the account used by store_data.py to connect:

- > mysqladmin –u root password ‘somepassword’
- > mysql –u root –p
- > create database tracker;
- > grant all on tracker.* to ‘user’@’localhost’ identified by ‘someotherpassword’;

The "resource" string in store_data.py will look like this: 'mysql://user:somepass@localhost/tracker'.

Copy the files inventory.py, store_data.py, cherrypy.config, and pysafe.config to a directory like /var/webapps. Open cherrypy.config and change the log.error_file path to something reasonable on your system, socket_port to the port you want the service to run on and socket_host to the hostname or IP address of the system. In pysafe.config, change tools.staticdir.root to the path that installed inventory.py to.

Make sure that inventory.py and store_data.py are executable, then run store_data.py. This will create a table in the 'tracker' database made earlier. Then execute inventory.py, which will start the application server.

On the client computer, things get a little trickier.

For Linux, a script needs to be written to run computer.py after networking is set up or by cron at some specified interval. The shell script goes in the /etc/init.d directory. Each distribution does things a little differently, so check out one of the other scripts in that directory to see what options are typical, i.e. start, stop, restart, and how to implement them. It may be

easier to copy an existing one and then just have it call computer.py instead of whatever it originally called.

Next, go to the /etc/rcS.d directory and create a symbolic link to the script placed in /etc/init.d. Name the link S##yourscriptname, where ## is a number representing the order in which it will be run. For example, this script depends on networking, so it would need a number higher than the networking script.

Normal procedures apply: make the script executable, change the ownership to root if necessary, etc. Test it, then test it again. Messing up the boot sequence requires booting into single user and fixing it.

Apple is similar, but of course uses a different folder structure. You will need to be root to do many of these steps. Copy a folder from /System/Library/StartupItems to /Library/StartupItems. Apple-installed stuff goes in the /System/Library, all of the local administrative stuff should go in /Library. Pick one that is similar to what computer.py will do, i.e., a script that uses some network resources. Change the name of the folder to computer.py. Edit the StartupParameters.plist file so that the values make sense for this service. Changes to the Description, Provides,

Requires, Uses, OrderPreference, and Messages values will be necessary. It might be an XML file or a text file, but the keys are the same. Make sure it's executable and owned by root. If the group ownership is set to the admin group and execute privileges are also set, then admin users can run the script while the system is running.

Windows startup and shutdown scripts are run after networking has been established and before networking is shut down, respectively. Place computer.py in the appropriate folder, c:\\WINDOWS\system32\GroupPolicy\Machine\Scripts\Shutdown or Startup. Run gpedit.msc from the Run prompt, expand Computer Configuration -> Windows Settings, and then click on Scripts (Startup/Shutdown). Double click on the appropriate one, click 'add' in the dialog box that appears, then browse to computer.py.

To make Python scripts run without having to enter 'python' at the prompt, the Python executable folder needs to be appended to the PATH variable. To do this, right click on My Computer->Properties->Advanced->Environment Variables, highlight the PATH entry and add the path to the Python installation folder to the end.

The administrator's computer, the one that will run the interface, needs wxPython (http://www.wxpython.org/).

Put pysafe.py and mcp.py somewhere in the search path, like /usr/local/bin, create a desktop launcher for pysafe.py and it is ready to go.

Maintenance

Very little maintenance is required, amounting to keeping a back up of the database. There are a million and one methods of doing that. One simple method is to dump the database with mysqldump, compress the file using gzip, and transfer it to another computer. If database corruption occurs, rerun store_data.py to dump the table and recreate it, then reload the previously saved database dump.

CHAPTER FIVE

TESTING

Introduction

While the focus of this project was on the development of a good, solid, usable interface, a solid foundation had a high priority. A good interface coupled with bad software is just another useless application. With that in mind, each component was subjected to a test suite designed to emulate normal use and determine if the component responded properly or failed. During the course of development, it did not always seem necessary to produce specific tests; however, following through revealed subtle bugs that may not have affected one component but certainly would cause others to fail once propagated, especially if it reached the interface. The message was clear: don't skip the testing.

Methods

Two different approaches were used in testing the application. The interface was acceptance tested through use and by examining debugging output by the administrator-side client and the database. For the other components,

unit test scripts were written to automate the process. As bugs were discovered and squashed, tests were rerun for a rudimentary regression test.

Component Tests

Each module had a corresponding set of unit test scripts written for it. In the test script, each function or method of the module was called as if it were being invoked by a proper part of the application. The client was tested by running it and checking the values it sent to the application server. For the application server, functions were written to emulate the client and the administrator modules and checking that responses were correct and data was sent properly to the database. The administrator functions were tested by submitting requests to the application server and printing the results to standard output. The database module was tested by generating random data with valid values and performing insertions, updates, and selects on the database. Valid values were picked as it was revealed during development that strings longer than the length parameter were truncated and shorter ones were inserted. With data validation being handled at the source of input, it didn't make sense to input values that were not valid, such as tag

numbers with characters and digits or MAC addresses with characters beyond ‘f.’

Interface Tests

Each event was tested by stepping through the process of using the interface to achieve a goal, such as selecting by building, and determining if the result was the expected one. The layout was tested in the design portion and remained more or less fixed.

Summary

The testing phase revealed some subtle bugs that could have been showstoppers. The biggest was the default values of an empty string being placed in the database when a client checked in without having an existing entry. The client check in would succeed, but the application would crash when it hit the DOM parser in mcp.py. The XML would be valid, but without any data between the first tags an exception would be thrown. It turned out that allowing the empty string to pass on to the virtual list control would cause that to crash. The fix was to store a string of a single space character in the database as a default value. It was these sorts of conditions that were not addressed during development that the test suites were invaluable

for. As stated previously, a good interface is dependent on a solid foundation. As far as the person using it is concerned, if it crashes, the whole program is bad.

Future Work

This is an area of the application that would benefit greatly from some enhancements. Two improvements to the testing system come to mind. The first is a better regression test system and the second is a means of automatically testing the interface. The former may be accomplished through the use of an established testing framework. The latter seems to be a much more difficult task as many methods exist, but are often tied to manipulating the underlying window manager to simulate events.

CHAPTER SIX

CONCLUSIONS

Review

The project consists of two parts: a literature review of interface design and an application designed with the principles gleaned from the review at the fore. Many disciplines have been involved with human-computer interaction, to include computer science, cognitive psychology, and cultural studies, among others. The trend over the years has been to move toward understanding how people interact with computers and giving significant consideration to that in the design.

Gaining a historical perspective through the literature review, some issues became quite apparent. Advocacy for the people component of the interface has been around for a long time. Its voice was not very loud, however, until personal computing became mainstream and the numbers of people affected grew large. With the addition of the cognitive psychology perspective to the body of research, better methods of accounting for the inherent limitations of people were established. This really

signaled a major shift away from adapting users to a system and toward adapting the system to better fit people.

The growth of cheap, high-speed data communications networks has made the world a smaller place in many ways. Businesses can operate around the clock, with locations around the world. People have the ability to connect with people in other countries from the comfort of their homes. This global communication necessitates the need for cultural awareness in order to reduce conflict and promote better communication. It is true for design, as well, when the interface is considered as communication between the designer and user. A basic understanding that what goes in one culture may not go in another could be the difference between something useful to a small audience and one useful to a very large audience. Taking advantage of internationalization and localization efforts and relying less on color and symbols to convey meaning was the method used in this project toward that goal.

Design first can work. While PySafe was built on the ashes of a former incarnation, it grew out of an interface-first approach. Decisions about what the overall activity was and the tasks required to achieve it were resolved before coding began. That resulted in a back end that was

focused on providing the functionality needed to perform the activity without straying from the goal. While PySafe is a rather small application (a good deal of the source code is dedicated to the graphical interface) with a do-one-thing approach, the development method is applicable to larger products.

Future Work

It is said that software is never finished: only too true with PySafe. Once a working application was realized, some additional features came to be desired. A method of importing or creating maps and being able to map alerts to locations by point and click would be welcome. Report printing from lists generated by searching, a means of running computer.py at startup for Windows clients, and an installer for all three components are under consideration. And, perhaps most importantly, documentation and a useful help facility within the graphical interface.

As for research, more study in semiotics and cultural studies are called for. There seems to be great benefit to be had from gaining insights into better methods of communication, whether through language or symbols. Of

course, more design work needs to be done as that is the best way to get better at it.

APPENDIX A

SOURCE CODE

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

"""computer.py - determines system type, then gathers the MAC address
   and traceroute dump.
"""

import os
import time
import platform
import re
import urllib

class MyOS(object):

    def __init__(self, host, timeLimit):
        if platform.system() == 'Linux':
            self.ostype = Linux(host, timeLimit)
        elif platform.system() == 'Darwin':
            self.ostype = Apple(host, timeLimit)
        elif platform.system() == 'Windows':
            self.ostype = Windows(host, timeLimit)
        else:
            self.ostype = None

class Apple:

    def __init__(self, host, timeLimit):
        self.setMAC()
        self.setTrace(host, timeLimit)

    def setMAC(self):
        """Retrieve MAC address, set object variable (Apple)"""

        f = os.popen('/sbin/ifconfig')
        data = f.readlines()
        f.close()
        pattern = '\tether [0-9a-z:]*'

        results = [re.findall(pattern, item) for item in data
                   if re.search(pattern, item)]

        maclist = [address.split() for item in results
                   for address in item]
        self.mac = maclist[0][1].replace(':', '')

    def setTrace(self, host, timeLimit):
        """Get traceroute output, set trace variable"""

        f = os.popen('/usr/sbin/traceroute %s' %host)
        time.sleep(timeLimit)
        g = os.popen('/usr/bin/killall traceroute')
        tracert = ''
        for line in f.readlines():
```

```
            tracert += line
        f.close()
        g.close()
        self.trace = tracert.replace('\n', ' ')


class Linux:

    def __init__(self, host, timeLimit):
        self.setMAC()
        self.setTrace(host, timeLimit)

    def setMAC(self):
        """Retrieve MAC address, set object variable (Linux)"""

        f = os.popen('/sbin/ifconfig eth0')
        data = f.readlines()
        f.close()

        sdata = data[0].split()
        self.mac = sdata[4].replace(':', '').lower()

    def setTrace(self, host, timeLimit):
        """Get traceroute output, set trace variable"""

        f = os.popen('/usr/bin/traceroute %s' %host)
        time.sleep(timeLimit)
        g = os.popen('/usr/bin/killall traceroute')
        tracert = ''
        for line in f.readlines():
            tracert += line
        f.close()
        g.close()
        self.trace = tracert.replace('\n', ' ')


class Windows:

    def __init__(self, host, timeLimit):
        self.setMAC()
        self.setTrace(host, timeLimit)

    def setMAC(self):
        """Retrieve MAC address, set object variable (Windows)"""
        results = []
        f = os.popen('ipconfig.exe /all')
        data = f.readlines()
        f.close()

        pattern = '^ *Physical Address.*'

        results = [re.findall(pattern, item) for item in data
                   if re.search(pattern, item)]
```

```
        maclist = [address.split() for item in results
                   for address in item]
        self.mac = maclist[0][-1].replace('-', '').lower()


    def setTrace(self, host, timeLimit):
        """Get tracert output, set trace variable"""

        f = os.popen('tracert.exe %s' %host)
        time.sleep(timeLimit)
        g = os.popen('taskkill /F /IM tracert.exe')

        tracert = ''
        for line in f.readlines():
            tracert += line
        f.close()
        g.close()
        self.trace = tracert.replace('\n', ' ')


class Computer(object):

    def __init__(self, host, timeLimit):
        myOS = MyOS(host, timeLimit)
        self.status(myOS.ostype.mac, myOS.ostype.trace)

    def status(self, mac, trace):
        self.params = {'mac' : mac,
                       'trace' : trace
                      }


def main():
    try:
        home = "http://yourserver.com:8080/inDB"
        host = 'somehost.com'
        timeLimit = 15
        comp = Computer(host, timeLimit)
        parameters = urllib.urlencode(comp.params)
        f = urllib.urlopen(home, parameters)
        print f.read()
    except:
        pass

if __name__ == '__main__':
    main()
```

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

"""inventory.py : Collect data from POST submitted by client,
   then update information in database.
"""

import cherrypy
import os
import datetime
import store_data

class InputData:

    #@cherrypy.expose
    def index(self):
        """ Uncomment the decorator if you want to use an html page to
            test submissions to the database."""

        return '''
            <form action="inDB" method="POST">
            <input type="text" name="mac" />
            <input type="text" name="trace" />
            <input type="submit" />
            </form>'''

    @cherrypy.expose
    def inDB(self, mac = None, trace = None):
        if mac and trace:
            store_data.connect_db()
            store_data.update_by_client(mac, trace)
        else:
            f = open("send_error.log", "a")
            f.write("%s check in failed" % datetime.datetime.now())
            f.close()

    @cherrypy.expose
    def getRecords(self, **kwargs):
        """Results returned as xml"""
        store_data.connect_db()
        #params = kwargs
        results = store_data.get_records(**kwargs)
        xmldoc = ""
        row_xml = """<row>
                     <location>%s</location><building>%s</building>\
                     <room>%s</room><mac>%s</mac>\
                     <ptag>%s</ptag><time>%s</time></row>"""
        for i in results:
            xmldoc += row_xml % (i.location, i.building, i.room, i.mac,
                                 i.ptag, i.timeIn)
        return "<results>" + xmldoc +"</results>"

    @cherrypy.expose
    def updateRecords(self, **params):
```

```
        store_data.connect_db()
        store_data.update_by_mcp(**params)

    @cherrypy.expose
    def count(self):
        store_data.connect_db()
        rows = store_data.count()
        return "<results><count>%i</count></results>" % rows

appconf = os.path.join(os.path.dirname(__file__), 'pysafe.config')
cherryconf = os.path.join(os.path.dirname(__file__), 'cherrypy.config')
cherrypy.config.update(cherryconf)
cherrypy.quickstart(InputData(), '/', appconf)



#!/usr/bin/python
# -*- coding: utf-8 -*-

"""inventoryTracker-gui.py: the launcher for the
   graphical interface."""

import wx
import mcp

class MyFrame(wx.Frame):

    def __init__(self, title, pos, size):
        wx.Frame.__init__(self, None, -1, title, pos, size)

        self.dataSource = DataSource()
        self.locationlist = [' ', 'SB', 'PDC']
        self.buildinglist = [' ', 'AD', 'AF', 'AH', 'AS', 'AV',
                        'BI', 'BK', 'CC','CE', 'CH', 'CO', 'CS',
                        'DD', 'ES', 'FB', 'FM', 'FO','HA', 'HC',
                         'HP', 'IC1', 'IC2', 'JB', 'PA', 'PE', 'PK1',
                        'PK2', 'PL', 'PS', 'PW', 'RF', 'SB', 'SH',
                        'SU', 'SV','TA', 'TC', 'TK', 'TO', 'UH',
                         'UP', 'UV', 'VA', 'YC']
        menuBar = wx.MenuBar()
        menuFile = wx.Menu()
        menuFile.Append(1, "Q&uit")
        menuBar.Append(menuFile, "&File")

        #menuEdit = wx.Menu()
        #menuEdit.Append()
        #menuBar.Append(menuEdit, "&Edit")

        #menuView = wx.Menu()
        #menuView.Append()
        #menuBar.Append(menuView, "&View")

        #menuObject = wx.Menu()
        #menuObject.Append()
```

```
        #menuBar.Append(menuObject, "&Object")

        menuHelp = wx.Menu()
        menuHelp.Append(2, "&About")
        menuBar.Append(menuHelp, "&Help")

        self.SetMenuBar(menuBar)

        self.CreateStatusBar()
        statusMsg = """Inventory is a thankless job.
                       Until the auditors arrive."""
        self.SetStatusText(statusMsg)

        self.Bind(wx.EVT_MENU, self.OnQuit, id=1)
        self.Bind(wx.EVT_MENU, self.OnAbout, id=2)

        self.notebook = self.createNotebook()
        self.createPanel()

    def OnQuit(self, event):
        self.Close()

    def OnAbout(self, event):
        DisplayText = """PySafe shows when a computer last \
checked in and helps you determine if you need to go check \
on one of your pies."""
        wx.MessageBox(DisplayText, "About PySafe", \
                       wx.OK | wx.ICON_INFORMATION, self)

    def createNotebook(self):
        nb = wx.Notebook(self)
        self.mymap = MyMapDisplay(nb)
        self.mylist = MyListDisplay(nb, self.dataSource)
        nb.AddPage(self.mymap, "Map")
        nb.AddPage(self.mylist, "List")
        return nb

    def createPanel(self):
        self.searchPanel = SearchCriteriaPanel(self, self.locationlist,
                                               self.buildinglist)
        graphicPanels = GraphicPanels(self, -1)
        sizer = wx.GridBagSizer(hgap=1, vgap=1)
        sizer.Add(self.searchPanel, pos=(0,0), span=(1,5),
flag=wx.EXPAND)
        sizer.Add(graphicPanels, pos=(1,0), flag=wx.EXPAND)
        sizer.Add(self.notebook, pos=(1,1), span=(1,4), flag=wx.EXPAND)
        sizer.AddGrowableCol(1)
        sizer.AddGrowableRow(1)
        self.SetSizer(sizer)


class SearchCriteriaPanel(wx.Panel):

    def __init__(self, parent, locations, buildings, ID=-1,
```

```
        pos=wx.DefaultPosition):
wx.Panel.__init__(self, parent, ID, pos)



self.locChoice = ' '
location = wx.StaticText(self, -1, "Location:")
self.loc = wx.Choice(self, -1, choices=locations)
self.loc.SetSelection(0)
self.loc.Bind(wx.EVT_CHOICE, self.OnLocChoice)

self.buildChoice = ' '
building = wx.StaticText(self, -1, "Building:")
self.build = wx.Choice(self, -1, choices=buildings)
self.build.SetSelection(0)
self.build.Bind(wx.EVT_CHOICE, self.OnBuildChoice)

room = wx.StaticText(self, -1, "Room:")
self.rm = wx.TextCtrl(self, -1)
self.rm.SetInsertionPoint(0)

mac = wx.StaticText(self, -1, "MAC:")
self.mc = wx.TextCtrl(self, -1)
self.mc.SetInsertionPoint(0)

idtxt = wx.StaticText(self, -1, "ID:")
self.idctrl = wx.TextCtrl(self, -1)
self.idctrl.SetInsertionPoint(0)

self.alert = 0
self.alertsButton = wx.ToggleButton(self, -1, "Alerts",
                                    size=(100,25))
self.alertsButton.SetToolTip(wx.ToolTip("Search for overdue
                             checkins"))
self.alertsButton.Bind(wx.EVT_TOGGLEBUTTON,
                       self.alertButtonClick)


searchButton = wx.Button(self, -1, "Search", size=(100,25))
searchButton.Bind(wx.EVT_BUTTON, self.searchButtonClick)

mainSizer = wx.BoxSizer(wx.VERTICAL)

searchSizer = wx.FlexGridSizer(cols=7, hgap=5, vgap=2)
searchSizer.Add(location, 0,
                wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
searchSizer.Add(self.loc, 0)
searchSizer.Add(building, 0,
                wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
searchSizer.Add(self.build, 0)
searchSizer.Add(room, 0,
                wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
searchSizer.Add(self.rm, 0)
searchSizer.Add((10, 10))
```

```
        searchSizer.Add(mac, 0,
                        wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
        searchSizer.Add(self.mc, 0)
        searchSizer.Add(idtxt, 0,
                        wx.ALIGN_RIGHT|wx.ALIGN_CENTER_VERTICAL)
        searchSizer.Add(self.idctrl, 0)
        searchSizer.Add((10, 10))
        searchSizer.Add(self.alertsButton)
        searchSizer.Add(searchButton)

        mainSizer.Add(searchSizer, 0, wx.EXPAND)
        mainSizer.Add(wx.StaticLine(self), 0,
                      wx.EXPAND|wx.TOP|wx.BOTTOM, 5)

        self.SetSizer(mainSizer)
        mainSizer.Fit(self)

    def OnLocChoice(self, event):
        self.locChoice = event.GetString()

    def OnBuildChoice(self, event):
        self.buildChoice = event.GetString()

    def alertButtonClick(self, event):
        self.alert = self.alertsButton.GetValue()

    def searchButtonClick(self, event):

        room = self.rm.GetValue()
        mac = self.mc.GetValue()
        pid = self.idctrl.GetValue()

        searchDict = {}

        if self.locChoice != ' ':
            searchDict['location'] = self.locChoice
        if self.buildChoice != ' ':
            searchDict['building'] = self.buildChoice
        if room != "":
            searchDict['room'] = room
        if mac != "":
            searchDict['mac'] = mac
        if pid != "":
            searchDict['ptag'] = pid
        if self.alert == 1:
            searchDict['alert'] = 1

        prev = app.frame.dataSource.GetCount()-1
        app.frame.dataSource.GetResults(**searchDict)
        new = app.frame.dataSource.GetCount()
        app.frame.mylist.list.SetItemCount(new)

        #last = max(prev, new)
        last = new - 1
```

```
        print 'last', last
        app.frame.mylist.list.RefreshItems(0, last)

        coords = { 'AD': (289,248), 'AF': (96, 125), 'AH': (289,119),
                   'AS': (115,187), 'AV': (560,340), 'BI': (334,144),
                   'BK': (321,311), 'CC': (234,265), 'CE': (427,142),
                   'CH': (285,285), 'CO': (455,284), 'CS': (302,145),
                   'DD': (248,251), 'ES': (112,131), 'FB': (202,236),
                   'FM': (94, 152), 'FO': (326,227), 'HA': (284, 97),
                   'HC': (452,235), 'HP': (503,123), 'IC1': (384,360),
                   'IC2': (621,258), 'JB': (501,213), 'PA': (350,280),
                   'PE': (515,165), 'PK1': (229,89), 'PK2': (464,81),
                   'PL': (375,213), 'PS': (350,173), 'PW': (72, 173),
                   'RF': (596,151), 'SB': (318,196), 'SH': (264,262),
                   'SU': (420,260), 'SV': (509,305), 'TA': (455,198),
                   'TC': (344,119), 'TK': (566,125), 'TO': (344,119),
                   'UH': (422,305), 'UP': (114,107), 'UV': (550,454),
                   'VA': (248,205), 'YC': (190,200)
                  }

        if self.alert == 1:
            dc = wx.ClientDC(app.frame.mymap.sb)
            dc.SetPen(wx.Pen("red", 1))
            dc.SetBrush(wx.Brush("red"))
            for row in app.frame.dataSource.rows:
                marker = row[1]
                dc.DrawCircle(coords[marker][0], coords[marker][1], 5)

            index = 0
            while index < new:
                item = app.frame.mylist.list.OnGetItemAttr(index)

                index += 1


class MyMapDisplay(wx.Panel):

    def __init__(self, parent):
        wx.Panel.__init__(self, parent)
        self.SetBackgroundColour("White")
        sizer = wx.BoxSizer(wx.VERTICAL)
        img = wx.Image("map_crop.png", wx.BITMAP_TYPE_PNG)

        w = img.GetWidth()
        h = img.GetHeight()
        img1 = img.Scale(w, h)

        self.sb = wx.StaticBitmap(self, -1, wx.BitmapFromImage(img1))

        sizer.Add(self.sb)

        self.SetSizerAndFit(sizer)
        self.Fit()
```

```
class MyListDisplay(wx.Panel):

    def __init__(self, parent, dataSource):
        wx.Panel.__init__(self, parent)
        self.list = VirtualListCtrl(self, dataSource)
        mainSizer = wx.BoxSizer(wx.VERTICAL)
        mainSizer.Add(self.list, 1, wx.EXPAND)
        self.SetSizer(mainSizer)


class GraphicPanels(wx.Panel):

    SPACING = 4
    COLUMNS = 1

    def __init__(self, parent, ID):
        wx.Panel.__init__(self, parent, ID)
        buttonSize = (100, 25)
        buttonGrid = self.createButtonGrid(parent, buttonSize)
        self.layout(buttonGrid)

    def createButtonGrid(self, parent, buttonSize):
        buttonGrid = wx.GridSizer(cols=self.COLUMNS, hgap=2, vgap=2)

        computerButton = wx.Button(self, -1, "Computer", buttonSize)
        self.Bind(wx.EVT_BUTTON, self.computerButtonClick,
                  computerButton)

        # Future work, add map creation functions
        #buildingButton = wx.Button(self, -1, "Building", buttonSize)
        #self.Bind(wx.EVT_BUTTON, self.buildingButtonClick,
                   buildingButton)

        buttonGrid.Add(computerButton, 0)
        #buttonGrid.Add(buildingButton, 0)

        return buttonGrid

    def computerButtonClick(self, event):
        dlg = MyComputerEditDialog()
        result = dlg.ShowModal()
        if result == wx.ID_OK:
            params = {'location': dlg.locChoice,
                      'building':  dlg.buildChoice,
                      'room': dlg.room_txt.GetValue(),
                      'mac': dlg.mac_txt.GetValue(),
                      'ptag': dlg.id_txt.GetValue()
                      }
            mcp.updateRecords(**params)

        dlg.Destroy()

    def buildingButtonClick(self, event):
```

```
        dlg = wx.MessageDialog(None, "Still working on this one.",
                               "In progress",
wx.ID_OK|wx.ICON_INFORMATION)
        dlg.ShowModal()
        dlg.Destroy()

    def layout(self, buttonGrid):
        box = wx.BoxSizer(wx.VERTICAL)
        box.Add(buttonGrid, 0,
              wx.ALL|wx.EXPAND|wx.ALIGN_RIGHT
             |wx.ALIGN_CENTER_VERTICAL,self.SPACING)
        self.SetSizer(box)
        box.Fit(self)

class DataSource:

    def __init__(self):
        self.GetResults()

    def GetColumnHeaders(self):
        return self.columns

    def GetCount(self):
        return mcp.count()

    def GetItem(self, index):
        return self.rows[index]

    def UpdateCache(self, start, end):
        pass

    def GetResults(self, **selects):
        self.columns, self.rows = mcp.getRecords(**selects)

class VirtualListCtrl(wx.ListCtrl):

    def __init__(self, parent, dataSource):
        wx.ListCtrl.__init__(self, parent,
           style=wx.LC_REPORT|wx.LC_SINGLE_SEL|wx.LC_VIRTUAL)
        self.dataSource = dataSource
        self.Bind(wx.EVT_LIST_CACHE_HINT, self.DoCacheItems)
        self.SetItemCount(dataSource.GetCount())
        columns = dataSource.GetColumnHeaders()
        for col, text in enumerate(columns):
            self.InsertColumn(col, text)
        self.SetColumnWidth(0, wx.LIST_AUTOSIZE)
        self.SetColumnWidth(1, wx.LIST_AUTOSIZE)
        self.SetColumnWidth(2, wx.LIST_AUTOSIZE)
        self.SetColumnWidth(3, 120)
        self.SetColumnWidth(4, 70)
        self.SetColumnWidth(5, 140)

    def DoCacheItems(self, evt):
        self.dataSource.UpdateCache(evt.GetCacheFrom(),
```

```
            evt.GetCacheTo())

        def OnGetItemText(self, item, col):
            data = self.dataSource.GetItem(item)
            return data[col]

        def OnGetItemAttr(self, item):
            if app.frame.searchPanel.alert == 1:
                return self.SetTextColour("red")
            else:
                return self.SetTextColour("black")

        def OnGetItemImage(self, item):  return -1


    class NotEmptyValidator(wx.PyValidator):

        def __init__(self):
            wx.PyValidator.__init__(self)

        def Clone(self):
            return NotEmptyValidator()

        def Validate(self, win):
            textCtrl = self.GetWindow()
            text = textCtrl.GetValue()

            if len(text) == 0:
                wx.MessageBox("This field must contain some text!",
                              "Error")
                textCtrl.SetBackgroundColour("pink")
                textCtrl.SetFocus()
                textCtrl.Refresh()
                return False

            else:
                textCtrl.SetBackgroundColour(
                         wx.SystemSettings_GetColour(wx.SYS_COLOUR_WINDOW))
                textCtrl.Refresh()
                return True

        def TransferToWindow(self):
            return True

        def TransferFromWindow(self):
            return True

    class IDLengthValidator(wx.PyValidator):

        def __init__(self):
            wx.PyValidator.__init__(self)

        def Clone(self):
            return IDLengthValidator()
```

```
    def Validate(self, win):
        textCtrl = self.GetWindow()
        text = textCtrl.GetValue()

        if len(text) != 5 or text.isdigit() != True:
            wx.MessageBox("Invalid property tag number.", "Error")
            textCtrl.SetBackgroundColour("pink")
            textCtrl.SetFocus()
            textCtrl.Refresh()
            return False

        else:
            textCtrl.SetBackgroundColour(
                     wx.SystemSettings_GetColour(wx.SYS_COLOUR_WINDOW))
            textCtrl.Refresh()
            return True

    def TransferToWindow(self):
        return True

    def TransferFromWindow(self):
        return True

class MacLengthValidator(wx.PyValidator):

    def __init__(self):
        wx.PyValidator.__init__(self)

    def Clone(self):
        return MacLengthValidator()

    def Validate(self, win):
        textCtrl = self.GetWindow()
        text = textCtrl.GetValue()

        if len(text) != 12 or text.isalnum() != True:
            wx.MessageBox("Invalid MAC.", "Error")
            textCtrl.SetBackgroundColour("pink")
            textCtrl.SetFocus()
            textCtrl.Refresh()
            return False

        else:
            textCtrl.SetBackgroundColour(
                     wx.SystemSettings_GetColour(wx.SYS_COLOUR_WINDOW))
            textCtrl.Refresh()
            return True

    def TransferToWindow(self):
        return True

    def TransferFromWindow(self):
        return True
```

```
class RoomNumberValidator(wx.PyValidator):

    def __init__(self):
        wx.PyValidator.__init__(self)

    def Clone(self):
        return RoomNumberValidator()

    def Validate(self, win):
        textCtrl = self.GetWindow()
        text = textCtrl.GetValue()

        if len(text) != 3 or text.isdigit() != True:
            wx.MessageBox("Room numbers have three digits,
                          use leading zeros if needed.", "Error")
            textCtrl.SetBackgroundColour("pink")
            textCtrl.SetFocus()
            textCtrl.Refresh()
            return False

        else:
            textCtrl.SetBackgroundColour(
                     wx.SystemSettings_GetColour(wx.SYS_COLOUR_WINDOW))
            textCtrl.Refresh()
            return True

    def TransferToWindow(self):
        return True

    def TransferFromWindow(self):
        return True


class MyComputerEditDialog(wx.Dialog):

    def __init__(self):
        wx.Dialog.__init__(self, None, -1, "Edit a Computer")

        id_label = wx.StaticText(self, -1, "ID")
        mac_label = wx.StaticText(self, -1, "MAC")
        location_label = wx.StaticText(self, -1, "Location")
        building_label = wx.StaticText(self, -1, "Building")
        room_label = wx.StaticText(self, -1, "Room")

        self.id_txt = wx.TextCtrl(self, validator=IDLengthValidator())
        self.mac_txt = wx.TextCtrl(self,
                                   validator=MacLengthValidator())

        #self.location_txt = wx.TextCtrl(self,
                                   validator=NotEmptyValidator())
        self.locChoice = ' '
        #location = wx.StaticText(self, -1, "Location:")
```

```
        self.location_txt = wx.Choice(self, -1,
                                      choices=app.frame.locationlist)
        self.location_txt.SetSelection(0)
        self.location_txt.Bind(wx.EVT_CHOICE, self.OnLocChoice)

        #self.building_txt = wx.TextCtrl(self,
                                        validator=NotEmptyValidator())
        self.buildChoice = ' '
        #building = wx.StaticText(self, -1, "Building:")
        self.building_txt = wx.Choice(self, -1,
                                      choices=app.frame.buildinglist)
        self.building_txt.SetSelection(0)
        self.building_txt.Bind(wx.EVT_CHOICE, self.OnBuildChoice)

        self.room_txt = wx.TextCtrl(self,
                                    validator=RoomNumberValidator())

        ok = wx.Button(self, wx.ID_OK)
        ok.SetDefault()
        cancel = wx.Button(self, wx.ID_CANCEL)

        sizer = wx.BoxSizer(wx.VERTICAL)

        flexSizer = wx.FlexGridSizer(3, 2, 5, 5)
        flexSizer.Add(id_label, 0, wx.ALIGN_LEFT)
        flexSizer.Add(self.id_txt, 0, wx.EXPAND)
        flexSizer.Add(mac_label, 0, wx.ALIGN_LEFT)
        flexSizer.Add(self.mac_txt, 0, wx.EXPAND)
        flexSizer.Add(location_label, 0, wx.ALIGN_LEFT)
        flexSizer.Add(self.location_txt, 0, wx.EXPAND)
        flexSizer.Add(building_label, 0, wx.ALIGN_LEFT)
        flexSizer.Add(self.building_txt, 0, wx.EXPAND)
        flexSizer.Add(room_label, 0, wx.ALIGN_LEFT)
        flexSizer.Add(self.room_txt, 0, wx.EXPAND)

        flexSizer.AddGrowableCol(1)

        sizer.Add(flexSizer, 0, wx.EXPAND|wx.ALL, 5)

        btns = wx.StdDialogButtonSizer()
        btns.AddButton(ok)
        btns.AddButton(cancel)
        btns.Realize()
        sizer.Add(btns, 0, wx.EXPAND|wx.ALL, 5)

        self.SetSizer(sizer)
        sizer.Fit(self)

    def OnLocChoice(self, event):
        self.locChoice = event.GetString()

    def OnBuildChoice(self, event):
        self.buildChoice = event.GetString()
```

```
class MyApp(wx.App):

    #def __init__(self, redirect=True, filename=None):
    #    print "App __init__"
    #    wx.App.__init__(self, redirect, filename)

    def OnInit(self):
        self.frame = MyFrame("PySafe", wx.DefaultPosition, (800, 800))
        self.frame.Show()
        self.SetTopWindow(self.frame)
        return True

if __name__ == '__main__':
    app = MyApp()
    app.MainLoop()



#!/usr/bin/python
# -*- coding: utf-8 -*-

"""mcp.py contains functions to connect to inventory.py for searches
   and updates."""

import urllib
import xml.dom.minidom


def getRecords(**kwargs):
    columns = ["Location", "Building", "Room", "MAC", "ID", "Time"]
    rows = []

    try:
        parameters = urllib.urlencode(kwargs)

        xmldoc = xml.dom.minidom.parse(
              urllib.urlopen("http://localhost:8080/getRecords",
                             parameters))
        resultsNode = xmldoc.firstChild

        for rowNode in resultsNode.childNodes:
            rowlist = []
            for tag in rowNode.childNodes:
                rowlist.append(tag.firstChild.data)
            rows.append(tuple(rowlist))

        return columns, rows

    except:
        empty = ('No connection to server', ' ', ' ', ' ', ' ', ' ')
        rows.append(empty)
        return columns, rows
```

```
def updateRecords(**params):
    try:
        parameters = urllib.urlencode(params)
        urllib.urlopen("http://localhost:8080/updateRecords",
parameters)

    except:
        print "No connection to server"

def count():
    try:
        xmldoc = xml.dom.minidom.parse(
                        urllib.urlopen("http://localhost:8080/count"))
        resultsNode = xmldoc.firstChild
        print xmldoc.toxml()
        rows = []
        for rowNode in resultsNode.childNodes:
            rows.append(rowNode.firstChild.data)
        return int(rows[0])
    except:
        return 1

def main():
    getRecords()

if __name__=="__main__":
    main()



#!/usr/bin/python
# -*- coding: utf-8 -*-

"""store_data.py handles all database interactions."""

import sqlobject
import sqlobject.sqlbuilder
import datetime

class CheckInRecord(sqlobject.SQLObject):

    location = sqlobject.StringCol(length = 15, default = ' ')
    building = sqlobject.StringCol(length = 3, default = ' ')
    room = sqlobject.StringCol(length = 3, default = ' ')
    mac = sqlobject.StringCol(length = 12, default = ' ')
    ptag = sqlobject.StringCol(length = 5, default = ' ')
    timeIn = sqlobject.DateTimeCol(default = datetime.datetime.now())
    trace = sqlobject.StringCol(default = ' ')

    def __cmp__(self, other):
        return cmp(self.ptag, other.ptag)

def connect_db():
    resource = 'mysql://user:password@localhost/tracker'
```

```
    connection = sqlobject.connectionForURI(resource)
    sqlobject.sqlhub.processConnection = connection

def update_by_client(mc, tracert):
    record = list(CheckInRecord.select(CheckInRecord.q.mac==mc))
    if record == []:
        row = CheckInRecord()
    else:
        row = CheckInRecord.get(record[0].id)
    row.set(mac = mc, timeIn = datetime.datetime.now(),
            trace = tracert)

def update_by_mcp(**params):
    record = list(CheckInRecord.select(
                  CheckInRecord.q.mac==params['mac']))
    if record == []:
        row = CheckInRecord()
    else:
        row = CheckInRecord.get(record[0].id)

    row.set(location = params['location'],
            building = params['building'],
            room = params['room'], mac = params['mac'],
            ptag = params['ptag'])

def get_records(**kwargs):
    days = '3'
    if kwargs == {}:
        records = list(CheckInRecord.select())
    else:
        sqlstatement = ""
        param = "check_in_record.%s='%s' AND "
        alert_param = "check_in_record.%s + interval %s day < now() "
        for eachArg in kwargs.keys():
            if eachArg == 'alert':
                statement = alert_param % ('time_in', days)
            else:
                statement = param % (eachArg, kwargs[eachArg])
            sqlstatement += statement
        sql = sqlstatement.rstrip(' AND ')

        records = list(CheckInRecord.select((sql)))

    return records

def count():
    return CheckInRecord.select().count()

def main():
    connect_db()
    CheckInRecord.dropTable(True)
    CheckInRecord.createTable(ifNotExists=True)
```

```
if __name__ == '__main__':
    main()


#cherrypy.config
[global]
log.error_file="/path/to/log/error.log"
server.socket_host = “your.ip.addr.ess”
server.socket_port = 8080
server.thread_pool = 10
tools.sessions.on = True


#pysafe.config
[/]
tools.staticdir.root = "/path/to/app/root/"
tools.sessions.on = True
```

APPENDIX B

TEST CODE

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

"""Test suite for computer.py"""

import computer

def main():
    computer.main()

if __name__=="__main__":
    main()

#!/usr/bin/python
# -*- coding: utf-8 -*-

"""Test suite for inventory.py:
       posts variables to the CherryPy server."""

import urllib

def test_inDB():
    f = urllib.urlopen("http://localhost:8080/inDB")
    print 'No parameters: ', f.read()
    parameters = {'mac': '112233445566', 'trace': 'foo bar baz'}
    params = urllib.urlencode(parameters)
    g = urllib.urlopen("http://localhost:8080/inDB", params)
    print g.read()

def test_getRecords():
    f = urllib.urlopen("http://localhost:8080/getRecords")
    print f.read()
    parameters = {'mac': '112233445566'}
    params = urllib.urlencode(parameters)
    g = urllib.urlopen("http://localhost:8080/getRecords", params)
    print g.read()

def test_updateRecords():
    parameters = {'mac': '112233445566', 'location': 'SB',
                  'building': 'UH', 'room': '012', 'ptag': '54321'}
    params = urllib.urlencode(parameters)
    f = urllib.urlopen("http://localhost:8080/updateRecords", params)
    print f.read()

def test_count():
    f = urllib.urlopen("http://localhost:8080/count")
    print f.read()

def main():
    test_inDB()
    test_getRecords()
    test_updateRecords()
    test_count()
    print "Done!"
```

```
if __name__=="__main__":
    main()

#!/usr/bin/python
# -*- coding: utf-8 -*-

"""Test suite for mcp.py"""

import mcp

def test_getRecords():
    cols, rows = mcp.getRecords()
    print cols, rows

    cols, rows = mcp.getRecords(location = 'SB')
    print cols, rows

def test_updateRecords():
    mcp.updateRecords(location = 'SB', building = 'PA', room = '231',
                      ptag = '43215', mac = '445566aabbcc')
    cols, rows = mcp.getRecords(building = 'PA')
    print rows

def test_count():
    print mcp.count()

def main():
    test_getRecords()
    test_updateRecords()
    test_count()

if __name__=="__main__":
    main()

#!/usr/bin/python
# -*- coding: utf-8 -*-

"""Test suite for store_data.py.  Inserts data, gets results,
   deletes data from database."""

import store_data
import random, os, time

seq = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
       'a', 'b', 'c', 'd', 'e', 'f']

buildings = [' ', 'AD', 'AF', 'AH', 'AS', 'AV', 'BI', 'BK', 'CC',
            'CE', 'CH', 'CO', 'CS', 'DD', 'ES', 'FB', 'FM', 'FO',
            'HA', 'HC', 'HP', 'IC1', 'IC2', 'JB', 'PA', 'PE', 'PK1',
            'PK2', 'PL', 'PS', 'PW', 'RF', 'SB', 'SH', 'SU', 'SV',
            'TA', 'TC', 'TK', 'TO', 'UH', 'UP', 'UV', 'VA', 'YC']
```

```
def generateMac():
    mac = ''
    for i in range(12):
        i = random.choice(seq)
        mac += i
    return mac

def generateRoom():
    return str(random.randrange(100, 500))

def generateLocation():
    locations = ['SB', 'PDC']
    location = random.choice(locations)
    return location

def generateBuilding():
    building = random.choice(buildings)
    return building

def generatePtag():
    return str(random.randrange(10000, 60000))

def generateTrace():
    host = 'csusb.edu'
    timeLimit = 15

    f = os.popen('/usr/bin/traceroute %s' % host)
    time.sleep(timeLimit)
    g = os.popen('/usr/bin/killall traceroute')
    trace = ''
    for line in f.readlines():
        trace += line
    f.close()
    g.close()
    return trace

def test_update_by_mcp():
    try:
        for i in range(100):
            params = {'location': generateLocation(),
                      'building': generateBuilding(),
                      'room': generateRoom(),
                      'mac': generateMac(),
                      'ptag': generatePtag()
                     }
            store_data.update_by_mcp(**params)
    except:
        print "Failure in update_by_mcp"

def test_update_by_client():
    try:
        trace = generateTrace()
        for i in range(100):
            mc = generateMac()
```

```
            store_data.update_by_client(mc, trace)
    except:
        print trace

def test_get_records():
    try:
        results = store_data.get_records()
        result = store_data.get_records(id = random.randint(1, 99))
        print "Select All: ", results
        print "Select One: ", result
    except:
        print "Failure in getResults"

def main():
    store_data.connect_db()
    test_update_by_mcp()
    test_update_by_client()
    test_get_records()
    print store_data.count()

if __name__=="__main__":
    main()
```

REFERENCES

Botting, Richard. Private correspondence. October 27, 2008.

Courage, C. and Baxter, K. (2005). *Understanding your users: A practical guide to user requirements*. San Francisco: Morgan Kaufmann.

Friedman, T. (2007). *The world is flat: A brief history of the twenty first century*. New York: Picador.

Gilb, T. and Weinberg, G. (1984). *Humanized input: Techniques for reliable keyed input* (Reprint ed.). Wellesley: QED Information Sciences, Inc.

Hofstede, G. and Hofstede, G. J. (2004). *Cultures and organizations: Software of the mind*. New York: McGraw-Hill.

Kralisch, A., Yeo, A. W., and Jali, N. (2006). Linguistic and cultural differences in information categorization and their impact on website use. *Proceedings of the 39th Annual Hawaii International Conference on System Sciences,* 93. Retrieved May 21, 2008, from http://doi.ieeecomputersociety.org/10.1109/HICSS.2006.254

Ledgard, H., Singer, A., and Whiteside, J.(1981). *Directions in human factors for interactive systems*. Berlin: Springer-Verlog.

Levin Institute. What is globalization?. Retrieved May 21, 2008, from http://www.globalization101.org/What_is_Globalization.html

Li, H., Sun, X., and Zhang, K. (2007). Culture-centered design: Cultural factors in interface usability and usability tests. *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing,* 1084-1088. Retrieved May 21, 2008, from http://doi.ieeecomputersociety.org/10.1109/SNPD.2007.489

Lowgren, J. and Stolterman, E. (2004). *Thoughtful interaction design*. Cambridge: The MIT Press.

Mehlmann, M. (1981). *When people use computers: An approach to developing an interface*. Englewood Cliffs: Prentice-Hall, Inc.

Nelson, T. (1987). *Computer lib/Dream machines* (Revised ed.). Redmond: Tempus Books of Microsoft Press.

Norman, D. (2002). *The design of everyday things*. New York: Basic Books.

Norman, D. (2004). Design as communication. Retrieved June 3, 2008, from http://www.jnd.org/dn.mss/design_as_comun.html

Norman, D. (2005). Human-centered design considered harmful. Retrieved June 3, 2008, from http://www.jnd.org/dn.mss/human-centered.html

Saffer, D. (2007). *Designing for interaction*. Berkeley: New Riders.

Schneiderman, B. (1987). *Designing the user interface: Strategies for effective human-computer interaction*. Reading: Addison-Wesley Publishing Company.

Schwartz, P. and Leyden, P. The long boom. *Wired*. Retrieved June 3, 2008, from http://www.wired.com/wired/archive/5.07/longboom.htm

de Souza, C. (2005). *The semiotic engineering of human computer interaction*. Cambridge: The MIT Press.

Spolsky, J. (2001). *User interface design for programmers*. Berkeley: Apress.

Tidwell, J. (2006). *Designing interfaces*. Sebastopol: O'Reilly Media, Inc.

Wikipedia. Internationalization and localization. Retrieved May 29, 2008, from http://en.wikipedia.org/wiki/Internationalization_and_localization