Improving the Distributed Evolution of Software through Heuristic Evaluation

By
Amy Jo Wooten
S.B., Computer Science at M.I.T., 2010

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY
January 2011
© 2011, Amy Jo Wooten. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author

Department of Electrical Engineering and Computer Science
January 14, 2011

Certified by

Emily Braunstein
MOAA Task Leader, Charles Stark Draper Laboratory
Thesis Supervisor

Certified by

Robert C. Miller
Associate Professor
Thesis Co-Supervisor

Accepted by

Dr. Christopher J. Terman
Professor of Electrical Engineering
Chairman, Masters of Engineering Thesis Committee
Improving the Distributed Evolution of Software through Heuristic Evaluation

By

Amy Jo Wooten

Submitted to the
Department of Electrical Engineering and Computer Science

January 14, 2011

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

In order to create the increasingly complex software systems needed to deal with today's technological challenges, we must be able to build on previous work. However, existing software solutions are quite often not an exact fit. Software developers have found multiple ways of approaching the problem of designing software that can be adapted as well as otherwise changed; Most of this effort has been aimed at the structural properties of the software, by creating open-architecture systems. However, there are still significant usability hurdles to overcome. A developer-oriented evaluation of open architecture interfaces could help meet some of these challenges.

In this thesis, I present a set of guidelines for designing a developer-oriented interface for software open architectures, developed through a survey of several related fields. I use these guidelines to design and implement an interface to the Maritime Open Architecture Autonomy, one such software framework. Finally, through two case studies, I demonstrate the usefulness of these guidelines as the basis of a low cost method of usability evaluation. Study observations and limitations are presented, as well as suggestions for further research into heuristic evaluation.

Thesis Supervisor: Emily Braunstein
Title: MOAA Task Leader, Charles Stark Draper Laboratory

Thesis Supervisor: Robert C. Miller
Title: Associate Professor
Acknowledgements

I have been blessed to have so many wonderful and supportive people in my life, instrumental to both this thesis and my personal development.

First and foremost, I would like to thank my advisors, Emily Braunstein and Rob Miller. You guided my first unsure steps with patience and understanding; you stood alongside me, offering continual support and encouragement as my ideas took shape; and when I was ready, you stood behind me, teaching me to take ownership of my work. You have been with me every step of the way. Thank you.

My deepest thanks and appreciation to the Charles Stark Draper Laboratory, for wholly supporting my professional and academic pursuits, and for the wonderful people I have had a chance to work with and learn from. Additional thanks the Naval Undersea Warfare Center, for helping make this thesis possible.

Thank you to the many individuals who guided and shaped me, both while at MIT as well as during my formative years: Mrs. Robbins and Mr. Pierce, for instilling a passion for science and math, as well as encouraging my creativity; J. Comer, for the consistency, accountability, and discipline to achieve my goals; A. Hunter, for conversations which helped me straightened out more than just my schedule; A. Silk, for always being there, and for helping me work through the things that matter most in my life; J. Knopp, for many good memories; and Professor Winston, for boldness.

And Mom: My biggest fan and number one critic. You always encouraged me to follow my passions and dreams. For 2 a.m. phone calls, for mending broken hearts and wings, for pushing me to expect more from myself and from life, for your unwavering support, and for being a source of joy and strength in my life: Thank you. All that I am, I owe to you. I love you, Mom.

Finally, all thanks and praise be to God, with whom all things are possible.

Publication of this thesis does not constitute approval by Draper or NUWC of the findings herein. It is published for the exchange and stimulation of ideas.
Table of Contents

CHAPTER 1: INTRODUCTION ....................................................................................................................5
  INTERFACE: A DEFINITION ..................................................................................................................7

CHAPTER 2: BACKGROUND ....................................................................................................................10
  THE CHANGING NATURE OF SOFTWARE: SOFTWARE EVOLUTION ............................................10
  DESCRIBING CHANGE .....................................................................................................................10
  INTRODUCTION TO USABILITY EVALUATION ..............................................................................12
  HEURISTIC EVALUATION ................................................................................................................12
  RELATED WORK IN HEURISTIC EVALUATION ..............................................................................12

CHAPTER 3: DEVELOPMENT OF THE HEURISTICS .............................................................................15
  APPROACH .........................................................................................................................................15
  GENERATING HEURISTICS ..............................................................................................................15
  THE HEURISTICS ...............................................................................................................................18
  DISCUSSION .......................................................................................................................................20
    Providing Information .....................................................................................................................20
    Reducing Mental Burden ...............................................................................................................23
    Freedom and Guidance .................................................................................................................27
    Consistency and managing variability ..........................................................................................29
    Errors ............................................................................................................................................30
    Minimalist Design .......................................................................................................................31
  MOVING FORWARD .........................................................................................................................32

CHAPTER 4: THE MOAA PROTOTYPE INTERFACE ...............................................................................33
  OVERVIEW .........................................................................................................................................33
  MOAA OBSERVATIONS ......................................................................................................................33
  DESCRIPTION OF PROTOTYPE INTERFACE ....................................................................................37

CHAPTER 5: TESTING .............................................................................................................................49
  RESEARCH QUESTIONS .....................................................................................................................49
  EVALUATORS ......................................................................................................................................49
  EXPERIMENTAL SETUP ....................................................................................................................50
  RECORDING OBSERVATIONS ............................................................................................................51

CHAPTER 6: OBSERVATIONS AND ANALYSIS .....................................................................................52
  DO THE HEURISTICS HELP INSpectORS FIND USABILITY PROBLEMS? ....................................52
    Heuristics that received more use .................................................................................................52
    Heuristics that were used less ......................................................................................................54
    Problems that were not covered by heuristics ............................................................................55
  WHAT CAN WE SAY ABOUT THE PROBLEMS FOUND? .................................................................56
  ADDITIONAL OBSERVATIONS ABOUT THE EVALUATIONS ...........................................................58
  CONSIDERATIONS ............................................................................................................................60

CHAPTER 7: CONCLUSIONS AND FUTURE WORK ...............................................................................61
  SUGGESTIONS .....................................................................................................................................61
    The heuristics ...................................................................................................................................61
Chapter 1: Introduction

Consider, for a moment, the building around you. At first pause, it appears a static structure. However a few moments of reflection bring details to the foreground—the soft glow of lights, quiet hum of the AC, steady plink-plink-plink of a leaky faucet. It is, in fact, a dynamic system of forces and interconnected parts, all working silently in the background.

It may also appear the building exists largely as it was constructed. However, as any building maintenance team will be quick to attest, it has most likely gone through several modifications, both major and minor. The addition of a new room, a heating system upgrade. Perhaps this is not the first time someone attempted to fix that faucet.

What if we were unable to fix things as they broke, or replace items before they malfunctioned? What if you did not understand what you could and could not change, or what the effects of your changes would be? How would you know whether adding a new lighting system required not only the electrical work, but additional structural support as well? How would you determine the unseen extent of your changes, or judge whether they were potentially dangerous?

These are challenges frequently faced by software developers.

Although largely invisible to the average user, every day we rely on software systems to work properly as well. While they do not undergo the traditional wear and tear that physical systems like buildings experience, they do experience corrective, perfective, preventative, and adaptive changes. Successful software, in particular, requires more adaptive changes as the customer base grows, spawning the need for new features and functionality. Domain analysis goes a long way, but it is impossible to anticipate, and thereby design for, all the changes the system will need.

Even at the outset of a project, existing software solutions are rarely an exact fit to customer requirements. Often, solutions contain a majority of the wanted functionality; however the crucial percent is not included. Developers are faced with the “buy it or build it” paradigm.

In order to continue providing solutions to meet our current technological needs, we need to be able to build on previous work- to understand it, to use it, and to change it when necessary. Despite this necessity, changing software remains a difficult activity.
In fact, it has been shown that if even 20% (some say 10%) of code needs to be modified, it is more cost effective to simply make it from scratch (Mili, 1995).

Software developers have found multiple ways of approaching the problem of designing software that can be easily changed and adapted. Most of this effort has been aimed at the structural properties of the software. Developers have moved to creating more open-architecture systems, leveraging existing techniques such as modular design and component based architectures.

These techniques have encouraged decentralized software evolution, the post-deployment development of software by independent parties. However, there are still significant usability hurdles to overcome. A developer-oriented interface to these open architectures could aid in meeting these challenges.
Throughout this thesis, we will typically talk about three main parties. These include:

**Platform developers:** the original software developers who create the platform or open architecture. This is typically a centralized group working together.

**Application developers:** the software developers who use the platform to create various applications. This can be in-house (for example, a large company divided into platform developers and those who make applications for the platform), or a third party. Additionally, it may be a group of people working together, or a single developer working on an application.

**End users:** the final users of the application that is produced. These are typically not software developers.

Additionally, we will refer to several different types of interfaces. The ones you should be familiar with are:

**Application interface:** the graphical front-end with which the end user interacts. While “the interface” of an application typically refers to the GUI, other less commonly seen interfaces include textual user interfaces (TUIs) and command line interfaces (CLIs). The
application developer may also provide some support (such as a user’s guide), which may be separate, or can be integrated into the interface.

Open-Architecture Developer Oriented Interface (OA DOI): the graphical front-end with which the application developer interacts.

Open Architectures are often delivered with a number of artifacts, such as the source code itself, documentation, various training materials, and tools for viewing, editing, and running the code. A good OA DOI will cohesively integrate the many tools and materials for that platform. There are several ways to do this. For example, a GUI for application developers may provide a way to navigate source code, tools for debugging projects, and links to documentation for the architecture.

![Diagram of Application Interface and OA DOI](image)

**Figure 2** An Application Interface (left) usually serves to separate the implementation of the software from its presentation, allowing users to interact with the software without knowing all the details of its construction. In contrast, an OA DOI (right) should provide an application developer lead-ins to the system, ultimately exposing the source code and tools in ways that support further change and development.

Currently, there does not exist any standard for what should belong in an OA DOI, nor is there any way for platform developers or UI experts to evaluate whether or not an interface has reached its goals.

This thesis approaches these issues by adapting heuristic evaluation for evaluating OA DOIs. The approach follows similar extensions of the heuristic evaluation method in evaluating other domains. It begins with a literary review in order to understand the nature of software evolution, and designing for change through the use of open architectures—why it is difficult, and the additional constraints imposed by a distributed development environment. From this, a set of new heuristics is developed specifically to support changing software—the activities of decentralized evolution.

The adapted heuristics guide the partial design and implementation of MOAA WorkBench, a prototype OA DOI for the Maritime Open Architecture Autonomy. This
interface incorporates and builds on many of the tools and training materials created by
the architecture developers at Charles Stark Draper Laboratory.

Finally, the effectiveness of the new heuristics in finding usability errors is demonstrated
through their use during heuristic evaluations of two open architecture interfaces: MOAA WorkBench and the Eclipse Plug-in Development Environment (or Eclipse PDE for short). Eclipse PDE is used to create, develop, test, debug, build and deploy Eclipse plug-ins and related products. It was built primarily to help developers use and extend the Eclipse Platform, although it can be used for other software development as well. Screenshots of Eclipse PDE can be found in the Appendix.

In this thesis, I approach the problem of designing for change from an OA DOI usability standpoint. Specifically, I contribute the following:

- A definition for an open architecture developer oriented interface-- one that can be evaluated with respect to changes that third party developers will make
- A set of guidelines for designing an OA DOI, based on usability principles derived from study of software reuse, open architectures, and software evolution
- An inexpensive yet effective method of evaluating OA DOIs, through adaptation of the Heuristic Evaluation method
- A prototype OA DOI for the Maritime Open Architecture Autonomy

The rest of this thesis is outlined as follows:

Chapters 2 and 3 introduce the heuristic evaluation technique and present the adaptations to the original heuristics.

Chapters 4 and 5 describe the interface designed and implemented using the modified heuristics. In addition, I describe the experimental setup for using the heuristics to evaluate both the MOAA prototype interface and the Eclipse Platform interface.

Chapters 6 and 7 present the results and analysis of the evaluations that were performed, recommendations for future work, as well as related work in the area of heuristic evaluation.
Chapter 2: Background

The Changing Nature of Software: Software Evolution

In the 1970's, software engineering emerged as a discipline for studying both software as well as the software development process. The increased research led to a focus on the methodologies, techniques, and practices involved. Initially, there was little attention paid to the development of software past deployment. Gradually there was a shift from viewing software as a product that was finished at deployment to having an entire lifecycle. This included the initial development as well as a maintenance phase.

Further research emphasized the importance of post deployment work when it was discovered that a significant proportion of software development cost occurs during the maintenance phase. In *Programs, Life Cycles, and the Laws of Software Evolution*, Meir Lehman noted that in 1977, only about 30% of software expenditures were spent on development, with the bulk (70%) being spent on maintenance. Studies today indicate that this problem is getting worse, with numbers ranging from 75%-90% of the total cost of software being spent on maintenance (Erlikh, 2000; Eastwood, 1993; Moad, 1990).

Lehman goes on to argue that evolution is an intrinsic property of software, asserting that programs which are used must continually undergo change, or else become progressively less useful (Lehman, 1980). In *The Mythical Man-Month: Essays on Software Engineering*, Brooks espouses a similar view. Any successful piece of software will necessarily need to be maintained (Brooks, 1975). As the customer base grows, requirements change; if the software is to survive, it will have to change to fit the customer's needs. Software will often need to be modified in ways that the original developers cannot even conceive, making it impossible to design up front for all necessary changes (Bennett, K; V T. Rajlich, 2000). As such, more and more developers have begun to realize the need for designing for the entire lifecycle, with software with evolution in mind.

**Describing Change**

Two parameters that can be used to describe the changes made to software are who makes the changes and when the changes are made. In confining ourselves to decentralized software evolution, we can further discuss the types of changes made past deployment, and the common techniques used to make those changes.

*Who makes the changes:* "Who" describes what party is making changes to the software. This is typically a central authority (ie, the original developers). However, more distributed development is being seen as well. Some examples of these include open source projects, as well software projects which have been subcontracted out.
When the changes are made: “When” describes at what point in the software’s life the changes are being made. Typically this is broken down into pre- and post-deployment evolution. Pre-deployment evolution includes all the changes in the software that occur before it is released, and are part of the “development” phase of the software life cycle. Changes which occur post-release are commonly referred to as “maintenance.”

Types of change: Maintenance refers to a broad variety of changes. Initially these were categorized into three groups. Corrective changes that usually comes to mind when we refer to maintenance—this category includes fixing discovered problems, like a bug patches, etc. Adaptive changes are usually introduced in responsive to a changing environment, including new user needs/requirements. Finally, perfective changes are aimed at improving performance or maintainability, and include things like optimization. (Lientz, 1978). Later, the International Standards Organization expanded maintenance to include preventative work, such as detecting and correcting aspects of the software that may be problematic in the future.

Commonly used techniques: In his 2000 dissertation Open Architecture: A Flexible Approach to Decentralized Software Evolution, Peyman Oreizy identify three main classes of software customization techniques varying in the amount of freedom given to the third party developers.

At the top we have behaviorally closed techniques. Generally, these allow developers to select and combine behaviors by enabling, disabling, or configuring pre-existing functionality. Examples of these include configuration files, user preferences and profiles. While they do not allow the software to add novel behavior, they are generally easy to understand and use, and allow the developer to tailor the software for well defined tasks and specific users.

The next class is that of behaviorally open techniques. Some examples include publishing Application Programming Interfaces (APIs), plug-ins, event architectures and component architectures. These typically allow the developers to extend and augment the software with new behaviors; however, they tend to not support removing or replacing built in behaviors. By allowing evolution along recommended paths, they represent a middle ground in terms of allowing developer freedom while insuring consistency.

Open code based techniques, which comprise the final class, allow unlimited changes to the software, so virtually all behavior of the software can be modified. Open source techniques generally offer the most flexibility as far as customization of the code. However, this technique suffers from a number of drawbacks. For one, it is more difficult to use. The burden of comprehension is placed entirely on the 3rd party developers who must have a thorough understanding of how the code works, as well as how to change it. Second, the changes may interfere with the existing behavior of the host application. For this reason, new changes must be thoroughly tested by the developers. Finally, if features are separately developed, they may interfere with each
other—so their integrated functionality must be tested as well. Oreizy refers to these problems as Change Analysis, Change Fragility, and Change Composition, respectively. (Oreizy, 2000)

Introduction to Usability Evaluation
Usability evaluation is the activity of assessing and measuring the usability of system. It is concerned with factors such as learnability, efficiency, memorability, user errors, and satisfaction. Usability evaluation can be done in a number of ways. It can be formative, whereby the purpose of the evaluation is to garner feedback toward improving the design, or it can be summative, used to determine the usability of a product that has already been deployed. It can be formal, providing quantitative feedback in the form of numbers, or it can be informal, with guidelines and rules of thumb. The evaluator/tester can be required to perform a highly structured set of tasks, or can be left free to explore the interface. Additionally, evaluation can be through inspection of the interface by UI experts, or through empirical testing with end users.

The available evaluation techniques vary with respect to these parameters, and thus have different things to offer. There is no “best” method for evaluating interfaces; in fact, a combination of two or more of these methods may lead to better results. Which method is chosen is dependent on a number of factors and constraints specific to the project.

Heuristic Evaluation
During heuristic evaluation, usability experts examine an interface to determine how well it adheres to established usability principles, called usability heuristics. Evaluations are performed independently, and at the end all the feedback is collected and turned into a problem report that is presented to the developer.

As an inspection method, heuristic evaluation is only performed by user interface experts, making it well suited for sensitive applications in which exposure to customer base is limited. Since the evaluations are performed independently by the UI experts, the experts do not even need to be on site, so the inspection can occur remotely. Additionally, it requires less coordination—user tests do not have to be scheduled, as experts may perform the evaluation at their leisure. Typically, only a small number (3-5) of UI experts is sufficient to determine a majority of usability infractions. Heuristic evaluation is also well suited toward evaluation early in the development cycle because it does not require an implemented interface. And finally, requiring fewer people, time, and set up makes the method low cost, relative to other existing methods. All these have contributed toward making heuristic evaluation an attractive option for discount usability testing. (Nielsen, 2009).

Related Work in Heuristic Evaluation
The growing interest in heuristic evaluation has lead to explorations of its effectiveness for evaluating the domains in which current methods of evaluation proved difficult or inadequate. This includes interfaces with expanded definitions of usability (Heather
Desurvire, 2004: Using heuristics to evaluate the playability of games), software without well-defined user tasks (Jennifer Mankoff, 2003: Heuristic evaluation of ambient displays; Zhou, 2004: Improving intrusion detection systems through heuristic evaluation), and collaboration based software (Kevin Baker, 2002: Empirical development of a heuristic evaluation methodology for shared workspace groupware.).

Evaluating “Playability”: Heuristics have become widely accepted method of usability evaluation for software, and show potential for other areas of evaluation, such as game usability. However, the goals of game play are different than those of software productivity. For software, the goal is often to produce applications that are easy to learn, use, and master. Games, on the other hand, are usually characterized as “easy to learn, difficult to master”. As such, an evaluation of the usability must go beyond the basic interface usability to determine whether the player will have a positive experience (one that increases their pleasure, immersion, and challenge) or a negative experience (where the player is bored, frustrated, or wants to quit the game). In this study, Desurvire et al develop a set of heuristics that assess additional properties of the game experience, including game play, story, and mechanics.

Ambient Displays: Ambient displays are aesthetically pleasing displays of information that sit on the periphery of a user’s attention and are used to monitor non-critical information; One such example would be a display for the outdoor temperature sitting at the bottom corner of a screen. Previously, very little work had been done in evaluation of ambient displays because existing evaluation techniques were focused on systems with clearly defined tasks and goals; with ambient displays, there are no such goals and tasks, other than providing information in a non-obtrusive manner. In a 2003 study by Mankoff et. Al, a set of new “ambient heuristics” was developed and used to evaluate two test displays. The study showed that the new ambient heuristics performed better than Nelson’s original heuristics in terms of major usability problems found by individuals; In addition, a greater percentage of problems overall was found. (Mankoff, 2003)

Intrusion Detection Systems: IDS’s are a security management tool used by computer network administrators for monitoring systems in order to detect inappropriate access. As with ambient displays, IDS’s do not have clearly defined user tasks; while users of other applications can have clear objectives about what to do and where to go, IDS users often cannot accurately predict when, why, and how intrusions occur, nor is there a set routine for how to respond. Following a similar approach to Baker and Mankoff, Zhou et. al developed a set of IDS heuristics after reviewing the state of practice in security management. These heuristics were then used to evaluate two IDSs, and found to identify significantly more usability problems than Nelson’s original heuristics. (Zhou, 2004)
Groupware: Previously, groupware faced serious usability problems, due in part to the lack of practical methods of evaluation for these systems. Existing methods which were designed to observe a single user's interaction with the system did not capture the usability problems associated with real-time collaborative work, while evaluation methods which observed group work were expensive, difficult to organize logistically, and did not generalize well (as the intergroup dynamics were highly variable and dependant on the group composition). In their 2001 study, Baker et. al adapted a theory of group behavior into a set of usability heuristics for groupware; in a subsequent study, they showed that these heuristics could be effectively used in heuristic evaluation to provide an inexpensive yet effective methodology for identifying team-work oriented usability problems. (Baker, 2001) (Baker, 2002)
Chapter 3: Development of the Heuristics

Approach

Work in heuristic evaluation shows several similarities in approach when adapting the heuristics toward the evaluation of a different domain. They typically begin with a survey of the current state of the field, obtained through either a review of current literature or consulting experts, both in usability and the relevant domain.

Following these examples, I used a similar approach. In surveying the current state of the field, my goals were multipart—

- To get a sense of the challenges and issues of related fields
- To examine how both the initial and third party developers responded to these challenges—both planned and unplanned, in theory and in practice
- To see if I could identify any overall ideas/themes/overarching trends

The fields I found most useful and relevant included software reuse, software architectures, software evolution, product family development, and distributed software development; however, general usability and software development principles were helpful as well. Deciding which areas to cover, and what to glean from each was challenging, as decentralized software evolution crosses so many fields. The question of which fields can offer the most insight should be reexamined for future work. This will be discussed later in the “Recommendations” section.

While developing the MOAA interface, I was able to make some general observations based on my conversations with MOAA developers. Discussions were primarily centered on developing MOAA, although they also shared experiences with other platforms as well. Additionally, throughout the course of this thesis I have had the opportunity to work with the Eclipse Framework while developing the MOAA interface. Reflecting on my experience as a first time “third party developer”, I documented the lessons that I learned along the way. While the development of the heuristics was a separate process, many of my own experiences as well as those relayed to me matched issues I had read about, validating their importance.

Generating Heuristics

I used two approaches for developing the set of heuristics. First, reflecting on the reading, interviews, and my own personal experience, I tried to pick out any consistent themes or issues that might serve as useful guidelines. Afterwards, I went through Nielsen’s original heuristics and thought about how they might be extended to this particular domain.
The final set of fifteen heuristics is not meant to be comprehensive in scope or depth; in fact, it is my impression that there are other additional heuristics that would be useful in evaluating software meant for further development. The purpose of this study is to explore whether it is possible to come up with a set that can be used to judge the usability of an OA DOI. Future work in this area should include refining this set of heuristics in order to come up with an optimal set. This will be discussed further in the “Future Recommendations” section.
<table>
<thead>
<tr>
<th>Original Heuristic</th>
<th>New or Modified Heuristic</th>
</tr>
</thead>
</table>
| Visibility of system status | • The interface should make explicit the underlying architecture—the elements, their properties and relationships  
• The interface should provide multiple views of the system  
• The interface should allow developers to know when changes have been made to the system, as well as provide the means to track and assess the structural and behavioral effects of those changes |
| Match between system and the real world | • The interface should use words, phrases, and concepts from the application domain  
• The interface should present information in a domain-task centered manner; Developers should know how manipulating the model maps to customization actions |
| User control and freedom | • The interface should make explicit the “open points” of the system: what the developer can add, remove, and modify  
• The interface should provide both guided adaptation and learning, as well as opportunities for exploration |
| Consistency and standards | • The interface should make clear the commonality between versions; It should make the distinction between core asset and application development, and allow both |
| **Error prevention** | • The interface should provide some indication of the “safety” of intended changes  
• The interface should indicate design rationale of the platform developers, and assumptions of the intended usage of components |
| **Recognition rather than recall** | • Developers should be provided with suggestions and incentives to customize/use methods afforded by system |
| **Flexibility and efficiency of use** | • The interface should support different skill levels, allowing both quick and easy customizations as well as more advanced changes  
• The interface should provide default values for customizations to allow for fast prototyping/easy start up |
| **Aesthetic and minimalist design** | • Developers should be able to view the system at the appropriate level of abstraction, or at different levels of abstraction  
• The interface itself should be customizable, allowing developers to control what they see |
| **Help users recognize, diagnose, and recover from errors** | • Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution. [Nielsen’s original heuristics, unmodified] |
| **Help and documentation** | • Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large. [Nielsen’s original heuristics, unmodified] |
Discussion
The rest of this chapter consists of an explanation of the new heuristics. The heuristics are grouped into five categories according to the broader area they address: heuristics concerning the presentation of information, reducing cognitive load and mental burden, striking a balance between user freedom and guidance, highlighting architectural similarities while managing variability, preventing user errors, and supporting minimalist design.

Providing Information
Several of the original heuristics, such as Visibility of System Status, are related to the availability and display of information. In order to be able to understand what is going on in the system, we should be able to identify what kinds system information the third party developer will need to keep track of. Because open architectures range from being very domain specific to being very general, this is often difficult to do. For example, software product families use open architectures to develop very similar software products within a specific domain, such as a line of mobile phone software. In this case, the general use cases and information requirements are probably well known. Extensive analysis is not always possible for more general frameworks (Eclipse Platform, for example—which can be used to develop a myriad of software applications). Software for developing applications in highly dynamic domains (with constantly changing information requirements), or domains which are not well understood, have the extra challenge of providing support for displaying information, even when the nature of this information is unclear.

New Heuristic: The interface should make explicit the underlying architecture—the elements, their properties and relationships

Irrespective of the application domain, the architectural information of the framework is usually helpful. Architectures play a variety of roles useful to software change tasks. Clear architectures support reuse at multiple levels and allow for reasoning at a higher level of abstraction. In addition, they may expose the dimensions along which a system is expected to evolve, the “expected evolutionary paths,” by making explicit the load bearing walls. Helping developers recognize and exploit existing patterns leads to less redundancy and complexity control. Finally, exposing top-level design decisions, the pieces, and their fundamental constraints allows the designer to reason about how to satisfy requirements. In Software Architecture: An Executive Overview (Clements, 1996), the authors describe the importance of architectural understanding:

"Deciding when changes are essential, determining which change paths have the least risk, assessing the consequences for proposed changes, and arbitrating sequences and priorities for requested
Changes all require broad insight into relationships, dependencies, performance, and behavioral aspects of system software components. Reasoning at an architecture level can provide the insight necessary to make decisions and plans related to change.

**New Heuristic: The interface should provide multiple views of the system**

The study of architectural representation includes languages, tools, and development environments specifically devoted to this task. Several papers (see Sidebar, below) on the topic echo the need for multiple structures or views. One analogy I found particularly helpful compared software architecture to that of an actual building. A contractor, architect, interior designer, landscaper all have different architectural views of a building because they interact with it in different ways. Similarly, a third party developer will be holding different "roles," and will need to interact with different sets of information. (Clements, 1996)

<table>
<thead>
<tr>
<th>Sidebar: Architectural Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is a plethora of suggested architectural views, with authors varying in what they deem is most important in a representation. Here is a sampling of some sets of views.</td>
</tr>
</tbody>
</table>

**Software Architecture: An Executive Overview** (Clements and Northrup, 1996)

- Whether or not structures represented are discernible at system runtime
- Whether the structures describe the product, the process of building the product, or the process of using the product to solve a problem.

**Software Design** (Budgen, 1993)

- **Structural Viewpoints**: static properties (structure) of the software
- **Functional Viewpoints**: tasks performed by the software, the data flow
- **Behavior Viewpoints**: cause and effect within the program
- **Data-modeling**: the data objects and the relationships between them

**The “4 + 1” View Model of Software Architecture** (Philippe Kruchten, 1995)

- **Conceptual (logical view)**: includes set of abstractions necessary to depict the functional requirements of a system at an abstract level.
- **Module (development view)**: frequently developed architectural structure, focuses on the organization of actual software modules, also hierarchy.
- **Process (coordination) view**: conceptual and module views have dealt with static aspect, process or coordination view focuses on runtime
behavior (how are entities created, communication mechanisms such as concurrency and synchronization).

- Physical view: mapping of software onto hardware.
- Scenarios: For illustrating use

Views are often not fully independent. Each view will most likely have to use a mix of terms and concepts, as well as levels of abstraction in their reasoning. Given the interconnectedness of this information, it is important to use not only the language of the related fields, but to show how different concepts, terms, actions, etc. map from one view to another. For example, training material may introduce a concept in the application domain. It may then describe how the concept maps to a component (or set of components) within the architecture. Finally, it may delve into the implementation details of a certain software class used to create that component.

*New Heuristic:* The interface should allow developers to know when changes have been made to the system, as well as provide the means to track and assess the structural and behavioral effects of those changes.

In addition to system-related information, the developer is interested in change-related information—keeping track of modifications that were made to the system, as well as the effects of those changes. There are several aspects of change that the developer may be interested in, such as when and where the changes are made, and if multiple developers are involved, who made the changes.

*Where the change takes place:* Software architecture changes can be partitioned into three categories according to locality (Clements, 1996):

1. **Local (code-level) changes:** These include development more or less contained in one module or unit of the code—for example, adding a class, modifying a single file, changing preferences, etc. In this case, there is one primary entity or module to keep track of.

2. **Non-local changes:** These include changes that involve multiple modules. For example, after extending the base class for new Perspective when creating an Eclipse plugin, you must go into the XML file for that plugin and list the new perspective under the Perspectives extension point. The XML file contains different extension points, and there is nothing that links the base class to the XML file. Even if the base class is created, without this extra step, the perspective will not be used; the plugin will not even be aware that the perspective exists.

3. **Architectural changes:** These include changes that affect the ways modules interact with each other, and generally require changes all over the system. Deciding to refactor software so that it uses a client/server design pattern (instead of implementing the behavior some other way) would be one example.
When the changes are made: The developer will want to keep track of their own changes temporally, as well. For example, for most interfaces, it is helpful to give some indication of progress on whatever task the user is accomplishing. While this is not always possible, for simple cases (such as filling out forms, templates, etc), this should be done.

In addition to keeping track of what is currently going on, it might be helpful to unobtrusively remind the developer what has been done. Depending on the application, domain, or task, the size of the change might vary from a quick fix that can be completed within minutes, to one that takes days, or weeks, of planning. For that reason, it might be helpful to save the state of the workbench, open recently used files, or offer some kind of indication of what the most recent changes to the system were.

Finally, offering suggestions for what should happen next (or a common next step after a task has been completed) might be helpful, even if the subsequent step is not necessary. For example, after implementing a base class for a commonly extended component, it might be helpful to indicate that it would be an appropriate time to set configurations for certain parameters referenced in that class—even if it is not necessary at that point, or if it is a more advanced use case.

Who makes the changes: It is desirable for the interface to provide support for tracking the changes the developer makes. However, if several developers are working to create an application, a developer will additionally want to be aware of changes that other developers have made to the system. Ideally, each developer’s additions or modifications would be self contained, however experience (as documented in several articles) has shown that in such cases, a large part of the cost is integration effort (Garlan, 1995; Mattsson, 1999). This includes both the integration of the developers’ additions into the host application, as well as integration between the contributions of the various developers—For example, just because individual plugins work well with the Firefox browser does not mean that they will work properly if you install several of them—they may fight for resources, or produce conflicting behaviors. Given that modifications often have invisible and/or non-local effects, it would be helpful to know when another developer has added/modified the system particularly in the case of non-local and architectural changes.

Reducing Mental Burden
One large hindrance to working with platforms and open architectures is they are often quite complex. As such, it is particularly important not only to deliver the information the developer needs, but to do it in the least taxing manner possible.
Several of the original heuristics address reducing the mental burden of the user through such steps as using plain and natural language, encouraging consistency with the real world, and using concepts, terminology, and phrases that the user is familiar with. These efforts are aimed at limiting the amount of new material they have to understand and relating new material to their existing knowledge by "speaking the user's language".

In our case, the user (developer) speaks several 'languages'. These include software development, application domain, and framework specific knowledge. While working with an open architecture, the application developer will find it helpful to think in one way or another at certain times. Speaking the user’s language means being able to switch between these seamlessly while explaining concepts and procedures.

New Heuristic: The interface should use words, phrases, and concepts from the application domain

Several papers on frameworks advocate the usefulness of presenting information in a domain-centered fashion—As one put it,

"The key problem is to find useful domain abstractions so that software components can be implemented without knowing the specific details of concrete objects." (Pree, 1995)

For traditional user interfaces, presenting information in a domain-centered manner allows the user to interact with interface while knowing as little as possible about what is "under the covers." For the developer, actually seeing and interacting with the source code is usually necessary, but providing a lead in through domain concepts may help conceptually tie the changes they wish to make to the structures they are modifying.

New Heuristic: The interface should present information in a domain-task centered manner; Developers should know how manipulating the model maps to customization actions

When application developers are presented with a new platform or architecture, one of their main concerns is whether the software can be used for their purpose, and how to accomplish this. Erich Gamma and Kent Beck describe the experience for application developers new to Eclipse Platform:

"Beginning with Eclipse feels a bit like parachuting blindfolded into Bangkok... When you land, you know you need food and shelter, but how are you going to get it? How can you map your clear desires onto the resources available?" (Gamma, 2003).
Using a domain-centered approach may not always present itself as the most obvious choice for describing a system—it may, for example, introduce redundancy in explanations, gloss over sections of the architecture that are not related to a domain concept, etc. However, it allows application developers to quickly assess whether or not the platform can support the desired functionality, as well as give them a sense of how easy or difficult their tasks might be.

*New Heuristic: Developers should be provided with suggestions and incentives to customize/use methods afforded by system*

Software open architectures and frameworks rely heavily on software reuse. It is the recycling of software development artifacts such as components, design patterns, interfaces, and the source code itself that provides potential for rapid prototyping. However, systematic software reuse remains a challenge, and is often not achieved. Organizations often provide incentives for developers to contribute to component libraries, but have less success promoting use of the components within repositories.

One obvious obstacle is if the developer does not make any attempts to locate a component. Frequently, this is labeled as “Not-Invented-Here” syndrome, and is attributed to reluctance on the developer’s part to use something they did not make. However, this is evidence that developers do, in fact, make a significant effort to locate and use components. Seven reuse failure modes have been proposed: No attempt to locate the component, component does not exist, component is not available, component is not found, component is not understood, component is not valid, or the component is not integratable (Frakes, 1996). While not-invented-here is large problem, a breakdown in any of these conditions will cause failure of reuse. Reproduced below is a graph presenting the distribution of reuse failures recorded at 29 software development organizations.
In their paper Supporting Reuse by Delivering Task Relevant and Personalized Information, Ye and Fisher argue that software developers are incapable or unwilling to reuse software if they are unaware of the existence of the reusable components, or they are unable to find, understand, or use them (Ye, 2002). Most research has focused on the techniques of browsing and searching tools to help developers locate components. However, these techniques are initiated by the developer, and are highly dependent on the developer’s knowledge of the repository. As such, they fail at promoting the reuse of components that the developer is unaware of. Providing a mechanism for suggesting customizations to make or components to use helps application developers use and change parts of the system they were previously unaware of.

**New Heuristic: The interface should provide default values for customizations to allow for fast prototyping/easy start up**

Many of the techniques used to create open architecture software encourage flexible, modular design. While this helps promote extensibility, it also has some drawbacks: often, the more options offered, the more decisions that will need to be made when developing the software. Frequently, this leads to heavy start up costs, as the new application developer is forced to consider each option and decide what is an acceptable choice. In addition, the developer may become tied up in details and grow frustrated. By providing default values, example choices, and appropriate ranges, some of the decision making can be delayed until the application developer has had some experience with the software and better understands the choices he or she is making.
Several of the heuristics are focused on user freedom. The user should always be aware of their options, and be should have easy access to those options.

*New Heuristic: The interface should make explicit the “open points” of the system: what the developer can add, remove, and modify*

In his thesis, *Open architecture software: a flexible approach to decentralized software evolution*, Peyman Oreizy suggests making explicit the “open points” of the system as a way of keeping the developer informed of what his or her options are (Oreizy, 2000). Using an architectural model based on components and connections, this means letting the developer know how to add, modify, or delete each component and connection/binding.

Nielsen’s original heuristic includes undos, redos, and emergency exits for leaving “unwanted states.” The amount to which this is possible ranges greatly in the techniques used to allow change, as well as what is considered a state.

As mentioned in Chapter 2, *Commonly Used Techniques*, platform developers use a number of techniques that vary along a scale of consistency and freedom. The more controlled and predictable the change is, the easier it is to provide assurances of consistency with the platform. The more freedom you allow the third party developer in the types of changes they are able to make, the more difficult it becomes to support those changes, and the final product, as part of a well defined state. For example, undoing a change to “user preferences” is fairly easy, and could be accomplished at the click of a “undo changes” button. Undoing the creation of a new module or class may require more steps—such as deleting the class/component, as well as any bindings or references associated with it. This could be a multi-step process, involving other components or pieces of the architecture. When the host developer allows modification at the architectural level, this becomes even more difficult, as the changes are potentially more widespread and impactful. Keeping track of all these behavioral/structural changes, and deciding what constitutes a state becomes more complex.

*New Heuristic: The interface should support different skill levels, allowing both quick and easy customizations as well as more advanced changes*

The host developers should decide what kinds of activities they will support and the amount of guidance they will give (in terms of tools, tutorials, documentation, etc), as well as what areas should be left for exploration. They will have to decide what kinds of changes require more “hand holding” and protection against errors. Although new users probably perform simpler, common tasks, they will require guidance simply because they are inexperienced.
with the system. Advanced users, on the other hand, are more likely to understand the system, but are also more likely to make complex changes that are more widespread in their effect. In general, commonly made customizations should be optimized so that after the initial learning, they may be performed quickly and without much fuss. On the other hand, users will most likely benefit from more detailed continued support for less frequently performed advanced changes.

New Heuristic: The interface should provide both guided adaptation and learning, as well as opportunities for exploration.

In addition to how much support to give to developers, the manner in which help is offered is also of concern. Nielsen points out:

“A little child will cry equally when held too tight or left to wander in a large and empty warehouse. Adults, too, feel most comfortable in an environment that is neither confining nor infinite, an environment explorable, but not hazardous.”

As previously mentioned, the information that third party developers need varies across levels of abstraction, as well as areas, including software development, architecture, and domain concepts. Because of this, it is difficult to present the information in a linear fashion. Developers will need some structured help; For example, they should have access to the specific information they need as they are performing a task (ie, they should not have to go digging for it, it should be presented to them). However, they should also be given the option to explore system and learn in more depth at will, and in a way that makes sense to them.

New Heuristic: The interface itself should be customizable, allowing developers to control what they see.

Finally, the application developer should be able customize the open architecture interface themselves. Platforms and open architectures usually have a large number of options, configurations, and commands. An interface that simultaneously displayed all of these at once would almost certainly be overwhelming. As such, it will likely be necessary to hide some of the options.

Some interfaces, called *adaptive interfaces*, automatically adjust the display. This is usually based on some model of the user, taking into account their skill level, and the likely relevant tasks. However, they have a number of downsides. For example, many users find it highly disorienting when a menu or screen changes, and they don’t understand why. Looking for an option that used to be available can also be aggravating. Additionally, these types of interfaces work best when there are clear user states and well known tasks—things which are
difficult to determine for application developers.

As such, a developer-oriented interface should allow the application developer to control which commands, views, and options they need according to their needs and use. As it is often impossible to anticipate the way software will evolve, providing the necessary tools to aid that development may not be possible. Building an interface that is flexible enough to allow application developers to integrate their own tools may be beneficial in this case.

Consistency and managing variability

While software platforms are often general enough to produce a wide range of software applications, they can be designed so that the bulk of the software can be reused to create many similar products, called software families, or software product line (Clements, 2003).

Software families tend to evolve in two ways. The first way is planned—through careful domain analysis, building much of the “core assets” up front. Future applications are then developed using this base. This approach is more commonly seen among mature, narrow domains, where requirements are better understood, and more of the planning can be done in advance.

Much more common is the case of incremental growth, which occurs in more dynamic/rapidly developing domains—for example, when a single product becomes successful, and a new “line” is started. Developers begin with the quickest way, by copying and pasting code, modifying it as necessary to fit the new situation. As different cases emerge, they add configurations, allowing them to enable and disable parts of the code more quickly. Eventually, modules are developed, allowing more flexibility in development. As the domain matures, a platform is created to allow more effective use of the components.

Families that develop this way tend to go through periods of growth and pruning. During the growth phase, new features, uses, etc. are added as new applications are explored. As patterns and similarities emerge, new abstractions are created. During the pruning phase, applications are mined for core assets, or reusable pieces, so that new functionality can migrate from applications back into the framework. This new, extended core is then used to develop more applications, and the cycle continues (Riva, 2003).

In this way, what is considered the “platform” and what is part of the “applications” is always in flux. Part of the platform developer’s responsibility is to decide when the switch should occur, although this is sometimes a difficult decision—How much variability warrants a new pattern?
New Heuristic: The interface should make clear the commonality between versions. It should make the distinction between core asset and application development, and allow both.

When presenting information, the platform developer should be concerned whether different words, situations, or actions should be presented similarly, as there is a tradeoff. If you present several separate use cases or applications, you experience rapidly expansion, code bloat, unwanted repetition, difficulties in maintainability and flexibility, etc. If, however, if you introduce a new pattern or abstraction, there is added complexity in understanding the new abstraction, when it is applicable, and understanding how it differs from other similar concepts or modules.

Regardless of where the separation occurs, the platform developer should make clear to the application developers when the abstractions and functionality they desire have already been built into the platform. This allows them to make use of existing functionality, rather than creating several different versions when they build applications.

Errors
Proponents of component based software and software frameworks used to envision “libraries” where one would simply find the piece that they wanted, and plug it in to their software with minimal effort. However, studies have shown that integration is difficult, and is a major cost in adopting software. People are spending more and more time as “system integrators,” gluing pieces together (Garlan, 2000).

Software created through composing independently developed components is difficult because it is hard to determine what assumptions each component makes about its operating context. In Architectural Mismatch: Why Reuse is so Hard (Garlan, 1995), errors arise as a result of assumptions about the following:

- Nature of components (infrastructure, control model, data model)
- Nature of connectors (patterns of interactions characterized by connector, the kind of data communicated)
- Global architecture (topology of system communications, absence or presence of particular components)
- Construction process (Some by instantiating a building block, providing a set of events and registration, etc)

In addition to understanding the host application as well as their own code, developers must determine whether the set of components will interoperate well (or at all), and whether the combined functionality matches what they need.
**New Heuristic:** The interface should indicate design rationale of the platform developers, and assumptions of the intended usage of components

Platform developers can help application developers by giving them a clear understanding of behavior of components, particularly by being as explicit as possible in addressing the common assumptions mentioned earlier. This includes how components operate and under what conditions, how to interact with them, and build on them. Example uses, as well as providing design rationale, are both often helpful in giving developers a sense of the nature and intended usage of components.

When assumptions are unaddressed, two problems that commonly arise are those of architectural mismatch and architectural drift. Architectural mismatch is when two or more components do not fit correctly together, producing unintended behavior (picture duct taping two pipes of slight different diameter together—you might still be able to transport water, but you will probably have problems somewhere down the line). Architectural drift is a similar, except on a larger scale. It is a more gradual eroding of the original architecture. The effects may be less apparent at first, but involve slowly weakening the “structure” of the architecture as a whole.

Checking for these and other errors before committing is difficult because many of the errors are only apparent at runtime. Additionally, errors often take a long time to show up and are may not be linked to one individual piece, but rather to the interaction of a number of pieces. Finally, these errors may be cumulative, although individually small and unproblematic, as in the case of architectural drift.

**New Heuristic:** The interface should provide some indication of the “safety” of intended changes

In addition to providing more reasoning, providing some indication of the cost (in terms of safety) of modifying a part of the system may be helpful in preventing errors. For example, how typical or untypical a certain change is, what different parts of the system are affected by tweaking a certain area, what kind of support is offered for tracking this change, etc. Giving the developer a sense of how widespread the effects of their decision may give them an indication of how much time they should spend on the choice. Decisions with fewer consequences, or that can be easily undone, may merit less analysis (for example, saving the developer the stress of wondering whether or not they can just accept the defaults presented to them during start up).

**Minimalist Design**

Any interface designer needs to find a way to strike a balance between providing visibility for existing functionality and not overwhelming the user. In order to
determine the minimal amount of information and functionality needed, they must first understand the user’s needs. For an OA DOI, this is a particularly difficult undertaking. The developer’s tasks are often not well defined or concrete, and may vary widely between uses, making it more difficult to pinpoint what should be immediately available.

**New Heuristic: Developers should be able to view the system at the appropriate level of abstraction, or at different levels of abstraction**

Providing multiple levels of abstraction can be a good way to approach this problem. A good abstraction does a number of things, such as suppressing details of implementation and simplifying understanding. Raising the level of abstraction increases the size of developer’s conceptual building blocks, allowing them to reason about the system. Lowering the level of abstraction allows them access to the details of construction, giving them a template for future development.

Choosing “the appropriate” level of depictions is difficult, as there seems to be a fundamental competition between the need for abstraction and concreteness. How do you describe something that can be used to build a huge range of products without a concrete example? And if you have a concrete example, how do you show which pieces are relevant only for that example vs. needed for any future application? If information is provided at too high a level of abstraction, the developer may fail to see the connection to his or her specific problem. If it is too concrete (coupled with there not being enough examples), they may not realize when something is application specific.

The host developer’s job should involve determining when (rather than if) certain information is relevant, and how best to present it. Information that is rarely needed might include information for infrequently performed tasks, experienced user cases, or introductory material that (once learned) probably does not need to be repeated. However, it might be helpful to include prompts leading to more information about these “rare” cases, so that the user knows that these options exist, even though they may not need the information at the present.

**Moving Forward**

The discussion above was a sample of some of the difficulties faced by the platform developers. The resulting heuristics were used to guide the development of MOAA’s DOI. The design process, as well as the final product, is described in the next chapter.
Chapter 4: The MOAA Prototype Interface

Overview

Two OA DOIs were used as case studies in order to test the effectiveness of the heuristics: The Eclipse Plugin Development Environment and MOAA WorkBench. MOAA WorkBench is a prototype developer oriented interface for the Maritime Open Architecture Autonomy. As part of this thesis, I designed and implemented the GUI portion of the DOI, incorporating some of the tools and materials provided by MOAA platform developers.

MOAA: A quick overview

Developed by Draper Laboratory, MOAA is a software framework for developing applications that control autonomous vehicles. MOAA incorporates all three classes of customization techniques, including a number of XML-based configuration options, a component architecture, and published APIs. Through these techniques, MOAA can be adapted and extended for various missions, vehicles, and domains.

MOAA is divided into the three following main areas:

The Domain Adapter: the “eyes and ears” of the system. The Domain Adapter handles the inputs and outputs for the vehicle through several interface Tasks. Adding new devices (such as sensors) to a current vehicle or adapting MOAA for a new vehicle typically involves the creation of new Tasks.

Situational Awareness: the “book of knowledge.” Situational Awareness stores information about various entities that the vehicle needs to know. These include information about the vehicle itself and its relation to the world around it, as well as other objects that exist.

Mission Planning: the “brain.” Mission Planning takes high-level goals for the vehicle and forms a plan for accomplishing them. The increments of functionality in the system are the Activities. Adding new functionality for a vehicle typically involves the creation of new Activities.

MOAA observations

To decide what to prototype, I interviewed the MOAA developers and users and collected feedback on their experiences with the platform. Despite the great time and effort put into making the system as extensible as possible, initial
application development revealed a learning curve associated with using the system. The following is a summary of my observations.

System Architecture Focus: A great amount of effort had been taken in designing the software to be as open, modular, and extensible as possible. As such, many of the training materials, pictures, and documentation were presented from an architectural standpoint, describing the software structure and its functionality. However, following a logical explanation of the architecture sometimes meant delving into architectural details that were less pertinent to performing common user tasks. Using an application development approach instead helps present system information while performing a number of functions: It helps give the application developer the bigger picture, keeps them closely tied to their goals, and shows them how those goals map onto the system.

As the MOAA development team began creating additional training materials and system pictures, they approached the system from an application development standpoint, focusing on the changes that third party developers would most commonly make. Continuing on this thread, both the text and graphic screens for the MOAA DOI were based on presenting new application actions to developers.

Incomplete Changes: Functionally, adding a component to MOAA often involves making several modifications. For example, adding a new Activity requires implementing a class, setting configurations for using that class, and adding it to the project. As typical with other MOAA extensions, several files are involved, and each file may need to be edited in multiple places: There are a variety of changes which need to occur, often using different techniques, at different levels, and in different locations. Because of this, it is difficult for users (particularly new users, but experienced users as well) to keep track of all the changes that need to take place. It was not uncommon to hear stories from application developers who added a new piece to the system, referenced it in other parts of the code, and could not figure out why it was not working-- only to discover they had forgotten to add the piece to a particular configuration file.

To address this, one goal for MOAA’s DOI GUI was to bring together all of the files that needed modification for each supported functional change, as well as add pointers to the appropriate location within each file to where these modifications would occur.

Options vs. Required Choices: MOAA supports a number of possible configurations and options. To an application developer unfamiliar with the system, trying to understand all the options at once could be overwhelming. However, it was sometimes difficult to know when decisions were required, and when options were truly optional.
One goal for the MOAA DOI GUI was making clear which decisions a developer was required to make, while still presenting other options they might be interested in. This helped to both reduce the mental burden on the developer, as well as facilitate quick start up.

Determining Knowledge Requirements: Additionally, sometimes developers had to make decisions involving concepts they did not yet fully understand. For example, several templates were provided for common extensions to MOAA, including two templates for creating new Activities. However, in order to choose between the templates, a new user would have to decide whether or not they needed to implement something called Replanning, which is a more advanced change. Even though new application developers would otherwise probably not need to know about this advanced option for a while, they had to make a decision about it up front. This resulted in them having to consult the training materials to learn what Replanning was, how it worked, and whether it was appropriate for them.

Determining the conceptual pieces needed to perform some of them common system tasks was difficult, however. Teaching a user how to modify the system often involved teaching them new concepts. These concepts were sometimes only describable in relation to other system concepts (which in turn were related to even more distant system concepts), making it difficult to clearly draw the line of what they absolutely needed to know.

Trying a different approach, the concept map starts with the “big picture” at its center: the three main components of MOAA (Domain adapter, Situational Awareness, Mission Planning). Branching outward from high-level to more specific directions, it introduces at most 3-4 new concepts at each step. Finally, at the edge of the map, implementation steps can be linked for instantiating various system concepts. The final result shows the path of concepts a new developer would need to be aware of and understand, in order to be able to create a functional change in the system. More concepts can be linked to concepts already in the map, providing additional context and information. While the concept map doesn’t solve the problem of exactly which concepts a developer needs to know, it does give them a sense of how closely related a concept is to the change he or she is trying to implement.
Progressive Examples

As noted in the developed heuristics, one of the main ways that developers learn a new program is simply by digging in and getting their hands dirty, using it, exploring it, experimenting, etc. When starting an application, it is often much easier to copy and paste something that already exists and modify it until it exhibits the desired behavior.

For open architectures, choosing the right examples is difficult for a number of reasons. Domain analysis, for example, is an important part of any user centered development process. However, in the case of frameworks, it is difficult to know even which domain will be used—MOAA, for example, can be developed to provide a framework for controlling a wide range of autonomous systems. As such, the platform developers have to resort to more abstract examples in order to illustrate concepts.

Examples tended to fall into two categories:

*Minimal functionality:* Similar to a “Hello World” example, these were good for demonstrating basics of the system. However, they tended to be very simplistic, and would require significant additional effort (both development and conceptual learning) to work into a complete product.
Complete product: These were fully functional application examples. Such examples were helpful because they were realistic—they demonstrate an entire system, with various options that the developer might be interested in. The problem is understanding which options are necessary, and which are applicable only in this specific application/instance.

As this seemed like an inherent problem with such systems, and developing examples was beyond my knowledge and expertise of MOAA, I did not try to address this problem with the GUI.

Inaccessible tools and resources

Although documentation was planned in advanced, some of the supplementary materials and support tools for MOAA were developed alongside it as the need arose. Because of this staged development, developers noted that several of the tools, libraries, and documentation felt like isolated pieces. Some of these resources received less use as a result—either because they required extra effort to individually find and launch, or because they were less salient and overlooked.

In the GUI, one of the goals was to push tools and other materials to the developer by providing these resources (or access to them) at times when they would likely be needed. An additional goal was bringing several isolated tools together in one central location, so that a developer did not have to seek them out individually.

Description of prototype interface

Following the adaptation of the heuristics and discussions with the MOAA development team, I generated a list of features that I thought should be included as part of the interface. Going through the list, I mapped each feature to the heuristics I thought it would satisfy. The final decision for which features to include was based on a) covering as many heuristics as possible, b) including features which could be inspected more easily, and c) time constraints. Additionally, I picked a set of features that would collectively support a number of related functional changes to the system.

The MOAA GUI was implemented using the Eclipse plugin framework, primarily because of the flexibility it would allow for integrating different tools and materials when building the display. The two basic mechanisms for creating a display with plugins are the use of Views and Editors. As suggested by the titles, Views are used to provide different presentations of a model, while Editors allow the user to manipulate and change it. Views and Editors can be arranged to provide multiple Perspectives of the system. For those familiar with the Eclipse developing environment, the Class Hierarchy and Package Explorer are two Views for viewing the elements of a project, while the Java Editor allows you to
open and edit the actual java files. Eclipse additionally provides template plugins for some well-known UI paradigms, such as Wizards.

Plugins can be used to create a standalone application, or they can be integrated into the Eclipse developing environment. The plugin for the MOAA GUI was installed in Eclipse. This allowed developers to leverage existing code development Views and Editors, such as the Eclipse C++ Editor, as part of the display.

The final display includes the following features:

*Three main perspectives—Learning, Development, and Run/Evaluate:* When generating a pool of possible features for the GUI, I noticed that they tended to fulfill one of three basic functions: to help users learn about system concepts or how to use the system, to aid in the development and modification of the system, or to allow developers to monitor the system and the effects of their changes. I used this as a basis for the layout GUI, dividing it into three main perspectives centered on these tasks. Because the main purpose of the GUI was to support changing the system, more time and attention was focused on the Development perspective.

*The Learning Perspective:* The purpose of the learning perspective was to provide resources which would help the developer learn system related concepts, either through directed search (such as looking up a concept to help complete a task) or through browsing and exploration. The section included various materials, including a MOAA platform developer rendering of the concept map, the user's guide and several system pictures. Each of these materials is contained in their own View. Views can be fully expanded; alternatively, they may be resized to only take a portion of the screen, stacked behind other views, or closed completely, allowing the developer control over what he/she sees. While the materials are contained inside the Learning Perspective, providing a central location for these resources, they may be opened independently. This allows one to open the User's Guide while they are developing, for example. In the future, this section could contain additional documentation, tutorials, and examples, although organizing these in a meaningful way will require more consideration.
2.2 Introduction to ADEPT

ADEPT is a mission planning and execution approach based on monitoring, executing, diagnosing, and planning. In the ADEPT approach, the system alternates between executing a plan and monitoring the progress of that plan. If there is any deviation detected by the monitor capability, the diagnose capability evaluates the problem and then, if necessary, the plan capability creates a new plan. The system then continues monitoring and executing until another diagnose-planning loop is needed.

Figure 2-2 ADEPT Loop

MOAA captures all ADEPT algorithms in an object called an activity. Activities provide user expansion of the MOAA system to provide new autonomy capabilities. Every activity has an associated objective that describes the goals of that activity. A set of activities is organized into a time based ordered tree structure that represents the mission to be accomplished. Activities on the left of the tree are the first to execute, with those towards the right of the tree representing future planned actions.

The root of the tree, the Mission Activity, has an objective that comes from the operator. Activities closest to the root of the tree are responsible for high level mission objectives, and hence have less detailed plans, than those closer to the leaves of the tree, which have

Figure 5: MOAA Concept Map (left) and User’s Guide (right) — two views within the Learning Perspective. These materials represent a JPEG and Word Document, each converted into a view. The ease with which this is done, along with the range of different materials that can be integrated into Eclipse Plug-ins, is one reason why the GUI was built using Eclipse.
The Development Perspective: The purpose of the development perspective is to help developers make a variety of changes to the system. These changes are modeled on exposing the “Open Points” of the system: Understanding how you can add/remove/modify a component of the system, as well as the bindings or connectors of that component. Currently, the GUI helps facilitate additions to the system—a decision made as part of the scoping process. The goal was to provide end to end support for extending two of MOAA’s main areas: the Domain Adapter (through the creation of new interfaces), and Mission Planning (through the addition of Activities). In the future, assistance for extending Situational Awareness may be added; extra support for removing and modifying components from all three areas could be added as well.

The Development Perspective has two main views—one graphic and one text based—providing two different approaches to developer options. In the text-based view, the options are ordered by the type of activity: learning about the system, creating a new piece, developing the component and integrating it into the rest of the framework, and running the system to evaluate the changes (the Learning option simply links back to the Learn Perspective, while the Run Option links to the Run/Evaluate Perspective). In the graphic view, the information is
presented based on location within a simple representation of the architecture. A box/arrow diagram represents the “components” and “connections” of the system. Within each component are the options for that part. For example, within the Mission Planning box, you can choose to either create a new Activity, or integrate it with the rest of the system.

Figure 7: The text-based view for the Development Perspective. Options are arranged by activity type on the left panel.
Wizards are provided to guide the developer through the process of creating a new Activity or Task. The wizards begin with a high level overview, describing how the components being added fit in to the bigger picture. Subsequent pages help tie the domain concepts to concrete classes that the user will have to implement. Class methods are shown as either mandatory or optional. Links beside each choice also lead to additional information on concepts involved, reasons to choose one way or the other, and the common or default choice, if there is one.
Figure 9: Wizard for creating a new Task. Clicking on the “Create a new Task” button leads to a wizard to guide the implementation of a new class, beginning with a description of a Task and how it relates to the rest of the system.
After the new Task or Activity has been created, there are other parts of the system that need to be configured in order for the new functionality to be integrated. There is a distinction between creating a piece and adding it to the system because after making a piece, the user may or may not want to use it in the open project; additionally, several projects might use it. Upon completion of a wizard, the new classes that were created open in the C++ Editor. Additionally, helper views pop open on the side of the screen to help developers with the integration. If a developer decides to perform this step later, they may close the view, and reopen it at their convenience.

The helper views are designed to help the user complete all the necessary changes to the system needed to fully integrate the new component they have just made. The changes are arranged by the locations of the changes, with each section providing a link to the file needed. Clicking on a link will cause it to appear in the editor area. The design helps address the previous problem of not being able to complete all the needed changes by providing one central location for all the changes that need to be made.

Figure 10: Wizard pages showing the available functions for an Activity. The developer may click next to a heading or decision point, in order to receive more information about the topic.
Each step is given a step name followed by a description of the step, introducing the concepts involved, and the modifications that need to be made. Following this is the link to the file that needs to be modified, as well as an automatically generated snippet of code to insert into the file. The snippets are based on either user input (prompted through text fields), or filenames from the open files in the editors. Finally, each section has a link to more information and examples.
as mentioned, in addition to creating a component and adding it to the system, further development is necessary to give it functionality. This is generally through the implementation of the method stubs found on the new Task or Activity class template (created by decisions within the wizard), and varies greatly depending on the developers needs. However, there is some functionality that most developers will find helpful, such as logging various types of information. Two additional helpers, available from either the graphic or text based views, assist developers in adding data logging or event logging to their components. These helpers are very similar in structure to the previous integration helpers: Each is arranged by files which must be modified, with sections containing step descriptions, links to files needed, as well as code snippets to copy and paste when appropriate.
The Run/Evaluate Perspective: The purpose of the final perspective is to provide a way for the developer to examine various parts of the system it is running. This closes out the end to end assistance of adding a component to the system—beginning with learning about the component, creating it, integrating and developing it, and finally observing its runtime behavior as part of the system. The current implementation contains only the Activity related portions of the Run Perspective; however, in the future, the display can be extended to include additional views.
Figure 14: Runtime Perspective. The Perspective is split into two Views: The Activity Tree Viewer (top) and a partially developed Log Viewer (bottom).
Chapter 5: Testing

Research questions

The purpose of this thesis is to test whether of the adapted set of heuristics for OA DOIs can serve as part of a low cost usability testing method. Toward that goal, we can identify two research questions:

Do the heuristics help inspectors find usability problems, using our modified definitions of usability and interface?

If we distill the lists of raw problems collected across all the inspectors into a single list of distinct total known problems, can we say anything interesting about the problems found? Some things we might want to look at include coverage—do some heuristics receive more coverage than others? Are there some heuristics that are never used? Are there problems that do not fall into heuristic categories?

Evaluators

In order to test their usefulness, the new heuristics were used as the basis of a heuristic evaluation for two open architectures: The Eclipse Platform and the Maritime Open Architecture Autonomy. Evaluators were recruited from the User Interface Group at MIT, and from the Human Systems Collaboration group at Draper.

While heuristic evaluation was originally intended for people with little usability experience, Nielsen later found that more experienced usability experts found more problems. Additionally, he found that “double” experts (who are both familiar with evaluation techniques, as well as have considerable domain related expertise) outperform evaluators who only have usability experience. For this study, the following areas of expertise may be relevant:

Usability Experience: Having knowledge of and experience with usability evaluation techniques.

Domain Experience: In this case, domain refers to the nature of the application being developed. Some frameworks are more domain-specific than others. For MOAA, the domain would be autonomous underwater vehicles.

Software Development Experience: The more software development experience a developer has, the more likely they are to have run into other open architectures, or the techniques used for facilitate change. Presumably this experience will affect how they interact with new open architectures.
Framework Specific Experience: Having previous experience with the specific framework presented.

For this study, we recruited eight evaluators, four for each of the two interfaces. All participants had some usability experience, although two of the evaluators had not previously performed heuristic evaluations. All the evaluators also had a computer science background and experience developing software. Three of the four evaluators for Eclipse had previous experience with the framework (developing plugins), while none of the MOAA evaluators had previously worked with MOAA. While Eclipse does not have a specific domain and MOAA does, none of the MOAA evaluators had any previous experience with underwater autonomous vehicles.

Experimental Setup

Evaluations were scheduled for two hours. Evaluators were given a brief explanation of the heuristic evaluation method (if they had never performed one), as well as a paper copy of the new heuristics and the set of severity ratings. They were asked to take a few minutes to read over the new heuristics and ask any questions regarding them before they began. They also received a brief, high-level description of the platform they were inspecting.

Normally, heuristic evaluation is performed in a walk up manner in which the evaluator inspects the interface. However, with frameworks, most of the time a user will be working within the context of a project. As such, it was decided that a better starting point would be a partially implemented project.

As mentioned, better outcomes are produced when the evaluator has both usability and domain expertise. However, heuristic evaluation is particularly well suited (compared to other evaluation techniques) for evaluators who have little domain related experience because the scribe, or observer, can guide the evaluator when using the interface, and answer questions. In addition, in cases where the evaluator is unfamiliar with the domain, it is appropriate to provide example use cases. These example tasks can help orient the evaluator, and give them a context for exploring the interface while performing the evaluation.

The tasks chosen for the evaluations were fairly simple, standard examples of adding to the platform. For Eclipse, evaluators were asked to “go through the motions” of creating a new perspective, a new view, and then adding the view to the perspective. These examples came from an Eclipse tutorial found in Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications. (McAffer, 2005). For MOAA, evaluators were asked to create a new activity and add data logging, as well as create a new task with event logging.

Evaluators were asked to keep these tasks in mind as they inspected the interface. They were instructed to evaluate all parts of the interface related to the tasks. They were encouraged to ask questions related to the interface, the
domain, or tasks while performing the evaluation. In addition, the Eclipse evaluators were provided with the tutorial from the *Eclipse Rich Client Platform*.

**Recording observations**

Nielsen suggests that heuristic evaluation can be performed two ways: where the evaluations are in a written report recorded by each evaluator, or where the evaluators verbalize their comments as they go through the interface. The first offers the benefit of being more formal, and possibly more thorough (the evaluator is forced to completely form and record their thoughts, assign it to a heuristic, and write a score). The second helps reduce the workload on the evaluator. For this study, evaluators were responsible for recording their own observations.

Evaluators were asked to rank each found problem according to severity, as well as select which heuristic the problem fell under and provide a short description. In addition, they were asked to take a screen capture of the area of the interface under question. They were provided with an electronic form for recording their answers. A scribe was present to answer questions, record responses, and take notes on the evaluation.

In practice, the form was used very little, if at all—the evaluators would begin using the forms, then the evaluations became more conversational, with the scribe doing most of the recording. The evaluations seem to flow better when the evaluator was concentrating on describing the error rather than stopping and recording it for themselves. This will be discussed in the following Analysis section.
Chapter 6: Observations and Analysis

At the beginning of the testing chapter, we began with two research questions, which we now revisit.

Do the heuristics help inspectors find usability problems?

All evaluations (save one, where the participant had no previous experience with heuristic evaluation) were successful in identifying usability problems. These ranged from simple errors with obvious solutions, to more complex problem for which the evaluator identified a problem but was unsure if a better solution was possible.

While they collectively found several problems, some of the heuristics received more use than others. Although frequency is not the only criteria for determining utility (for example, a heuristic may be used only sparingly, but finds critical problems), in general it is desirable for a heuristic to find several problems.

<table>
<thead>
<tr>
<th>More Frequently Used</th>
<th>Less Frequently Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make the architecture explicit</td>
<td>Domain-task centered presentation</td>
</tr>
<tr>
<td>Provide multiple views</td>
<td>Provide guidance and allow exploration</td>
</tr>
<tr>
<td>View and track changes</td>
<td>Indicate Safety and Rationale</td>
</tr>
<tr>
<td>Use domain related concepts</td>
<td>Distinction between common and application specific</td>
</tr>
<tr>
<td>Make “open-points” prominent</td>
<td>Support easy and advanced changes</td>
</tr>
<tr>
<td>Easy start up</td>
<td>Different levels of abstraction</td>
</tr>
<tr>
<td>Provide suggestions for use</td>
<td>Make interface customizable</td>
</tr>
<tr>
<td></td>
<td>Plain, constructive error messages</td>
</tr>
<tr>
<td></td>
<td>Easy to use and find documentation</td>
</tr>
</tbody>
</table>

Heuristics that received more use

The distribution of heuristics used was quite uneven, and although some evaluators were able to find more problems than others, they tended to rely on the same heuristics.
Several of the heuristics that performed well were those that helped resolve existing ambiguity. Evaluators would be going through the interface, and were confused about something—for example, information that was presented, or a decision that they had to make. Usually they would get confused, do an immediate search of the area to see if they could resolve their confusion, and if not, they would check the list of the heuristics to see which heuristic would have addressed the issue. Heuristics related to providing more information, or providing information in different ways were most often cited. These included provide multiple views of the system, the ability to view and track changes, and use domain related concepts.

In another common situation, evaluators would be going through the motions of their task and inspecting elements, and find they were stalled or unable to complete the task. Some reasons for this included things like not knowing what the next step was, or not finding the part of the interface that needed to perform the next step. Heuristic related to modification or change were most often used in these cases, such as make “open-points” prominent, provide suggestions for use, or make the architecture explicit.

As mentioned, there are other factors than frequency of use that determine a heuristic’s utility. In addition, there are several reasons that a particular heuristic may receive a lot of use. These include:

- This type of interface contains many problems addressed by this heuristic
- This particular interface contains many problems addressed by this heuristic
- This heuristic is easy to remember and use: As mentioned, this was the first time the evaluators had seen this set of heuristics. If a heuristic is well understood and easy to remember (either because it is near the top of the list, or it does not contain any ambiguous terms, or had catchy wording), evaluators may feel more comfortable using it, and will be more likely to recall it when looking for a heuristic to cover a problem.
- This heuristic covers more general problems: A heuristic which is a “catch all” for a bundle or class of problems is more likely to receive more use than a heuristic which applies to a very specific problem.

It is likely that the use of the heuristics was affected to some extent by all of these factors. Future studies adapting heuristic evaluation for OA DOIs will want to address extra variables to determine the effect of each. For example, by using the adapted heuristics on both application GUIs and OA DOIs, we can compare how the heuristics performed on each, confirming that the heuristics perform better for this type of interface. Performing the evaluations with the adapted heuristics on several OA DOIs would help ensure that the results and observations from this study could be generalized to evaluating other OA DOIs and were not specific/too heavily dependent on the two open architectures we
chose. Finally, work on the wording of the heuristics could also help confirm that the performance of the heuristics (whether good or bad) was due to the heuristic content rather than its presentation.

**Heuristics that were used less**

There were a number of heuristics that were used sparingly, if at all.

Some of the heuristics did not perform as well because there were no specific, salient violations. For example, while *provide guidance and allow exploration* is useful, it seemed as though finding a specific instance where this was violated during interface inspection was difficult for the evaluators. Usually an interface will provide some range of guidance, using a number of techniques spread throughout the interface. If that range is too narrow, or leans more heavily to one side (not enough support, or too restrictive), unless it is egregious, it may be difficult to pinpoint as a problem. For example, there were a few instances in which the evaluator wished to learn more, or wished to know whether the information they were viewing was essential to completing the task or just background knowledge, but saying “I want some unstructured learning right here” just did not occur. Similarly, while developers sometimes noted the inadequacy of their current view (and with Eclipse, could remember previous instances of interaction with the architecture that could have benefited from providing different levels of abstraction), they never stopped and said “A higher/lower level picture would have solved this problem” during the course of the evaluation.

Surprisingly, evaluators were quick to point out when something was missing or unclear, but tended to use the most general heuristic possible that would address it. For example, the heuristic for *provide multiple views* received widespread use. Heuristics that could be applied to a lacking a specific type of view (wanting domain-task centered presentation for a view, seeing different levels abstraction, or tailoring a view to support easy and advance changes) received less use. In the future, it might be interesting to explore a testing setup varying the generality of heuristics to see if it affects individual heuristic use. However, it seems unlikely that there is any real need to normalize for generality during evaluations, as many problems often fall under more than one heuristics for reasons other than specificity.

Wording and heuristic ambiguity were also an issue for a few of the heuristics. One example was *indicate safety and rationale*. Most of the evaluators were confused by what “safety” meant in this context and asked for clarification while preparing for the evaluation. It is possible that this ambiguity lead to uneasiness using the heuristic. Another heuristic which evaluators had difficulty with was distinction between common and application specific. The purpose of this
heuristic was making clear what belonged to the core (the platform), which is shared, and what should vary between versions of applications. The evaluators stumbled over what “versions” were—Some thought that versions were two similar but mostly different applications that the end user would interact with, and others thought that versions referred to the version of the platform they were using—for example, using Eclipse API that had been revised. While those were the main two, I suspect that there were 8 slightly different interpretations of this heuristic, and that no one felt entirely comfortable with it.

Some of the heuristics had less opportunity to be used simply due to limitations of the testing set up. Referring again to the distinction between common and application specific heuristic, it is unclear whether they really understood the difference between the core assets and application specific pieces from just one encounter with the system. This heuristic may have been more helpful if the evaluators were required to make several applications, as would be more typical of real application developers. However, in the context of this study, they were only going through the motions of making a single application. Another heuristic that was not often used was make interface customizable. Evaluators were quick to point out when more information would have been helpful, or a different view would have been nice, however they were suggesting persistent changes to the views. They did not differentiate between when a developer would want a certain elements of a view to be visible or hidden. In a similar vein, supporting easy and advanced changes did not come up very often. While it makes sense that developers of different skills levels might want to interact with the system differently, during the evaluation they were all novices with the architecture, and hence were unable to think in terms of “if I did this many times, it would get tedious,” or “I would not need these explanations after I’ve created this piece a couple of times.” It is still very possible that expert users that had spent a lot of time with the interface would be able to tell when there was too much information or guidance that they no longer needed.

Finally, some of the problems seemed more secondary to other concerns. For example, if the effect of picking a certain option was ambiguous, most evaluators were thinking more in terms of “I wonder what this means” instead of “I wonder how big an effect this choice potentially has.” As such, the heuristic for indicating the safety of an intended change received less use than previously anticipated. In a few instances, the evaluator did ask if it was possible to undo choices later, although they did not ask what that would entail. In the end, they still selected choices they felt uncertain of if they felt like it was the mostly likely to be right, regardless of the cost of undoing the selection.

Problems that were not covered by heuristics
There were several problems that were not found that did not fall under a heuristic. These were most often functional errors—such as pressing on a button and expecting parsed code that did not display correctly. Most of these
had already been noted previous to the evaluation, although a few new errors were found.

Additionally, there were a few usability errors found which did not fall under the modified heuristics but would have fallen under Nielsen's original heuristics—or problems which could be caught by either, but seemed more of a general usability problem than one which specifically applied to architectures. Because of this, it is suggested that it might be useful to perform both an open-architecture oriented evaluation to spot architecture related issues, as well as an unmodified heuristic evaluation for spotting general usability errors.

One point of interest in this study is to determine the usefulness of open architecture heuristics in finding usability problems. Closely related is the question of whether this particular set of heuristics could be modified to be more or less useful. If a large number of usability problems had been discovered which did not fall under the modified set of heuristics, it would have been interesting to see if a new heuristic could be gleaned from the set of problem. However, this was not the case—as mentioned, there were few problems that did not fall under a heuristic, and of those nearly all could be categorized as functionality problems or general usability problems that Nielsens' heuristics would have caught.

One possible reason for this which will be discussed in the following section is the evaluators' lack of experience with multiple architectures. Perhaps if they had been used to looking at architectures, both effective and ineffective ways of presenting them, they would be more likely to find architecture-presentation related problems, even without defined heuristics.

What can we say about the problems found?

Most of the problems that were found during the evaluations fell into three main categories: Errors of wording or information, errors involving processes, and errors involving views.

Errors with wording and information: In all the evaluations, errors with wording were quite prevalent. Perhaps because of the lack of exposure to either the architecture or the application domain, it was very easy to spot errors in terminology that may have been missed by the platform developers (or others with more exposure to the interface).

Closely related were errors with information. Evaluators were quick to find places were not enough information had been provided in order to complete the task, as well as places where there was insufficient support for finding more information. For example, for nearly every step there was a "More information" link near the heading of the step. However, evaluators would read the step and read through the description—at which point they would be confused, but would have forgotten about the information link close to the title. Not
identifying the link as a link or realizing it led to information they would need were also problems.

Other times, evaluators did not realize that they were receiving information vital to performing the tasks of the interface, and skimmed over the text. To counter this, one evaluator suggested labeling information as pertaining to steps, or background information about the system.

Errors of wording and information were associated with a number of heuristics. These included easy to use and find documentation (when they had to go digging for a meaning, and could not find it), and provide suggestions for use. Make the architecture explicit was also cited, as proper naming would have helped the user deduce the function of the piece and relationship to the rest of the application.

At times, it felt as though evaluators found these errors very quickly, they struggled for moment associating them with a heuristic. While all the errors eventually found heuristics, this might suggest that the evaluators were simply settling for one of the possible choices. In the future, a heuristic more targeted toward finding wording and terminology errors may be useful.

Errors of process: Evaluators frequently got stuck on how to execute particular steps related to their tasks. Question revolved around “What do I do now?” “Am I finished?” “Do I need to fill all of these out?” Common sticking points were not understanding whether or not they had a decision to make (ie, whether certain steps were optional and they could use the default), not knowing all of their options, and not knowing how to choose between options. Heuristics most often cited for these problems included make “open-points” prominent (which would have helped clarify when or not a decision was necessary), make the architecture explicit, and indicate safety and rationale.

These errors included words and terms that were used by had not been previously introduced or defined, not sufficiently explaining the difference between like terms (for example—all the evaluators stumbled between choosing to add an “Extension” vs. an “Extension Point” in Eclipse), and inconsistent terminology (using two or more terms to refer to the same thing). The evaluators were also helpful for finding words that were ambiguous or had multiple meaning in different domains.

A few times evaluators needed assistance because they skipped steps, performed steps in the incorrect order, or performed incorrect steps. For many cases, it seemed that more options left them unclear of how to proceed. Several evaluators indicated that it would be helpful to receive feedback about where they were in the overall process, as they were sometimes even unsure whether or not they had completed the task. In these cases, the heuristic for easy start
up (so they would have fewer decision up front), as well as view and track changes (so they knew where they were in the overall process) were often used.

Errors with views: A number of view related error and difficulties were found during the evaluations. First, evaluators sometimes had difficulty knowing which view they had to navigate to next. As mentioned earlier, because many of the steps are not linear, there is not a set path to proceed through the interface. To help with this, when designing the MOAA interface, the “next” views automatically popped up when the evaluator finished one step in order to help them with the next likely step. However, it turns out these were sometimes more disorienting than helpful. Evaluators were more confused by the change of screen—wondering what happened to their old screen, and what they were now looking at.

In addition, we realized that placing a helpful view in front of the evaluator was not sufficient. The views themselves do not explain themselves, and need some interpretation. Once or twice, evaluators actually navigated away from the needed view because they did not realize it was what they needed now to perform an action.

Finally, evaluators seemed pretty reluctant as a whole to explore other parts of the interface. While this was helpful to sticking to the task at hand, it also had important consequences: If there is no linear work flow/set path through the interface, and a certain view is not somehow introduced, it will most likely not be found and evaluated.

Evaluators attributed their difficulties understanding and navigating views to a number of things. Several wanted feedback on where they were in the process of making changes, using the view and track changes heuristic. The lack of introduction or prompts for views was sometimes considered a violation of provide suggestions for use. Also cited was make “open-points” explicit: they did not know what kinds of changes they could make with the view they were given, or, when told they were at the right view, how their desired changes mapped to the current view.

General usability errors: Finally, as mentioned earlier, several general usability errors and bugs were found that did not fall under the modified heuristics, but would have fallen under Nielsen’s general heuristics.

Additional observations about the evaluations

Mental Fatigue: Most (7/8) of the evaluations did not take the full two hours, although there was usually more of the interface to inspect. In fact, most took about half of that time, ending a little around the one-hour mark.

A possible reason is that the task produced considerable mental fatigue: the inspector had to keep in mind some portion of the platform architecture, some
application domain information (the task they were trying to perform), as well as the new heuristics. Some of this strain might be alleviated by providing more of the information beforehand, including the heuristics, so that the evaluator may become familiar with the materials before the evaluations.

Need for help with the interface: Nielsen suggests that one advantage to having an observer record the evaluator’s comments (as opposed to the evaluator recording their own comments) is that it allows for the observer to assist the evaluator if they have limited domain experience, and need to have parts of the interface explained.

This was very useful, as many evaluators had difficulty navigating the interface—Because there is not a well-defined workflow, it is hard to know where to go to next. Some of the screens that the evaluator interacts with are used more than once for the same task (you come back to it multiple times); Some of the steps can be performed in different orders (for example, editing multiple files—as long as they are all modified, it does not matter which order you see them).

Because of this, and because some of these steps may span several days if fully implemented (for example, creating and fully implementing multiple classes is not usually done in a two hour span), there is not usually a step by step ordered series of screens in which the user is forced to interact. Instead, users handle a lot of the navigation on their own, leading to confusion when they were not sure where to go next.

This was true even for evaluators who had previously used the interface (for Eclipse). As one evaluator put it, unless you had used the interface previously—and recently—you would probably be unable to perform the evaluation without someone else there.

Toleration of “fuzziness”: The process seemed to work better for evaluators who were able to tolerate a larger amount of “unknown”, or fuzziness up front. Some evaluators were able to look at the steps, try to go through the pieces of information and screens which someone would need to perform them. Other evaluators very much wanted to get the “whole picture”, and understand how their piece fit in to the overall workings of the architecture. While some connective knowledge is helpful and necessary, trying to understand the rest of the system during the span of the evaluation is probably not possible, and leads to some frustration.

One of the evaluators suggested it would be helpful to let both future evaluators and the users know that it was acceptable to not know everything at a certain point in the process—and just to keep on going. Otherwise, they would get stuck trying to understand some aspect of the system that was not entirely necessary, hindering their progress.
Considerations

**Limited exposure to other architecture interfaces:** Many of the evaluators had little previous experience/exposure with architectures—this was probably the only interface they had evaluated, or maybe they had experience with only one more. With traditional heuristic evaluation, the evaluator has had the opportunity to interact with/has been exposed to many similar interfaces. During the evaluations in this study, many of the evaluators expressed a lack of confidence in their ability to evaluate architectures simply because they had not worked with several architectures. They did not know which errors were problems that could be easily fixed, problems that could be fixed but required significant effort, and which were “hard problems,” which might not have a solution.

**Limited exposure to the heuristics:** When performing the evaluations, the evaluators would sometimes get “caught up” in the tasks and would forget about the heuristics. Some tried to counter this by first exploring the interface, then going through each and every heuristic, and think about whether the interface had violated that heuristic so far. However, this means that the problems found are the ones that are salient in the evaluator’s head/what they can remember. On the other hand, if they tried to keep the list of heuristics “in their head”, they might never use a heuristic if they forget about that heuristic.

One suggestion to counter this would be to give the list of heuristics to the evaluator ahead of time, so they can spend more time studying it/committing it to memory. Additionally, having a “keyword” for each of the heuristics so that it is easier to recall might be helpful. (Example keywords are provided in the table at the start of this chapter; for the full version presented during the study, refer back to Chapter 3, or to the appendix). It is possible that some of the heuristics were not used as often simply because they were harder to remember.

**Limited exposure to interface:** Finally, in a typical heuristic evaluation, the evaluator is able to explore most of the interface. With a platform of sufficient complexity, it is nearly guaranteed that the evaluator will only see a fraction of the interface. Because of this, there are some errors the evaluator simply will not catch, particularly those that involve subtle interactions of the system, or language inconsistencies in large APIs.
Chapter 7: Conclusions and Future Work

Conclusions

Changing software is a challenging but necessary task: Most developers interact with and build on work done by others. A number of techniques have been used to design software that is flexible, modular, and easier to change. In addition, platform developers provide applications developers with training materials, documentation, and other tools to assist them. Despite these efforts, many open-architectures are still difficult to learn and use. Cohesive GUIs integrating architecture-specific support may help address some of the remaining usability issues.

For this study, we set out to determine whether the method of heuristic evaluation could be extended to evaluate open architecture developer oriented interfaces (OA DOIs). We determined that it was, in fact, possible to identify open-architecture specific problems with heuristic evaluation. However, as the first run, we made some observations about the process that could help subsequent evaluations go more smoothly. For example, some consideration should be given to the evaluators chosen—the evaluators should have a background in software development, and preferably have experience with multiple architectures. Additionally, special steps should be taken alleviate the mental burden associated with using and learning the open architecture. Finally, the scope and limitations of the evaluation need to be kept in mind—the evaluation is simulating tasks that typically last longer than the significantly longer than the evaluation. Other suggestions on the heuristics and OA DOIs follow.

Suggestions

The heuristics

Remove or modify underperforming heuristics: As described in the previous chapter, several of the heuristics did not receive much use. Some of the time (Distinction between common and application specific, Indicate safety and rationale), this was due to ambiguous wording. Changes to phrasing, as well as providing a description of the heuristics before the evaluation, should help clear ambiguity.

Add a heuristic that more specifically deals with wording issues: As mentioned, a large number of wording errors were found, although there is not a specific
heuristic for dealing with these types of problems. Evaluators often had difficulty choosing a heuristic to attribute these problems to. As such, it seems like another heuristic addressing errors in terminology would be appropriate.

More methodical way of gathering heuristics: When coming up with heuristics, one of the difficulties was deciding which areas of study to look at. I picked fields which I believed were useful and relevant. Future research may want to explore other fields, as well as determine a more methodical way of deciding which fields should receive more or less emphasis.

Future research questions
As mentioned, the purpose of this thesis was to investigate whether the original heuristics could be adapted specifically for software frameworks, and to use them during heuristic evaluation. The measure of success would come from reflecting on the findings to see if they yielded any interesting results. Now that it has been established that this method can be useful for evaluating these types of interfaces, there are several more specific research questions we can address:

Number of problems: One measurement of success is the total number of problems. A good set of heuristics will be able to find many usability problems for a relatively small number of heuristics. As of now, there is no “baseline,” comparing the number of problems found evaluating these interfaces with the new heuristics vs. Nielsen’s heuristics. In addition, it might be interesting to look at the types of problems found under each set of heuristics: Are the same problems found, using different heuristics? Are very different problems found?

Severity of problems: Additionally, not all problems are of equal weight. In finding problems through heuristic evaluation, Nelson differentiates between two types of problems, calling them “major” or “minor.” Major usability problems are those that have serious potential for confusing users or causing them to use the system erroneously while minor problems may slow down the interaction or inconvenience users unnecessarily.

For this study, evaluators were provided a scale of 0-4 for judging the severity of problems found. However, most of the evaluators did not make much use of the forms, including the scale. When they did, they did not express much confidence in choosing a number for the evaluation (using expressions like “I guess”). A next step would be compare the incidence of major vs. minor problems.

Frequency of problems found: In evaluating usability, it would be desirable to find a method or set of heuristics that will find problems that frequently pop up; A method which found very uncommon problems would probably be of less utility than one which found problems that came up frequently. Future studies
may want to compare evaluations across interfaces to see how many problems are found by most of the evaluators.

Inspector performance: Finally, for this study we only looked at the problems found as one measure of the usefulness of these adapted heuristics; We could also explore questions regarding inspector performance. These include looking at the numbers of problems that individual inspectors find, whether inspector performance is consistent across systems, whether there are certain classes of problems (for example, problems found by most inspectors, or only by inspectors which generally find many problems), and the benefit to cost ratio (problems found compared to number of inspectors needed).

Thoughts on Open Architecture Developer Oriented Interfaces
Finally, while the purpose of this thesis was to explore one way of evaluating an OA DOI, a closely related topic is how to design an OA DOI. It is likely that there is a good deal of overlap because the goals of the design help form the basis for what is considered a successful interface during evaluation. Similarly, problems encountered during an evaluation may be helpful in generating design goals. With that thought, here are a few observations and suggestions based on tester interactions with OA DOIs:

Tell them what you’re telling them: Open architectures are often quite complex, even despite good software practices. This is part of their inherent nature, and any open architecture interface will need to present a lot of information. Application developers, on the other hand, are tasked with sorting through all this information and trying to understand if it is relevant to their goals. Throughout the evaluations in this study, evaluators often had difficulty digesting the needed information: Either they could not find the information, or they found it but did not understand what it meant, or did not realize they were looking at information they needed to perform their tasks. DOIs can help this process by providing meta information about what they are presenting. For example, instead of just presenting a picture of the architecture when they’re likely to need it, explain what it is and why they need it. Similarly, when giving information about steps that need to be performed, use labels like “Background information,” “Basic concepts,” “Steps.” Simply delivering the information is not good enough, as they may skim over or skip it. Cues on what the information is and how it relates to their goals will help direct their attention.

Any plan is better than no plan: Navigation was a recurring issue with the evaluations. As mentioned throughout this study, open architectures often do not have defined workflows, leading to collections (rather than sequences) of pages, steps, and options inside GUIs. This created a number of difficulties, such
as developers getting stuck and not knowing where to go next, or not knowing whether they have completed a step (since a page contains several unnumbered fields and no “Finish” button).

It is true that there is a myriad of ways a developer could progress with a project. Trying to plan out all the possible paths might be like trying to describe all the possible sequences of crayons used for a drawing when you’re not sure what the final picture will look like. However, you can probably go ahead and suggest a couple possibilities. The developer may end up using a path, or parts of one. At the very least, it will give them a sense of the types of paths other developers have taken, including the pages and resources that are available.

It’s okay not to know it all: Finally, I am struck by the lack of confidence users display about their decisions. When using applications, one would hope that the user has a generally pleasant experience—that they understand their choices and options and feel confident about their decisions. However, due to the complexity of open architectures, users will often have to make decisions without knowing all the details up front. By assuring users that it is alright to have a more shallow understanding of some concepts for now, and by giving them the opportunity and means to come back to these decisions and topics at a later time, you can reduce their anxiety, and increase their comfort with their decisions.
Bibliography


Human-Computer Interaction (EHCI'01). (May 11-13, Toronto, Canada).


Nielsen, J. “How to Conduct a Heuristic Evaluation.”


