

A Short Study in Two Solitudes

Gerald D. P. Dueck
Department of Mathematics and Computer Science
Brandon University
dueck@brandonu.ca

Abstract

During the creation of a software system, participants engage in a variety of activities. This paper describes the activities that take place in developing a computer-science solution to a software design problem and attempts to match solution elements to selected processes of software-engineering design. Not unexpectedly, a mismatch occurs, which leads us to wonder about not what is common, but what is missing.

1. Introduction

What aspects of Software Engineering should be included in a Computer Science curriculum? The answer to this question may help us to better understand the dichotomy between Software Engineering and Computer Science. To get to the answer, this paper considers the activities of Computer Science in the creation of a software system and reexamines them in the context of Software Engineering.

This presentational approach is entirely contrary to the corollary of the complaint made in [1], namely that “the study (of software architecture) is following practice, not leading it. Research still involves observing the design principles and actions used whilst developing real systems and abstracting the commonalities.” Notwithstanding, working backwards from a system design to determine what activities might have taken place is still a useful way for educators to provide students with perspective on the roles and activities of software engineering.

2. The problem: design a software system to ...

Automate the exchange of paperless documents between individuals at a small University. Documents should be made available only to those individuals selected by the provider of the document. Participation by central authority should be kept to a minimum. The success of the system depends on its widespread adoption by a disparate group of users, each of whom has varying degrees of comfort with respect to computer systems. The success also depends on aspects of the design that allow end users to develop novel ways to use the system, unanticipated by the designers.

The idea behind reducing participation by central authority is to remove barriers to use. An example of such a barrier is the experience an individual might have while negotiating with the local manifestation of IT Services to create a new location in the system for documents to be shared among members of a particular group selected by the

individual. It is much easier to send a broadcast email with an attachment and even easier to use sneaker net. That ease of use should be available in the document system. The user knows what is required; forcing the user to explain the requirement to another individual who holds “the keys to the castle” is a barrier made particularly unnecessary if a suitable automation can be found.

Some documents are restricted. Users have to be confident that the system can limit the readership and that they understand how to control the system to achieve restriction goals.

A straightforward design is to expose folders and documents over the World Wide Web. These concepts are assumed to be familiar to most users regardless of the desktop environment with which they are most accustomed.

Folders support *ad hoc* document adjacency. This is a familiar browsing paradigm and is enhanced by two additional features, one common and the other not quite so common. The first is a full-text search facility, well known to users of the Internet. The second is cross-linked folders and documents. The latter enhancement allows a document provider to categorize documents along various lines of interest, while searching allows a reader to arrange documents according to some current interest. Both techniques simply manifest different kinds of document adjacency.

3. Design Activities

Many aspects of this software system can be accomplished with the processes described in the Software Engineering Body of Knowledge (SWEBOK) [2]. However, the Software Design Knowledge Area specifically does not address I-design, defined as “invention design, usually performed during the software requirements process with the objective of conceptualizing and specifying software to satisfy discovered needs and requirements.” [2], pp 3-1.

How would a computer scientist approach the design of this system? One approach is to build a proving ground using as much off-the-shelf technology as possible and then invent the bits that fall between the cracks. These are the bits that have as-yet unknown solutions and performance measurements — the I-design. In this system, they turn out to be the user interface and, not unexpectedly, the data model. The latter incorporates the obvious data objects — folders and documents — but since restricting document access is as important as providing access in the first place, the data model must also accommodate fine-grained security.

3.1. Proving ground

The design activity continues by selecting technologies for the proving ground: browser, server, data storage. The particular brand of Web browser is not important. In fact, since the system is to serve users with a variety of computer-platform preferences, it is not advantageous to exploit the features of any one Web browser. .NET is chosen from among the many technologies available for the Web server and application development environment, with the constraint that the application will not use uncommon browser facilities. A relational database management system that supports stored procedures and that communicates conveniently with .NET is chosen for the data store.

Additional decisions are made at this point. The data store is separated from the Web server by interposing a Web-services layer; the performance degradation attributable to this decision is not known but expected to be no worse than linear. As much processing as possible is pushed down into the stored procedure layer. This increases performance

by reducing the number of round-trip calls and by using precompiled queries. It also results in a design discipline that leaves the Web-services layer as thin as can be achieved, casting it as a mere API.

3.2. Data and security models

The data and security models emerge by constructing sample folder and document entities in the database and manipulating the relationships. It becomes apparent the folder-document paradigm should be treated as a graph instead of a hierarchy. Doing so allows both folders and documents to be contained in more than one parent folder. Revisiting the problem statement, it turns out there are good reasons for exposing the multi-parent idea to the user; it supports user-defined document adjacency and aids in the presentation of search results.

The security model that results from this design activity is best described as record-level protection on both entities and relationships. The graph model is applied to the groups-and-users paradigm and appears to fit well. Some difficulties arise in applying the graph model to privileged operations, but this is resolved by controlling them through the tab-and-component design of the user interface. This turns out to fit the graph model nicely, but has consequences in the design of the user interface.

3.3. User interface

User-interface (UI) design initially follows along with the data-model design, yielding user-interface elements that display various views of the entities and the beginnings of a vocabulary to control them. Techniques are developed for dynamically loading web-page UI elements from table-driven specifications in the tab-component graph and dynamically binding both inter-element events and intra-element row-set data. A coherent user-interface paradigm emerges that supports both the users view and the programmers view.

The central idea of the UI paradigm is that a user is navigating through a world of containers and items. When visiting a container, UI display elements allow the user to see the contained items and containers (forward navigation), the containers that contain the container (backward and cross navigation), and the containers that contain a selected item (cross navigation) — four elements in all. The user can add navigational edges to the graph by creating a new container, linking to an existing container, creating a new item or linking to an existing item. The user can edit the permissions on any entity in the four display elements or can add permissions. This container-and-item paradigm is then applied to folders and documents, groups and users, and tabs and components.

3.4. Completeness

With design paradigms for both the data model and the user interface in hand, the next step is to determine the number and nature of all the elements required to flesh out the model. This is done both to get an idea of the implementation effort and to explore the range of the design algebra.

3.4.1. Enumerate the data-storage interface. The data model uses the composite pattern [3] to model the whole-part relationship [4] of folder-document, group-user and tab-component containment. These relationships are expressed in the form of three separate graphs with security links via permissions tables to the group-user graph. The

roles of selected database tables are summarized in Table 1. There is a 1:n relationship between each table and the corresponding permissions table and between the permissions table and the Auth table.

Table 1 Database Tables

<i>Table</i>	<i>Role</i>	<i>Permissions</i>
Folder	container entity	FolPerm
Document	item entity	DocPerm
FolDoc	N:m relationship	FolDocPerm
FolFol	recursive n:m relationship	FolFolPerm
Auth	container and item entity	AuthPerm
AuthAuth	recursive n:m relationship	AuthAuthPerm
Tab	container entity	TabPerm
TabComp	item entity (weak)	TabCompPerm

The security model supports permissions on both entities and relations. The security model is open in the sense that new operations and permissions can be defined for any class of entity or relation. The permissions tables relate operations on specified entities or relations to specific users in the Auth tables. Closure is achieved where the Auth tables apply the same permission mechanism to themselves.

Aside from entity-specific operations, the 8 permissions tables control 4 common operations: list, grant, change and delete. For each container-item graph, there are 10 list operations, as shown in Table 2. All return row sets, but the first two always return a row set with just one row.

Table 2 List Operations

Item by id
Container by id
Items in container by container id
Containers in container by container id
Containers containing an item
Containers containing a container
Permissions on an item
Permissions on a container
Permissions on a container-item relationship
Permissions on a container-container relationship

In addition to the list operations, for each container-item graph there are four update and delete operations on the entities and relations and another four of each on the entity- and relation-permissions tables for a total of 16 operations

Containers have two kinds of add operation: container-container and container-item. There is a difference between creating a new container or item and simply creating a new relationship between two existing entities. This adds another 4 operations.

In summary, each graph has 10 list operations, 16 update and delete operations, and 4 create operations for a total of 30 operations. There are three graphs: document-container, group-user and tab-component, for a total of 90 basic operations in the model.

These operations define most of the interface for the data-storage engine (DSI). Each DSI method is parameterized by at least the auth_id of the authorized entity performing the operation and the id of the entity on which the operation is performed. The opera-

tion itself is implicit in the method. The implementation is required to perform authorization checking before proceeding with the operation. To determine if an operation can proceed, there must be some entity in the corresponding permissions table *p* for which *p.auth_id* matches *auth_id* or the *auth_id* of any group directly containing *auth_id*, *p.ao_id* matches the entity *id* and *p.operation* matches *true*.

3.4.2. Enumerate the UI. Each graph requires four display elements and each display element requires several modal pages: four edit-permission pages, four add-permission pages, four link-creation pages and two entity-creation pages, for a total of 18 elements per graph and 54 elements for all the graphs.

3.5. Look for abstractions

The design exploits symmetries. Symmetries occur in the container-item graph model, in the protection model, in the data-store queries that support the models and in the UI elements. The design process continues by attempting to discover the abstractions that describe the symmetries.

Without using abstraction, a straightforward implementation would simply hand-code each of the 90 DSI operations and the 54 UI elements. The drawback to this approach is that maintaining consistency in the presence of changes to the model requires manual changes to a large number of places in the implementation.

Genericity and inheritance [5] are two ways to address the drawback. Using genericity and starting from a model description, code for the DSI operations and UI elements can be generated in a completely consistent way. Using inheritance, commonalities are factored into behaviour classes while specifics can be factored into instance classes in a complex variation on the “extract superclass” refactoring [6] (pp. 336).

Depending on the programming languages available, both approaches are used. Inheritance is used in the Web server where a language like Java or C# is employed; genericity is used in the DSI and in the SQL queries, since Web services and SQL stored procedures use languages that do not support inheritance. It should be noted that genericity here does not mean C++ templates but rather source-code generation. However, the semantics are isomorphic.

One technique to discover the abstractions in the user interface begins by writing a portion of the system using text-editor cut-and-paste methods to replicate code elements and make global changes. The portion has to be significantly large that the true symmetry can emerge. Programming discipline is applied to retain the “shape” of the code so it remains as similar as possible, using what Martin calls “untyped polymorphism” [4]. At this point, the similarities are refactored into subclasses of an abstract “instance” class and the differences are refactored into subclasses of an abstract “behaviour” class.

Without refactoring, each of the three graph structures requires hand coding for each of 4 display elements per graph for a total of 12 display elements. After refactoring, the three graphs all make use of one container-list class and require hand coding of 12 classes derived from a list-instance class. Similar gains are made after refactoring the edit- and add-permissions elements.

There is a significant difference in terms of productivity between the straightforward implementation and the refactored one. Although the 12 display elements are remarkably similar, the differences that do exist are scattered throughout the code. Even after implementing one set of 4 as a working prototype and using a text editor to clone the set and morph the scattered differences into a second set of elements, the development and debugging time amounts to as much as 1 day per component. Some of the detailed

work, such as making data and buttons visible or manifesting other visual attributes dependent on row-specific data, is incredibly meticulous and error prone. Making changes that affect the general look and feel of the entire interface requires changes in each of the 12 components that are sometimes non-trivial.

It is important that the scale of the user interface has not changed. From the user's point of view, the number of screens and the navigation between them is the same. The strength of this approach lies in its application of abstraction; the benefit is a remarkable reduction in implementation time and more control over the final result.

It is equally important that the look and feel of the user interface has been separated from its implementation.

4. Comparison with software engineering

In following through with the plan for this paper, the next two sections describe interesting design tensions that arise when attempting to pigeonhole design aspects into categories that differ across disciplines.

4.1. Software architecture

Baragry and Reed offer a syllogism to which, they suggest, the research community ascribes implicitly: "traditional engineering disciplines design and build systems which exhibit a level of design abstraction known as the system architecture. Software developers build systems and can identify high level design abstractions. Therefore software systems have a system architecture" [1]. This raises the question: What is the architecture of the system described in this paper?

Part of the answer lies in examining decisions taken to achieve the design. According to [7], architectural decisions identify the system's key structural elements, the externally visible properties of these elements and their relationships. A simple test is offered by [8] to prevent being overwhelmed by considering too many decisions: a decision that can be deferred to a lower level is not an architectural one.

The major points of the design use 1) a Web browser and Web server to implement a distributed client-server application; 2) a folder-document model to support end-user navigation through adjacency; 3) an abstraction achieved by leveraging the folder-document model into a container-item model and reused in the protection scheme and elsewhere in the design; 4) a relational database to model the folder-document-permission arrangement and a conventional file system for document storage; 5) a Web service to define the data-storage interface.

Decisions regarding some of these design points can be deferred and are therefore not part of the architecture. The relational database is firmly hidden behind the DSI and could be replaced by other technology in a transparent way. However, arguing that a component as significant as a relational database, which implicitly influences the design through its well-known strengths and limitations, is not part of the architecture may be difficult to do convincingly. Similarly, recognizing that the Web service is just another tool for binding relegates the decision to use it to the level of non-architectural detail, even though it is a significant technology.

Perhaps there is another approach. Structural elements are subdivided into three categories of requirements in the architectural relationships of [9]: domain, non-functional, and installation. Domain requirements include business-process functionality; data and their relationships; and timing between functions. Non-functional requirements include application look and feel and runtime performance. Installation require-

ments include available site hardware platforms, middleware and communications software. Using these categorizations, the document-folder and group-user models are essential to the domain architecture; the UI, which includes a component-tab data model, and the performance implications of using a relational database belong in the non-functional category; the Web server, the application-development environment and the Web service fall into the installation-requirements category.

In this decomposition, observe there is an abstraction being reused in different categories. Paraphrasing Martin [4], an abstraction is an amplification of the essential and an attenuation of the detail. The container-item abstraction fits this description. But while the component-tab instance of the container-item abstraction is incorporated seamlessly in the present design alongside the folder-document and group-user instances, it is separated arbitrarily by the architecture categorization. This raises the question of which approach is better; it is not clear how to resolve this tension.

4.2. User interface design

Despite the presence of a fundamentally sound and complete data model, the user interface can make or break a system. Users acquire a mental model of objects in a system; in a successful user interface, the mental model corresponds with the program model [10]. In Microsoft's Inductive User Interface guidelines (IUI), features of applications are broken into screens or pages that are easy to read and understand [11]. Nielsen and Molich's heuristics are 15 years old and still applicable [12], [13]. Usability design exercises with cognitive walkthroughs [14] and the application of older techniques like action analysis are all important in creating a suitable user interface. The literature is rich with suggestions for designing and evaluating user interfaces, but one thing is clear: developing a successful user interface is a cut-and-try process that involves the participation of end-users.

The user-interface reduction described in section 3.5 together with a robust implementation of the data model provide an environment in which user-interface ideas can be prototyped rapidly, decreasing the cycle time for round-trips to the usability lab.

Recall that the UI requires 18 elements for each of three container-item graphs and that many of the elements bear similarities both within and across graphs. The similarities were exploited to increase consistency for the users' experience and reduce development time through recognizing abstractions and creating reusable components. It is not clear how to do this at initial design time; there is only the vague idea that reusability will emerge if only the correct abstractions can be found, and that symmetry is the key.

The reduction was not achieved using a software-engineering process. The approach used to achieve it is not really amenable to the processes of software engineering because it is exploratory and therefore contains unknown risk factors. But it is only by achieving this result that the true architecture of this part of the system begins to emerge.

5. Conclusions

The software system described in this paper incorporates elements from both Computer Science and Software Engineering. The elements include a file-system-like protection scheme; a relational data model and associated queries; refactoring and design patterns; performance analysis, implementation-cost analysis, and project management techniques; user-interface design and evaluation. The technologies include application-

development platform, Web-page generation, Web services, and database and file-system services. Thus building the system requires education in traditional computer-science topics as well as software-engineering processes, in addition to training that leads to mastering development and operational technologies.

The real breakthrough in designing the system comes with the ability to “program the abstractions,” an idea taken from Martin Fowler’s vision: “one of the strongest qualities of language workbenches is that they alter the relationship between editing and compiling the program. Essentially they shift from editing text files to editing the abstract representation of the program.” [15] This may appear to simply imply the need for inventing yet another tool. In fact, for the system at hand, that is exactly what needed to be done. But programming the abstractions is more than just another technology. And this paper suggests that finding and manipulating the abstractions is not a measurable process and therefore not engineering. It may be science, in the sense there is hypothesis (an abstraction must exist) and experimental method (produce the symmetries and examine them to find the abstraction).

Two solitudes or not, commonalities exist between computer science and software engineering. But in the end, there are aspects of design that transcend the application of process and approach the intangibles of art.

References

- [1] J. Baragry and K. Reed, “Why is it so hard to define software architecture?” Proceedings of Software Engineering Conference, IEEE, 1998, pp. 28 – 36.
- [2] Guide to the SWEBOK, IEEE, 2004.
- [3] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- [4] Robert C. Martin, Designing Object Oriented C++ Applications Using The Booch Method, Prentice-Hall, 1995.
- [5] Bertrand Meyer, “Genericity versus inheritance”, Conference on Object Oriented Programming Systems Languages and Applications, ACM SIGPLAN Notices, Volume 21, Number 11, November 1986, pp. 391 - 405.
- [6] Kent Beck, John Brant, William Opdyke and Don Roberts, Refactoring: Improving the Design of Existing Code, Addison Wesley Professional, 1999.
- [7] L. Bass, P. Clements and R. Kaman, Software Architecture in Practice (2nd edition), Addison-Wesley, 2003.
- [8] T.R. Malan and D. Bredemeyer, “Less is More with Minimalist Architecture”, IT Professional, IEEE, vol. 04, no. 5, pp. 48, 46-47, September/October, 2002.
- [9] K.S. Barber, T. Graser and J. Holt, “Providing early feedback in the development cycle through automated application of model checking to software architectures”, Proceedings of 16th Annual International Conference on Automated Software Engineering, IEEE, 2001, pp. 341 – 345.
- [10] Joel Spolsky, User Interface Design for Programmers, Apress, 2001; also available as a Web article, <http://www.joelonsoftware.com/uibook/fog0000000249.html>, October, 2001.
- [11] Microsoft Inductive User Interface Guidelines, Microsoft Corporation, 2001, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwui/html/iuiguideelines.asp>.
- [12] R. Molich and J. Nielsen, “Improving a human-computer dialogue”, Communications of the ACM 33, 3 (March 1990), pp. 338-348.
- [13] J. Nielsen, and R. Molich., “Heuristic evaluation of user interfaces”, Proc. ACM CHI'90 Conf, 1990, pp. 249-256.
- [14] John Rieman, Marita Franzke and David Redmiles, “Usability evaluation with the cognitive walkthrough”, Conference on Human Factors in Computing Systems, SIGCHI, ACM, 1995, pp 387-388.
- [15] Martin Fowler, “Language Workbenches: The Killer-App for Domain Specific Languages?”, 2005 <http://martinfowler.com/articles/languageWorkbench.html>