Lessons Learned on Design for Modifiability and Maintainability

Peraphon Sophatsathit
Department of Mathematics, Chulalongkorn University
Bangkok, Thailand
email: sperapho@chula.ac.th

Abstract
Maintainability is perhaps one of the most important aspects of software development in that maintenance costs account for at least 50% of software system lifetime costs. A common maintenance practice is adding new capabilities to satisfy the evolving system dynamics, thus contributing to the ever-growing expenses. This paper discusses some lessons learned from system modifiability that affect software maintainability from the design standpoint. Deciding which design approach is most appropriate for the underlying system requirements plays a major role in software product operability and extensibility.

keywords: design for change, maintainability, modifiability, software IC.

1. Introduction

According to an observation by Intel's Gordon Moore: the number of transistors on new integrated circuits doubles every 18 months. Software development, on the contrary, has lagged considerably behind its hardware counterpart. In the past, software was incorporated into the system as an after-thought. Software cost was virtually non-existent compared with that of hardware. Implementation was difficult, not to mention obsolete documentation by the time the software was in full operation. The advent of Internet connectivity, however, has brought about new paradigms of software implementation and use. Users are no longer limited to hard copy manuals or out-of-date floppy updates. They can directly download myriad of new packages as fast as developers can put out the up-to-minute sources. This, in turn, shortens the lifecycle of software development considerably. New capabilities, accompanied by the latest look-and-feel interface, are constantly required to enhance the existing version. As a consequence, it is inevitable for developers to seek the simplest means for system modification such that they can churn out software as fast as they can.

2. Background and Motivation

The traditional Waterfall Model has since become a thing of the past which, in many cases, falls short of handling modern software development process. The inherent system dynamics has greatly influenced the use of software products. Many software packages obtained from various sources on the Internet can be enhanced, upgraded, or even re-installed with a few mouse clicks. Existing system requirements may not suit tomorrow's applications. Thus, modifications, ad hoc user-interface rearrangement, or quick-and-dirty patches to support new system capabilities are required and often carried out in a hasty manner. Such ill-planned practices have entrenched and hampered most software maintainers' productivity. As a consequence, maintenance costs account for at least 50% of software system lifetime costs [8]. Perhaps the largest maintenance cost for both hardware and software history is the notorious "Y2K" problem.

A number of software development techniques have been established over the years to cope with systematic software construction. New design paradigms were introduced and documented. Despite such efforts, the immaturity of software engineering discipline makes it hard to keep pace with hardware counterparts. This is in part due to the fact that software creation is an abstract process that does not have any scientific experimental support [5]. Other causes include economic issues, technical problems, and the deadline effects.

It is therefore imperative that maintainability be taken into account during the design stage—a provision known as design for change. In so doing, subsequent modifications can be accomplished with minimal time and efforts.

Bearing the aforementioned philosophy in mind, a revolutionary web-based electronic catalogue prototype incorporating text, image, as well as engineering drawings was devised. The main objective is to promote various Small and Medium-Sized Enterprises (SMEs) of automobile, electronics, mold and die-casting industries in the Association of South East Asian Nations (ASEAN) region, where Thailand's Board of Investment (BOI) serves as the Office of the Secretariat.

The remaining sections are organized as follows: Section 3 describes some important design principles and implementation details. Section 4 summarizes the results obtained from overall system redesign. Section 5 discusses some of the lessons learned from the work. Additional work is proposed in Section 6.

3. Design Modifiability Framework

The project employed conventional web-based development approaches such as HTML, CGI (Perl), and minimal Java code to ensure a broad applicability over a wide range of browsers. Standard web page constructs were used, i.e., home page with links to subsequent detailed pages, frame structure, banner, and categorized search. From the inception of the project, the system underwent many open-ended requirements since project data were obtained from six participating countries[1]. Part of the early project requirements and system specifications were established by the principal investigator to serve as the initial development guidelines for this web-based electronic catalogue on a PC-based information system. Such irrelevant software requirements specification (SRS), made up by the developers instead of the actual users, implicated against all known software engineering principles and heralded a disastrous course of system development.

The first *evolutionary* prototype was rapidly developed in the absence of actual input data specifications from all member countries. As a consequence, system construction resulted in a number of inconclusive design issues, namely, incompatible input data format, mismatched attribute categorization of data items, common graphical user interface (GUI) layout, the structure, portability, and search capability supports of the data repository, and data communication among member countries. The revised prototype was based on a small set of historical data. No sooner was the prototype released than various change requests and incompatibility of actual data format overwhelmed the maintenance efforts to the point where patch-up work was no longer feasible. The change traffic in terms of lines of code (LOC) was staggering despite the simplicity of the language used. Maintenance redesign was imminent.

Moreover, delays in project data acquisition, inadequate documentation, unexpected explosion of ill-formed specifications, and hasty patch-ups resulted in an unstable design despite several evolutions of modifications. It was decided that a wheel-reinvention was inevitable.

During the operation of the prototype, concerns of retrieval speed became increasingly significant as the number of data records grew. User-defined searching and sorting modules[2] tailored to suit customized result display formats could not scale up and proved to be too slow to yield satisfactory performance. It was found that early unit test with a small set of records did not reveal any shortcomings of a customized sorting. Each record retrieved from a back-end database engine[3] via embedded SQL statements was pushed onto a hash stack for various sorted display formats. As the number of records and users grew, this arrangement became too high a price to pay. Results obtained from the back-end SQL were rearranged and stored in the hash stack. The operation was unbearably time-consuming. As a consequence, a decision was made to revert such functionalities to the back-end database engine whilst displaying the results in a system-defined (default) format. The move proved to be effective and acceptable. Retrieval time was substantially reduced from 5 minutes to 20 seconds, a fifteen-fold performance gain.

Bearing a mindset for most flexible modifiability, the revamped design was concentrated on loose coupling of various system components and yet maintaining centralized data control according to the following principles:

---

[1] Indonesia, Malaysia, Myanmar, Singapore, Thailand, and Vietnam.
[2] By means of associative arrays in Perl.
[3] Postgres95.

- centralized global constants,
- mnemonic constant representation,
- well-formed modularity,
- dynamic display, and
- flexible operation and support utilities.

The above principles established a set of design framework for future modifiability and maintainability, which subsequently proved to be quite fruitful:

Centralized global constant design enabled different system parameters and constraints modification to be accomplished quickly despite the obvious violation to common coupling principle. Perhaps the sole purpose of this design scheme was to gather string constants, forming a centralized literal pool accessible from everywhere. This literal pool in turn facilitated substitution of existing constants with new ones without having to locate every occurrence of these string constants throughout the project's code. As such, code maintenance was improved to some extent.

Mnemonic constant representation facilitated code maintenance. A direct effect of mnemonic representation was program comprehensibility. Logical TRUE/FALSE conditions were replaced by more readable OK/NO and SUCCESS/FAILURE pairs, depending on the context and identifiers used. With such a simple substitution approach, the mnemonic constant representation, in conjunction with centralized global constants, consolidated a redesign scheme, thus improving program readability considerably.

Well-formed modularity allowed program components to be loosely coupled. Subsequent modifications due to requirement changes could thus be carried out easily. The true benefit of smaller module size was for testing purpose. Unit and integration testing activities were greatly simplified. A number of search provisions could be made effortlessly. For example, categorizing search was implemented by means of a general algorithm operating upon indirect pointer reference (see @record below) rather than simple variable reference, thus eliminating many special purpose categorized modules.

Dynamic display reflected immediate change of display layout whenever a new entry is added to the display template (in the form of mnemonic constant representation and global constants) without having to modify the source program. This flexibility was due, in part, to the choice of language used to implement the display module.

Operation and support utilities furnished a menu-driven application layer for non-technical users to interact with the system. Typical transactions were data import and export to and from external archives, source and data archives, and system documentation. The layered architecture yielded two fold benefits for change maintenance. On the one hand, the underlying algorithmic modifications were kept transparent to user code layer. On the other hand, any change in user-related operations could be accomplished without affecting other layers.

Based on these five modifiability techniques, the redesign process was carried out to rectify the following aspects of the project software:

System architecture

The first mock-up prototype was driven by the functionality of each web page. As such, many system functions were redundant which resulted in duplicated code (with slight variations such as title, caption, number of choices, etc.), bloated file size, and incoherent/inconsistent system modification. Thus, overall functional partitioning of the system requirements specification was employed in order to strive for module independence by means of structural building blocks [7]. Such an undertaking ensured module extensibility and modifiability. Typical partitioning of system functionalities included file handling, I/O handling, database handling, and module inter-connectivity. To maximize module modifiability, certain module-level design weaknesses were avoided, namely, coincidental cohesion, and global atomic variable (with the exception of global arrays). This architectural framework provided design infrastructure, self-containedness, and abstraction for maintenance.

One typical design modification of the first mock-up prototype was redesign of a straightforward field structure (data table) to denote *generic record* which composed of the following fields:

<field_A, field_B, field_C, field_D, field_E>

having the corresponding tiles:

<"Company", "Address", "Country", "Type of Business", "Contact">

Due to non-uniformity of data configuration and availability of the corresponding data, this template (data table) was further classified into smaller subgroups in order to accommodate combination of data groupings. For instance, record subgroup I may contain all the fields shown above, record subgroup II may contain only field_B and field_D, and record subgroup III may contain field_A, field_C, field_D, and field_E. Such subgroup discrepancies were not discovered during the design of the first mock-up prototype owing to the lack of record specification, thus posing considerable maintenance efforts as data requirements changed. The notion of consolidating this "data table" into an orderly structure (generic record) was never anticipated until actual record specification was available. To cope with such unpredictable variations and yet be capable of providing flexibility for change, redesign of the parametric data table was accomplished by means of a list of lists as follows:

```
@record =
(
    [ "field_A", "field_B", "field_C", "field_D", "field_E" ],
    [ "field_B", "field_D" ],
    [ "field_A", "field_C", "field_D", "field_E" ],
);
```

Such a construct furnished greater extent to subgroup modification coverage. Common processing algorithms were consolidated and applied to this generic construct, whereby new subgroups can be added at will without any code modification. The change lessened subsequent maintenance efforts considerably.

The above flexibility of contextual modification had led to the notion of resource generics that encapsulated static and dynamic retrieval, as well as result display. A combination of HTML and Perl script implementation yielded the latest information whenever change in the system data repository took place. The decision on this implementation scheme was made to preserve backward[4] compatibility as older operating platforms were prevalent.

Cohesion

The main objective of this design effort is to maintain the optimal level of cohesion and coupling. Various well-known and acceptable design-level cohesion paradigms [1] were applied to render better comprehensibility and modifiability for code-level maintenance. Upon redesign of record structure, all single non-associated variables previously used to represent each field in the data table were replaced by a list variable. Thus, iterative cohesion was instituted. Moreover, a number of hard-coded back-end (SQL) variables were parameterized to support loose coupling that would otherwise have been tightly bound by Perl variable scoping rules.

Module size

By virtue of the aforementioned flexible system architecture guidelines, module size (measured in LOC for simplicity rather than Function Points as in [4]) was considerably smaller due to the use of a flexible generic interface. Algorithmic parameterized invocation was established to minimize module interdependency. Unfortunately, the number of modules increased slightly, which resulted in higher invocation overhead. Maintenance benefits, however, outweighed the overhead since modularity offered greater module interchangeability.

Literal constants

Many front-end hard-coded constants were replaced by mnemonic literals. All literal names and corresponding values were tabulated in order to identify their roles in module processing. The approach adopted the technique for requirements specification and module interface specification state machine suggested by [6] in the form of a valued-table to facilitate subsequent cross-referencing as follows:

---

[4] Windows 3.1 and Netscape 3.0 client

Table 1 Value-table

Name	Value	Routine appeared
Good	1	kword1
badval	NO_TAB	select
Bad	0	tail
@metaf	“Industr_”, “Process_”, “Country_”	search

Documentation

Besides comments in source code, two documents were created to accompany the project deliverables, namely, the general project document and the user's manual. The former narrated project background, rationale, requirements, high-level design, impact, and benefits, whereas the latter elucidates operational "how-to" procedures for interested users. Many conventional techniques were employed to represent descriptive models and design paradigms. For instance, transformation of information was denoted by Data Flow Diagrams, the high-level (modular) functional processing sequence was represented by a hierarchical calling tree, and the architectural overview was charted by a block diagram.

Style

The importance of coding style is well recognized in the software development community. Insofar as the project’s standard practice was concerned, coding style was one of the top issues from the inception of the project. Fortunately, by virtue of language similarity between C and Perl, it was simple and straightforward to follow standard guidelines established in [3]. Not only does good coding style make source code easier to read, but also lends itself to quicker change.

4. Implementation Results

The first version of the system prototype consisted of three display options, namely, company display, product display, and category search display. The system invocation was triggered by standard http activation procedure, i.e., execution of WWW invocation script (*index.html* in this case). The script in turn invoked a number of cgi scripts beginning with *start.cgi*. Subsequent scripts such as *company.cgi*, *product.cgi*, and *search.cgi* were invoked based on individual user's selection. A diagram of module calling sequence is shown in Figure 1.

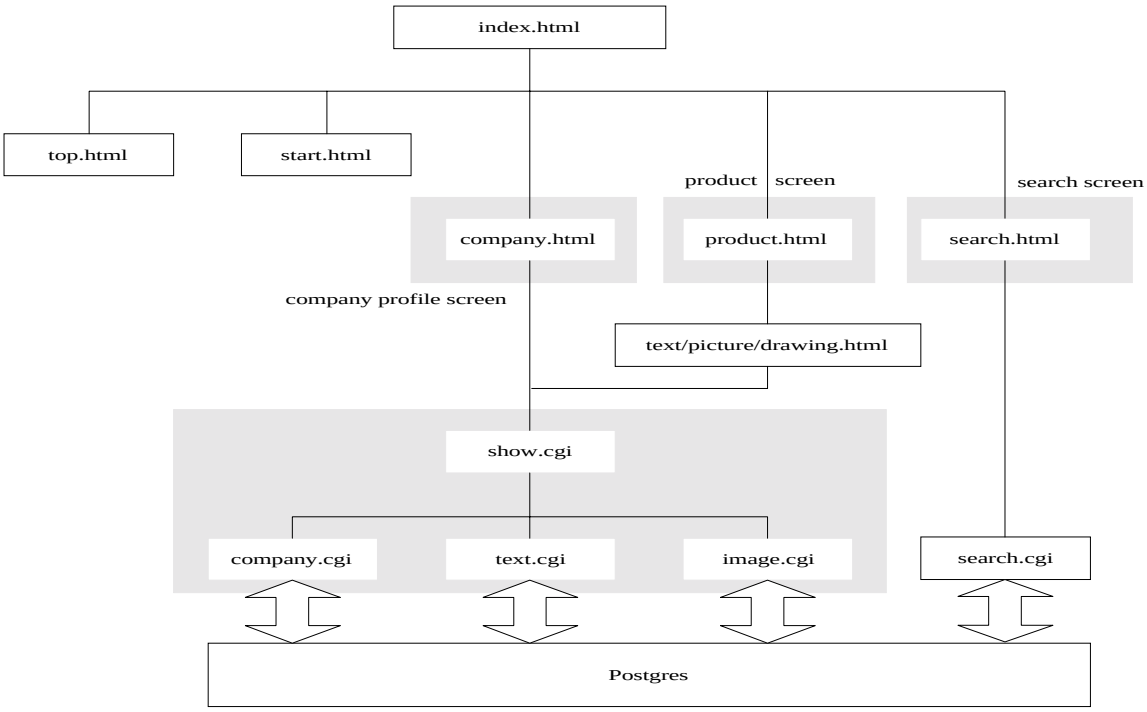

Figure 1

As the system evolved, user's requirements and data changed considerably. Most of the existing scripts were no longer applicable. The product display screen (product.html) was consolidated with the company display screen (company.html) by means of a link in the company's profile screen, dropping the product screen entirely. This was because many pieces of information were incorporated for the completeness and coverage of domestic and foreign participating companies. For instance, product group (of product screen) and business segment (of company screen) are primarily two overlapping functions referencing almost identical sets of data. The redesign screen consolidation forced the entire search procedure to be rewritten to accommodate all the changes. In addition, new requirements on more search selections, in particular, product category and country search options, were incorporated to enhance the previously limited word search capability.

The biggest challenge of the system redesign, perhaps, was the obligation to maintain the original structure so that re-learning on the part of code maintenance would be kept to minimum. By employing the aforementioned modifiability framework, it turned out that the seemingly formidable tasks of preserving original structure, yet rearranging and adding sizable code to the existing source program were relatively straightforward. This was due in part to the richness and elegance of Perl script that made all the changes simple. Consolidation of company and product display screens was achieved in two man-days. Table 2 summarizes major modification activities conducted during functional requirements change.

Table 2 MM for Code Maintenance

Activities	Man-day	Remarks
company & product	2	
search	10	data from foreign companies encompassed different attributes and formats
system table display	1	new features were added
product link	1	embedded in the product description of company profile screen

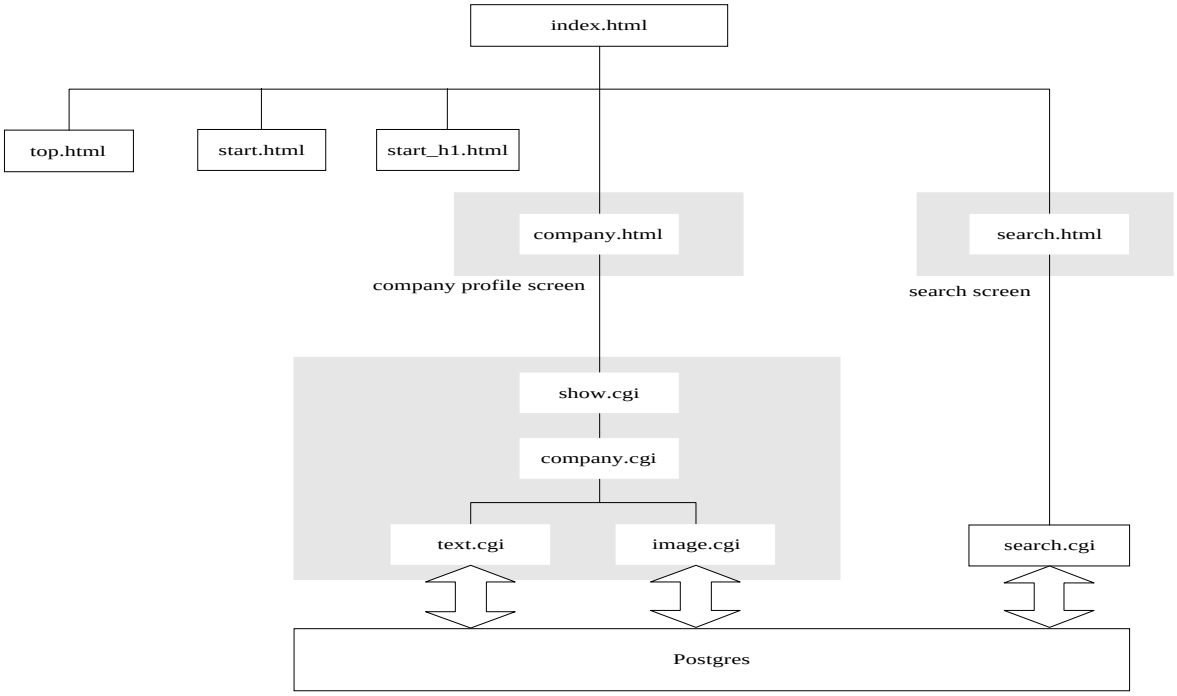

Figure 2

Figure 2 depicts the revamped structure of the original design. It can be seen that the two structures differ slightly, especially in the shaded areas. Detailed modifications are considerably different which could not be seen by virtue of information hiding principle. Many software maintenance aspects mentioned earlier were accomplished to some extent, notwithstanding any major architectural redesign. For instance, change in search module, identifier renaming, code clean-up, and style formatting were accomplished without much difficulty.

5. Impact on Maintainability

The lessons learned from this project suggest some precautionary development activities that must be heeded. They are:

- *Cleanroom development* where errors should have been prevented in the first place;
- *Flexible design for modifiability* so as to cope with future change;
- *Portability considerations* when system will grow out of control under current environment, thus requiring a larger and perhaps different operating environment; and
- *Well-defined/non-evolving user's requirements* to prevent any addendum of ever-growing demands.

As the users became aware of project maintenance upon delivery, such lessons were methodically transferred so that the users would have adequate technical background and hands-on experience[5] to handle future maintenance activities, realize systematic change methodology, be properly trained for technology transfer, as well as appreciate the importance of maintenance.

In addition to the technical aspects of maintenance, it was decided to abolish all new development assistants during the project renovation stage. Brooks' Law [2] was tested on training one new staff for a brief period. It was a re-affirmation of the Law that the added personnel turned out to be a burden rather than a useful aide.

Hindsight analysis also revealed that much of perfective maintenance efforts involved extensive testing, particularly regression testing, to ensure satisfactory performance and correctness. All provisions for

[5] One system administrator and a programmer were trained to carry on the operation and maintenance responsibilities.

modifiability (by the users after delivery) was less effective than expected due to low change traffic and the lack of technical expertise on the users' part. The biggest impact, however, was the development of a new SRS for the production system. The lessons learned made the users aware of software engineering principles as they became involved in various development activities, in particular, requirements on *design for change* were demanded by the users to explicitly spell out. Flexible design was finally recognized and well-accepted as part of the standard practice.

6. Future work

Some maintenance guidelines that can be further investigated are tools and techniques used to facilitate maintenance activities such as visual tools, techniques for better program understandability, coding standards, and new design paradigms. It is envisioned that maintenance tasks for this type of programming could be categorized into two modes, namely, typical maintenance mode and evolutionary maintenance mode. Typical maintenance mode calls for:

- hypertext-style reference between identifiers, parameters, and function signatures;
- calling tree visual aids for code inspection; and
- syntax-sensitive editors.

Evolutionary maintenance mode, on the other hand, encompasses:

- reverse engineering; and
- localized execution during maintenance tests.

As maintenance is no longer an after-thought, design for change is of the utmost important aspect for simplifying maintenance tasks. The ultimate goal of software development and maintenance is to strive for what its hardware counterpart accomplished long time ago—implementation of software IC. Numerous efforts such as the Portable Common Tool Environment (PCTE) and Common APSE Interface Set (CAIS) have been taken to establish a common software interface so as to achieve “plug-and-play” capability.

Acknowledgments

This work was supported by Thailand Board of Investment (http://www.boi.go.th), Office of the Prime Minister, Thailand. The current version was developed by an independent software developer and can be accessed at http://www.asidnet.org.

References

[1] Bieman, James M. and Kang, Byung-Kyoo. *Measuring Design-Level Cohesion.* IEEE Transactions on Software Engineering, Vol. 24, No. 2, February 1998.

[2] Brooks, JR., Frederick P. *The Mythical Man-Month*. Addison-Wesley Publishing Company, 1995.

[3] Cannon, L.W., et al. *Recommended C Style and Coding Standards*. October, 1992.

[4] Furey, Sean. *Why We Should Use Function Points*. IEEE Software, March/April 1997.

[5] Hartmanis, Juris. *On Computational Complexity and the nature of Computer Science—Turing Award Lecture. Communications of the ACM*, October 1994, Vol. 37, No. 10.

[6] Hoffman, Daniel and Strooper, Paul. *Software Design, Automated Testing, and Maintenance*. International Thomson Computer Press, 1995.

[7] van der Linden, Frank J. and Muller, Jurgen K. *Creating Architectures with Building Blocks.* IEEE Software, November 1995.

[8] Sommerville, Ian. *Software Engineering, 5th Edition*. Addison-Wesley Publishing Company, 1996.