The Journal of Systems and Software 82 (2009) 851–861

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Determining factors that affect long-term evolution in scientific application software

Diane Kelly

Royal Military College of Canada, Department of Mathematics and Computer Science, Kingston, Ontario, Canada

ARTICLE INFO

Article history:
Received 2 June 2008
Received in revised form 21 November 2008
Accepted 23 November 2008
Available online 3 December 2008

Keywords:
Scientific software
Characteristics of scientific software development
Long-term software evolution
Empirical study
Model of change

ABSTRACT

One of the characteristics of scientific application software is its long lifetime of active maintenance. There has been little software engineering research into the development characteristics of scientific software and into the factors that support its successful long evolution. The research described in this paper introduces a novel model to examine the nature of change that influenced an example of industrial scientific software over its lifetime. The research uses the model to provide an objective analysis of factors that contributed to long-term evolution of the software system. Conclusions suggest that the architectural design of the software and the characteristics of the software development group played a major role in the successful evolution of the software. The novel model of change and the research method developed for this study are independent of the type of software under study.

1. Introduction

The research described in this paper is a case study of three specific factors affecting long-term evolution of an industrial example of scientific application software. Examples of scientific application software (e.g. Sanders and Kelly, 2008) are software that model weather and ocean currents, support the design of aircraft engines, simulate the load and stress on bridges, provide guidance for orthopedic surgery, and simulate subsystems for nuclear generating units. An important characteristic often shared by scientific software is a lifetime that is measured in decades (Boisvert and Tang, 2001). This suggests that scientific software can provide interesting examples for studies in long-term evolution.

The research method that we describe in this paper is not dependent on the type of software under study. However, the conclusions that we draw for our case study of scientific software are applicable to software that shares the same salient characteristics. We provide a detailed description of the software used for our case study.

In our research method, we blend two ideas to provide a detailed understanding of how our study software responds to changes in its environment. To support our research method, we developed a model of change that includes these two ideas.

The first idea is to categorize environmental changes by their source rather than by their impact on the software. Godfrey and German point out this distinction in their discussion of the difference between maintenance and evolution (Godfrey and German, 2008). For example, the ubiquitous categorization by Swanson (1976) classifies by determining whether the change corrects, adapts, or perfects the software. In our classification, we identify the source of the change based on the area of knowledge that embodies the change. We propose that there are five knowledge domains that represent knowledge important to any software system.

The second idea, that of change filters, is to explicitly include in our model the fact that not all change in the environment actually results in change to the software. For example, a software system designed to be ported to different hardware platforms, will not be have to be changed when a user purchases one of these platforms.

The choice of change filters in our model depends on the goals of the study. In our case, we chose three change filters to support our goals of identifying the impact of specific factors. The choice of the first factor was obvious: how successful evolution depends on the design of the software itself. Significant amounts of research are dedicated to designing software for change and in this case study, we assess the impact of the software design on handling change. Mens et al. (2005a,b) suggest that it is informative to look at the impact of people on successful evolution. We divided 'people' into two groups: those who had direct hands-on experience with the software such as users and developers, and those who did not have hands-on experience but set policies that impact the software. These became our three change filters that we included in our study.

E-mail address: Kelly-d@rmc.ca

The results of our study suggest that the architectural style providing the basic structure of the study software contributed positively to its successful evolution. As well, the characteristics of the user and developer groups had a visible impact on change in the software. Interestingly, there was little visible impact from the group of people who set policy. The skills of the developers and the architectural style of the software absorbed any impact from this group.

From a previous study (Kelly, 2006), our case study software exhibited continued growth that reflected the findings of Godfrey and Tu in their study of Linux (Godfrey and Tu, 2000). Godfrey and Tu found that Linux provides a counter example to Lehman's Second Law of Software Evolution (Lehman and Ramil, 1996). This Law states that the rate of growth of a system declines as the system ages, and that a systems' complexity increases unless work is done to reduce it. The similarity between Linux and our case study software may lie in the system architectures, in that both systems were designed as a set of "engines" that process input data. The research described in this paper further examines the impact of this architecture (amongst other factors) on handling change.

Details of the research are given in the following sections. Section 2 describes salient characteristics of the software under study and its environment. Section 3 provides details of the steps carried out in the study, including identification of sets of changes in the source code, classification of those changes using a concept of knowledge domains, and the introduction of the concept of change filter. Section 4 discusses the results from the classification and the application of the concept of change filter. Section 5 concludes. Appendix A provides data examples from the case study software that illustrate the classification that was done.

2. Description of the case study software

The software used for our case study is considered to have been actively and 'successfully' maintained over the study period of about 20 years. Characteristics of the software and its environment includes experienced and long-resident development groups, conservative management approaches to software development, little documentation such as change records, steady and stable growth of the software, and a software architectural style typical of scientific software. This section expands on these characteristics.

The particular scientific software used in the study simulates thermal hydraulic subsystems in Canadian designed nuclear generating stations. This software was first released in 1975, and is still actively maintained and used. Awards were given to its developers from peers in the Canadian nuclear industry. That, its long life, active maintenance, and its use for safety-related, operational, and design issues in the international nuclear community points to the success of this software. Given this success and the fact that the software and its environment conformed closely to many of the characteristics described in Boisvert and Tang (2001) as being common to scientific software, this case provides an interesting example to examine for long-term evolutionary factors broadly applicable to scientific software.

Four versions of the software were made available for this research. A decision was made to request versions as widely separated in time as possible in order to improve the chances of seeing effects that were most significant over the long-term.

The oldest version available was a hardcopy of a compiler listing dating from 1980. This version is referred to as the base version (T_0). Three electronic source versions released around 2000 were chosen as the target versions, designated T_1, T_2, and T_3. Each target version was developed by a different developer group for a different nuclear station. One version, T_2, included changes to the underlying theoretical models to address a major new direction of intended use for the software. All target versions were ported from mainframes to workstations. Each target version evolved in response to a different set of requirements. Many of the changes were unpredicted. More correctly, they were unpredictable, particularly given the length of time this software has been in use. In 1985, a study by Weiss and Basili on scientific software for unmanned spacecraft found that the most frequent type of change was "an unplanned design modification" (Weiss and Basili, 1985). It is recognized that "many changes actually required are those that the original designers cannot even conceive of" (Weiss and Basili 1985). The design of the study software had to absorb these unpredictable changes. One question we pursued in this research was the impact of the software design on evolution success.

2.1. Design of the case study software

The case study software is designed with what we call a data-driven architecture, shown in Fig. 1. This is a typical architectural style for scientific software (Arnold and Dongarra, 2000). The aim of the architecture is to push as much application information into data, leaving the code as an engine that responds in different ways depending on the data. This also minimizes changes to the source code with respect to anything that can be readily described solely with data.

The architecture includes a "compiler" (input preparation) that transforms the extensive input data into the format required for an internal database. The input data itself is modular, with keywords in the input that indicate the type and quantity of the following data. The modular input data allows substantial flexibility in what the user can submit as data. This flexibility in the case of the study software was built primarily around changes to equipment and configurations of equipment in the station. The "compiler" is structured to handle each input data type. As well, the calculation engine (data processing and analysis of results) is also structured in a way that reflects the entities represented in the input data types. The design of the engine is modular, with each module handling different pieces of equipment. It was found that the size of the software modules varied substantially. There was no commitment to any heuristic of maximum size. From common understanding amongst the developers, the commitment was to preserve the architecture of the software and to preserve the names of the variables. In the study described in Kelly (2006) it was found that 70–81% of variables from T_0 were preserved in name and use in versions T_1, T_2, and T_3. As well, the architecture was preserved by all versions of the software.

The internal database (Fig. 1) is the backbone for the activities carried out by the software. The internal database is structured

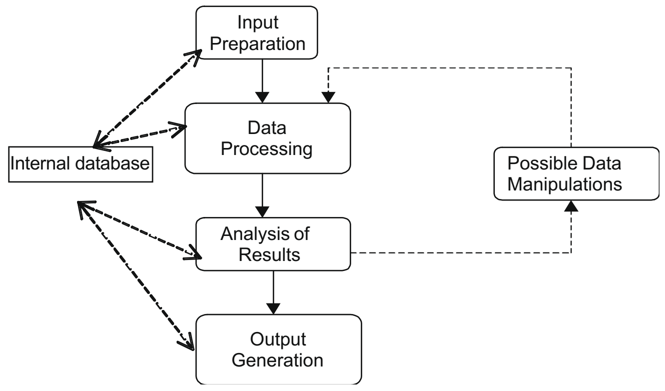

Fig. 1. Data-driven architecture for scientific software.

as a set of FORTRAN common blocks. A common block is a named region of storage that can be referenced by one or more compilable program units. The common block has associated with it a set of variables that have been assigned a storage location within it. Any program unit that has access to the common block has access to the variable names associated with that common block.

The set of common blocks representing the internal database is a core element of the design of the case study software. The variables associated with the common blocks represent quantities that come from initial input to the program (input preparation), interim calculations (data processing, analysis of results, and possible data manipulations), final results for reports, values needed for checkpoints and restarts, values to be shared with other software systems, and values to be transformed to other formats (all various forms of output generation). The contents of the common blocks represent critical steps in the processing carried out by the program. As the procedural part of the code grew, so did the number and size of the common blocks.

The common blocks can be considered as a two-tier hierarchy, with each named common block as the first tier, and the individual variables associated with each common block as the second tier. The details in the two-tier hierarchy were not static over the 20 years studied, but certain aspects of its design structure were surprisingly stable (Kelly, 2006).

2.2. Characteristics of the developer and user groups

Scientific software is primarily developed by scientists who do not identify themselves as software professionals (Boisvert and Tang, 2001; Sanders and Kelly, 2008; Segal, 2005). These scientists may be from application domains such as physics and astronomy, chemistry and chemical engineering, biology, medicine, psychology, electrical engineering, mechanical engineering, and civil engineering. Often scientists knowledgeable in software as well as their application domain are the main users of the software.

Both the developer and user groups associated with all four versions of the software are highly trained scientists and engineers with knowledge of the physical components and the underlying theory of what they are modeling. None had formal training in software development. A period of 2 years was considered minimal for a new user to gain sufficient knowledge to independently undertake studies using the software. This was because of the complexity of the software and of the associated engineering domains. In some cases, the distinction between user and developer was blurred, with some staff performing both roles. Turnover of staff was very low and residency time (length of time working with the software) often matched the time each version of the software was in production. Confidence in the software was high, so use was high. Communication amongst users and developers was ongoing, open, and frequent. There was a surprising consistency in practices and approaches amongst the developers even though nothing was explicitly written down.

2.3. Policy setting

Policy concerning the software was set by upper management and passed down through different levels of management to the developers and users of the software. Policy was usually implicit rather than directly aimed at the software and its development. The software was not the deliverable for the business unit; instead, the studies carried out using the software were the deliverables.

Since the software simulated events at the generating stations, and the nuclear facilities were under the guidelines of a regulator, the need for the software to be available, current, and trustworthy was high. The software had to be kept in step with changes at the stations and confidence in the software had to be maintained.

As observed by Boisvert and Tang (2001), no wide ranging changes were supported simply for the sake of updating to "more modern" software tools or practices. There was an expressly conservative approach taken. At one point, however, all versions of the software were moved from mainframes onto workstations when the mainframes were phased out.

3. Detailed description of the steps in the study

3.1. Reconstructing the modifications to the source code

Studies aimed at improved understanding of long-term evolution of any type of software have depended on detailed historical records being available in some form. As noted by Kemerer and Slaughter Kemerer and Slaughter (1999), this greatly restricts the number of systems that can be studied. Mens et al. (2005a) also mention the need to study long-lived, industrial-sized software systems. Our research method allowed us to examine such a system.

Many studies in long-term evolution look at time-based increments of change to discover mechanisms that describe the change process. For example, Eick et al. (2001) use change management data residing in a version management system to measure factors that indicate code decay. Anton and Potts (2003) glean information about telephone services from 50 years of directories, looking for patterns in growth of functionality. Instead, in this study, the author was interested in the cumulative effects of change rather than the process of change.

Using the base version of the software from 1980 and the three target versions from 2000, a set of modifications needed to be generated for each target version (differences between the base version and each target version). This posed a problem in identifying a suitable unit of modification to be used in the study. Changes to the source code were extensive over the 20-year period. In some code units, every line of source code had been changed – probably many times. This again is in agreement with general observations made in Boisvert and Tang (2001) about scientific software.

Before describing our reconstruction of the modifications to the source code, we distinguish between changes that occur in the software environment and changes that actually manifest themselves in the source code of the software.

The term *modification* will be used for a change that is observable in the software code, and the term *change* will be used otherwise.

A decision was made to use modifications to *data structures* as a surrogate for wider modifications in the software. There has long been evidence that this is a valid surrogate.

Brooks (1987) emphasized the central importance of data structures. In 1980, Yau and Collofello (1980) defined a stability measure for software maintenance based on modifications to variables in a module. They explained that modifications to variables constitute a primitive subset of the maintenance activity. In 2000, Godfrey and Tu (2000) observed that growth patterns of several different metrics all told the same story. The set of metrics included numbers of variables as well as lines of code and numbers of source files. This implies that metrics based on variables alone give a valid profile of the software system as a whole.

3.2. Modifications identified

It was decided that the variables and data structures contained in the internal database (Fig. 1) would be the surrogate for modifications for the entire source code. The reasoning was the following.

Input data for the software, all of which is stored in the internal database, is typically tens of thousands of data items. A large proportion of derived data is also stored in the internal database. Part of the reason for this is for recovery purposes: input data, derived data, and output data are all dumped to recovery files, using the internal database as a source for the dumps. The internal database is sensitive to and reflective of changes in input, output, algorithms used, and interfaces between the modules. In a separate study (Kelly 2006), the ratios of numbers of modules to database structures varied only from 2.0 to 2.5 across all versions studied. The growth in variables in the internal database was remarkably in step with the growth of the code itself. Past research (e.g. Godfrey and German, 2008; Yau and Collofello, 1980) plus this close correlation in growth supported our decision to use the database variables and structures as a surrogate for change to the software as a whole.

A two-level list of modification types was used to help decide the granularity of the modification to be included in the study. The assumption for this list of modification types is that the role of a variable between two versions has not changed.

For the individual variable, the following list was used:

V1: variable deleted;
V2: variable created;
V3: variable renamed;
V4: assignment of initial value changed;
V5: representation of initial value changed (e.g. numeric replaced by a parameter);
V6: variable moved from one common block to another;
V7: declaration of variable changed (e.g. single to double precision).

For the common block, the following list was used:

G1: entire common block deleted;
G2: new common block created;
G3: name of common block changed;
G4: syntax changed (e.g. addition or removal of initiating or terminating characters or keywords; collecting together of particular variable types within the common block);

G5: reordering of variables other than for reason G4.

These two lists of modification types were used to identify a set of modifications between the base version T_0 and the three target versions, T_1, T_2, and T_3. An example is shown in Appendix A.

In total, the number of modifications identified were:

T_0 to T_1: 1352;
T_0 to T_2: 1538;
T_0 to T_3: 1017.

3.3. Developing a change model as a classification scheme

As noted by Lehman and Ramil (2003), software is embedded in an ever-changing environment, to which it must respond. They (Lehman and Ramil, 2003) provide a diagram of their feedback model of software evolution as the software inevitably adapts to changes from different domains. However, in their diagram, exogenous change is characterized as a single input. We propose a model where change coming from the software environment is considered in more detail. Exogenous change is considered as a detailed set of change drivers. We use this set of change drivers to examine how the software responds (possibly differently) to each.

Perry (1994), Belady and Lehman (1976) and Brooks (1987) all suggest possible change drivers. The most detailed discussion is that of Perry (1994). He discusses different "dimensions" of software evolution: *domain*, *experience*, and *process*. We have redefined Perry's dimensions along new axes: that of multiple domains of knowledge, with the realization that experience and process are embodied in each domain of knowledge.

Guindon wrote that (Guindon, 1990) "high level software design is characterized by ... the integration of multiple domains of knowledge at various levels of abstraction." If knowledge changes in any domain that has contributed to the development of the software, then changes in knowledge become drivers for modifications in the software. We use this idea as a basis for our change model. Any drivers for change related to software are based on knowledge coming from multiple knowledge domains. We identified five knowledge domains that embody the knowledge required to write any piece of software. The domains are defined in Table 1. Detailed discussion can be found in Kelly (2004).

It is proposed that modifications for any application software system can be driven by changes in one of the five knowledge domains. In workshop discussions at an IBM CASCON conference in 2003, we explored the applicability of the five knowledge domains to examples of military application software, scientific application software, medical imaging software, and business software and found that the five knowledge domains appeared to be a complete and sufficient set.

We observed that a software system responds differently to changes from one domain than it does to changes from another domain. For example, a system designed to be portable may respond well (i.e., have few modifications) to hardware changes from the execution domain (see below), but not be amenable to changes that increase maintainability. Changes to increase maintainability would come from knowledge in the software domain.

The next step in our study was to use the five knowledge domains to categorize the modifications we identified between the base version and the three target versions.

3.4. Classification by knowledge domain

Each of the modifications identified as described in Section 3.2 were associated with the knowledge domain that drove the change that resulted in the modification.

Three main steps were carried out:

(i) For each modification, a natural language description was provided describing the reason for the change associated with the modification.
(ii) For each modification, a knowledge domain was identified as the driving knowledge domain that initiated the change.
(iii) The number of modifications in each knowledge domain and for each target version were tallied.

To complete the first two steps, extensive interaction with the scientific developers was required. There was little documentation available that described modifications made to the source code.

Table 1
Knowledge domains for software development.

Knowledge domain (KD)	Description
Physical world knowledge	Knowledge of physical world phenomena pertinent to the problem
Theory-based knowledge	Theoretical models that provide the (usually) mathematical understanding of the problem
Software knowledge	Representations, conventions and practices used to construct the computer solution
Execution knowledge	Knowledge of the software and hardware tools used to create, maintain, control, and run the computer solution
Operational knowledge	Knowledge related to the use of the computer solution

Table 2
Example modification types for each knowledge domain.

Knowledge domain	Example modification types associated with domain
Physical	Changes to model events in the physical world; e.g. new equipment in the station, changed parameters for existing equipment, unexpected events in the station that need to be modeled; designing for a new station
Theory based	Changes to implement new or improved theory models; refinement of existing theory models
Software	Coding style changes; changes to support consistency or maintainability, e.g. parameterization of known quantities such as atmospheric pressure and π; clean up of unused code; regrouping of variables, variable name change where semantics are preserved
Execution	Changes due to hardware changes, compiler changes, e.g. change in word size, storage devices, naming conventions allowed, imposed new restrictions
Operational	User requested changes other than to model physical changes at the station; e.g. increased user control over existing functionality, interfaces to other software systems, backwards compatibility, increased performance of the software, increased storage or number of available parameters to allow more detailed studies

Associations were made between the modification and the knowledge domain that drove the change.

Table 2 gives example modification types that were associated with each knowledge domain as being driven by that domain.

Samples for the first two steps using the case study software are given in Appendix A.

The counts and percentages of identified modifications for all three target versions and for each of the knowledge domains are given in Tables 3 and 4.

Before completing Tables 3 and 4, we hypothesized that the modifications to the case study software would exhibit the following characteristics consistently across all versions:

– The architectural design of the code would absorb changes driven by the physical domain due to the ability to model physical entities mostly with changes in data.
– Any changes driven by the theory domain would be visible as modifications in the code, particularly for T_2 where extensive changes were made to the underlying theory.
– Due to the conservative approach to code changes, modifications driven by the software domain would be few in number.
– Because hardware dependencies in the original code design were not encapsulated, modifications driven by the execution domain would be extensive, especially due to the need to port the software from mainframes to workstations.
– Counts of modifications driven by the operational domain would be random due to the differing types of requests from the distinct user population.

Table 3
Counts of modifications identified in each target version and for each knowledge domain.

Counts	T_1	T_2	T_3
Physical world domain	70	350	315
Theory-based domain	156	417	40
Software domain	716	555	200
Execution domain	122	63	440
Operational domain	288	174	22
Totals	1352	1538	1017

Table 4
Percents (individual count over total for each target version) of modifications identified in each target version and for each knowledge domain.

Percents	T_1	T_2	T_3
Physical world domain	5	23	29
Theory-based domain	11	27	4
Software domain	53	36	20
Execution domain	9	4	43
Operational domain	21	11	2

The results in Table 4 did not agree with these hypotheses except for the modifications driven by the theory domain. As expected, version T_2 showed considerably more modifications in the theory domain than the other two versions. However, there were marked differences in the proportion of modifications in all other knowledge domains amongst the different versions that could not be explained without further study. The change model with five knowledge domains driving change was enhanced to include the notion of change filters.

3.5. The effects of change filters

3.5.1. Overview

Whether or not any changes in any one knowledge domain actually manifest as modifications in the source code depends on a variety of characteristics of the software and its environment. For example, changes to physical equipment in the generating station generally can be handled by changes in software data, so modifications to the source code usually are not necessary. In these cases, the design of the source code *prevents* modifications from being made to the source code.

In a similar example, management policy mandated the porting of the software from mainframe computers to workstations. In effect, management policy *advocated* the necessary modifications to be made to the source code.

To model the effect of software design and management policy, we added the concept of “filters” to our change model. Intuitively, filters can advocate changes, hence cause modifications, or prevent changes, hence reduce or eliminate modifications.

More completely, the effect of the filters on change coming from a knowledge domain can result in *no modifications, limited modifications,* or *extensive modifications* to the code itself. The effects of a filter are as follows:

Advocate (strong or weak) – changes are encouraged or caused to be extensive.
Neutral – this filter has no discernable effect on changes either to advocate or to prevent.
Preventor (strong or weak) – changes are discouraged or minimized.
Unknown – the effect of the filter is not known.

The choice of specific change filters depends on the purpose of a study. For our study, we are interested in the effect of the software design and of the people associated with the software, more specifically, people who are policy makers and people who are “hands-on” with the software.

The analysis we carried out matches predictions to observations. We predict the extent of modifications in the software based on the five knowledge domains and on the impact of the change filters. We observe actual modifications to the source code. We then reason about differences between these predictions and observations.

We chose three change filters for our study: *software characteristics, associated population*, and *external policy*. Fig. 2 shows the enhanced change model with five knowledge domains driving change, our three chosen change filters affecting change, and the modifications that ultimately appear in the source code.

The first filter is ‘‘software characteristics’’. For example, does the design of this software accommodate specific types of changes without modification? If it does, then the software characteristic filter would be a strong preventor of modifications to the code for those types of changes. Are specific types of changes handled through well understood and systematic modifications? If so, the software characteristic filter would be characterized as either neutral or a weak advocate of change. Does the software implementation depend heavily on specifics in its environment, such as a type of hardware? Then the software characteristic filter would be characterized as a strong advocate for any changes coming from the execution knowledge domain.

The second filter we chose is ‘‘associated population’’. This is the group of people who have direct hands-on experience with the software, usually as developers and users. Examples of the effects of this filter include experienced and active users who advocate changes to support their use of the software, long-resident developers who implement new ideas to increase the maintainability of the code, designers who have deep understanding of multiple domains and can find the most effective way to make modifications for a change, and a large user group over a long period of time inventing new uses for the software.

The third filter we chose is ‘‘external policy’’. These are stakeholders who are not hands-on with the software but set policy for the environment in which the software system lives. For example, regulators who require adherence to specific standards, customers who set time or financial constraints, managers who set priorities and govern staffing, group leaders who mandate style guides or choose hardware platforms. Each of these factors can contribute to either preventing further modifications to the code (e.g. always running under tight fiscal constraints) or advocating modifications to the code based on changes in the knowledge domain (e.g. adherence to a new quality standard or change of hardware platform).

The filters can be used to make predictions of the amount of modifications that should be seen in the source code due to the influence of that filter.

For our case study software, Table 5 lists the characteristics of the filters that were considered to make predictions.

3.6. Comparison of predicted to observed

An assessment of the environment of the software using each change filter allowed us to predict the effect of the change filter on changes driven by each of the five knowledge domains. These predictions are collected in a table of predictions. The predictions can be compared to observed modifications classified into each of the knowledge domains. This provides data to examine and confirm predictions or, more interestingly, reveal conflicts that invite further examination.

4. Study using predictions from the change filters

The existence of three parallel versions of our study software system provided an opportunity to compare and contrast the outcomes of the predictions and observations. The characteristics of the development environments that are discussed below are those that were consistent over the 20-year period of the study. Table 6 gives the outcome of the analysis done using our specific three change filters and compares these to the observed percent modifications. A discussion is provided in the following sections.

4.1. Physical domain

Changes driven by the physical world are those such as the installation of new equipment, changes in configurations due to aging, accidents, or upgrades, plans for new stations, and new rules for operating procedures. Not all those changes appeared as modifications to the code.

Examining the influence of the external policy filter means examining the policies of those stakeholders who set policy but who are not normally hands-on with the code. In our case study, the influence of regulators, upper management, and other customers is to encourage the software to reflect all changes in the station that are relevant to the software. The software is simulating the station and reasonable correspondence to the physical characteristics of the station is required. As a result, the external policy filter strongly advocated changes coming from the physical world to be expressed in the software. This is true for all three target versions.

Associated population is comprised of stakeholders who are hands-on with the code, generally users and developers of the soft-

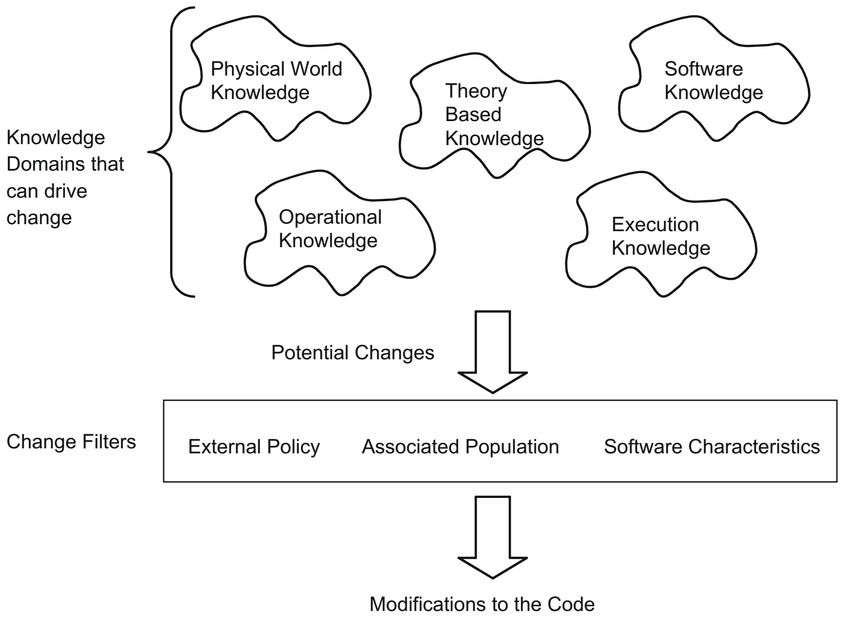

Fig. 2. Change model.

Table 5
Characteristics considered to predict the effect of change filters.

Change filter	Characteristics considered to predict effect of the change filter
Software characteristics	– encapsulation
	– software characteristics
	– architecture
	– naming conventions
	– modularization
	– complexity
	– consistency
	– repeated structures
	– stability
	– supporting documentation
Associated population	– length of residency of individuals
	– associated population
	– activity level of group
	– size of population
	– comfort level with software
	– trust in software
	– evident skills in knowledge domain
	– common culture amongst individuals
	– learning curve for software use or maintenance
	– level of communication amongst individuals
External policy	– coding style mandated
	– external policy
	– process and procedures mandated
	– regulatory standards mandated
	– changes made in operating environment or tools
	– changes made in staffing
	– fiscal policies
	– time priorities
	– prioritization of changes from different knowledge domains
	– policies concerning use of software
	– changes to purpose of software
	– software qualities important to business
	– decisions related to expected lifetime of the software

ware. For all three target versions, the developer groups were knowledgeable in the physical world domain. However, developers in this group generally did not actively initiate changes coming from the physical world – they were reactive in their position, and hence neutral in their influence on changes from the physical world. The job of the users however was to accurately model the stations with the software. Hence, users advocated changes from the physical world. The amount of change influenced by the users was proportional to some extent to the length of time the software had been available for use and the amount of experience of the user group. In this, there were differences amongst the three target versions. Versions evolving into T_1 have been in service for about 20 years. Versions evolving into T_2 have been in service about 10 years. Versions evolving into T_3 have been in service considerably less than 10 years. Hence the associated population filter was estimated as having increasing effects as shown in Table 6.

The original design of the software was set up to handle changes in equipment, operating parameters, configuration of equipment, numbers and types of equipment, and so on, all within one station. The original design was not set up to handle the changes required to model an entire new station. Target version T_1 was used for one of the original stations. Version T_2 and T_3 included changes for a new station. The prediction is that the software characteristics filter is a preventor for change for T_1 and an advocate for change for T_2 and T_3.

4.2. Theory domain

A general observation for scientific computational software (Boisvert and Tang, 2001) is that sweeping changes are not done without good reason. External policy in our case study generally was conservative with respect to sweeping changes to the software, including changing the underlying theory. This was evident for target versions T_1 and T_3. However, a mandate was put forth to make major changes to the underlying theory for target version T_2. Hence the prediction for the influence of the external policy filter is neutral for T_1 and T_3 but a strong advocate for T_2.

The associated populations for all three target versions were knowledgeable in the theory-based domain. However, the developers and users generally did not advocate sweeping changes to the theory, so were usually neutral in their effect on the modifications. However for T_2, a developer was added to the team specifically to implement the theory changes mandated by external policy. As a result, the influence of the associated population for T_2 is to strongly advocate change in this knowledge domain.

The software was not designed for sweeping changes to the underlying theory. Theory changes to individual components that model pieces of equipment generally were handled through input data or were encapsulated in a small number of modules and data structures. As such, the software characteristics filter is predicted to be a weak advocate for localized theory changes and a strong advocate for sweeping fundamental theory changes.

4.3. Software domain

External policy can be described as neutral, or even a weak preventor, since sweeping changes to the software were not encouraged. Because of the complexity of the application domain itself success for the software benefits from stability in the software itself. This tendency is discussed in Boisvert and Tang (2001). This is true for all three target versions.

The associated population for the software domain is the developer group. The resident time for the development staff for T_1 was long. At least one developer was present the entire interval between the base version and the release of T_1. Tasked with providing primary user support for T_1, this developer was interested in long-term maintainability of the code, given his own long-term interest in working with the code. There is evidence that this developer carried out extensive changes that he felt would support maintainability. His influence is to strongly advocate modifications to the code. The average resident of time of the key developer for T_2 was shorter than that for T_1. Hired to do theory-based modifications, his interest in modifications due to software changes was small, and only carried out as necessary to support theory-based modifications. His influence for T_2 is neutral. For T_3, the primary knowledge areas of the developer population seemed to be physical world and operational. Software related modifications were carried out only as necessary. The influence for T_3 is neutral.

The structure of the base version is modular and well documented. As shown by a related study (Kelly, 2006), even though the software grew substantially as measured by lines-of-code, the underlying architecture was preserved and the overall design of the software supported change in a consistent and obvious way. This was true for all three target versions. The influence of the software characteristics on change is predicted to be neutral.

4.4. Execution domain

All three target versions were influenced by a decision to move the software system from mainframe computers to workstations. The predicted influence of external policy is to strongly advocate change.

For T_1 and T_2, staff with execution domain knowledge were hired to help port the codes. The goal of this staff was to minimize the effects of the porting exercise. This staff worked closely with the existing development staff and gained considerable software knowledge specific to the system. Their efforts can be characterized as a weak preventor of modifications due to execution domain changes. The strengths of the development staff for T_3 with respect to porting code is unknown.

Changes coming from the execution domain impact compilers, operating systems, word size, file and directory structure, tool support, and type and extent of memory storage. The code in the base version contained extensive platform-dependent constructs. Little attempt was made to contain this code. As a result, the predicted effect of the software characteristics filter is a strong advocate of modifications for all three target versions.

4.5. Operational domain

Because the software was a key component in safety studies, there was an understanding from management that users were provided with whatever was needed to get the job done. The influence of the external policy filter on changes from this domain for all three target versions is predicted to be a strong advocate for modifications.

The associated population for the operational domain is the user population. The population for all three target versions were highly educated, motivated, capable, long residence time, and well trained. As a result, they had confidence in the software and used it extensively. The difference between the three target versions is time in production. T_1 was in production for 20 years, and had a user group of about 50 people on a steady basis, sometimes increasing to as much as 200. T_2 was only available for use about 10 years, but drew on the same user group as T_1. T_3 was in use less than 10 years with a much smaller user group. The predicted influence of this filter is documented as shown in Table 6.

The data-driven architecture of the software facilitates extensive user control over the execution and the output of the software. The design also provides a systematic template for adding new features. The expectation is that the software filter is either neutral or a weak preventor of modifications based on changes in this domain for all three target versions.

4.6. Discussion of comparison of predictions to observed

The filters chosen for this study are not perfectly independent. The effects of the filters certainly overlapped in such cases where management policy resulted in the hiring of an individual who modified the software in order to satisfy mandated policy. However, the characterization of the effects of each filter can be made independently. It was in the analysis of the results where independence had to be considered. Fortunately, there was only one case where there was a strong interaction between filters. If the interaction were more wide-spread, then a different set of filters would have had to have been considered.

The last column in Table 6 shows either *yes* or *no* to indicate whether the predictions from each change filter correlates with the observed percent modifications in each knowledge domain. For example, the percent modifications for the physical world are T_1: 5, T_2: 23, and T_3: 29. The percent modifications for T_1 is small indicating something acted as a strong preventor for change. The percent modifications for T_2 and T_3 are similar to each other and larger than that for T_1, indicating that something acted as an advocate for change for both target versions. The predictions for the software characteristics filter exhibits this profile for the three target versions. In other words, the overriding effect on change coming from the physical world was the design of the software for all three target versions. The following sections discuss these comparisons in more detail.

4.6.1. Physical domain

The predictions from the analysis with the three change filters for the physical domain shows a correlation of the software characteristics change filter with the observed percent of modifications from this domain. The effects from either external policy or associated population appear to be overshadowed by the design of the software. What appears to be true is that the design of the software is successful for the routine and/or predicted changes at the station. The percent of modifications in T_1 in this knowledge domain are small. However, the percent modifications in this knowledge domain in T_2 and T_3 are much larger. Both these two versions included the addition of new stations. The software design could have been changed to accommodate such changes. Apparently such modifications to the design had been discussed by the developer groups of T_1 and T_2. The decision was not to do the modifications since they were complex and lengthy. The addition of a station in comparison was infrequent and the modifications for the addition were systematic and well understood by the experienced developer groups. This is an example where a quantitative metric of change belies careful design considerations.

Table 6
Observed modifications versus predictions using change filters

Knowledge domain	Change filter	T_1	T_2	T_3	Change filter that agrees with measured modifications
Physical world	External policy	Strong advocate	Strong advocate	Strong advocate	No
	Associated population	Strong advocate	Strong advocate	Weak advocate	No
	Software characteristics	Strong preventor	Strong advocate	Strong advocate	Yes
	Measured % modifications	5	23	29	
Theory based	External policy	Neutral	Strong advocate	Neutral	Yes
	Associated population	Neutral	Strong advocate	Neutral	Yes
	Software characteristics	Weak advocate	Strong advocate	Weak advocate	Yes
	Measured % modifications	11	27	4	
Software	External policy	Neutral	Neutral	Neutral	No
	Associated population	Strong advocate	Neutral	Neutral	Yes
	Software characteristics	Neutral	Neutral	Neutral	No
	Measured % modifications	53	36	20	
Execution	External policy	Strong advocate	Strong advocate	Strong advocate	No
	Associated population	Weak preventor	Weak preventor	Unknown	Yes
	Software characteristics	Strong advocate	Strong advocate	Strong advocate	No
	Measured % modifications	9	4	43	
Operational	External policy	Strong advocate	Strong advocate	Strong advocate	No
	Associated population	Strong advocate	Less strong advocate	Weak advocate	Yes
	Software characteristics	Neutral	Neutral	Neutral	No
	Measured % modifications	21	11	2	

4.6.2. Theory domain

Predictions of modifications from this knowledge domain are correlated with all three change filters. There was an interaction of the three change filters to produce the extent of modifications observed in T_2. Fundamental changes to the underlying theory were mandated by policy makers (external policy) and staff knowledgeable in the theory-based domain were hired to carry out the changes (associated population). The design of T_0 accommodated localized theory changes, e.g. for individual pieces of equipment, but was not amenable to extensive changes to the fundamental equations. The effects of the choice to use T_0 as a base for the new theory rather than starting from scratch is seen in the extent of modifications apparent in T_2 as compared to T_1 and T_3. The changes to accommodate the new theory in T_2 were not systematic nor well understood. T_1 and T_3 both had localized theory-based modifications generally related to changes from the physical domain. The higher percent of modifications in T_1 as compared to T_3 is likely explained by the longer production life of T_1. In both cases, the software characteristics of the design seem to accommodate localized theory-based changes well. Changes to T_2 related to the fundamental theory were still ongoing at the time of the study. The results of our analysis suggest that T_0 may not have been a good choice of base for the new theory.

4.6.3. Software domain

The change filter best correlated with the observed modifications is the associated population filter. In other words, the characteristics of the developer group had more impact on modifications to the code than either software characteristics or external policy. The comparison of T_1 to T_2 and T_3 is interesting in that the focus is on the effect of the main developer associated with T_1. Over half the modifications observed in T_1 were in the software domain. Some of these modifications were related to breaking large modules up into smaller ones, and introducing intermediate data structures to handle new functionality. In a related study of the stability of the software over time (Kelly, 2006), it is suggested these changes were not necessarily for the best, even though all were well meaning. The belief that smaller modules means better design actually disrupted the long standing structure of the code. Median fanout increased only for T_1, possibly increasing cognitive load on the developers maintaining this version. More significantly, the percent variables corresponding to vocabulary in the physical world or theory dropped to its lowest value in T_1 as compared to the other versions. Variable names that correspond to understood quantities in the physical or theory-based domains are important to the developer typically maintaining this software, since these developers are scientists, not software specialists. The decrease of variable names that are immediately recognizable from the scientist's knowledge domain would have a negative impact on the scientist-developer understanding the code. The software changes in T_1 may have a significant impact on the maintainability of the code.

4.6.4. Execution domain

The associated population appears to have had an overwhelming effect on the percent of modification observed from the execution domain. In 1975, it would have been impossible for the software designers to foresee the type and extent of changes coming from the execution domain. The software was written to make use of hardware at the time in the most efficient way to solve the application problem at hand. However, the impact in percent of modifications seems to be offset by staff knowledgeable in porting software. Surprisingly, the measured percent of modification due to platform change was extremely small (4% and 9%) when carried out by knowledgeable developers. This observation seems to indicate that hardware dependence may not necessarily be of as much concern with this type of software as other factors such as the knowledge level and experience of the associated population.

4.6.5. Operational domain

The associated population, embodied as an active user group over a particular length of time, correlates best with observed percent of modifications. The data-driven design of the software either absorbs user requests, particularly observed in T_3, or renders changes into systematic patterns that are well understood by the long-term experienced developers. In discussions with the developers, there seemed to be a shared understanding that supported how to make modifications driven by this knowledge domain.

4.6.6. Overall comments

It is interesting to note that in this analysis the design of the software was the overriding cause of modifications for only one knowledge domain. In that one case (physical world), it was well known that the software was not designed to suppress a particular type of modification, and a conscious decision was made to leave it as such. For all other knowledge domains, associated population was a major factor in percent observed modifications. In this particular study, the skill and experience of the people had a major effect on the modifications to the software. For only one knowledge domain (theory based), did external policy have a role, and that only partially, in being the controlling factor in the modifications observed in the code. Again, this seems to point to the importance of the associated population. By inference, we can observe that the characteristics of the design of the software was generally not the overriding factor in the percent of observed modifications over the 20-year study period. This would imply that the original design of this software was well suited for its purpose.

5. Conclusions

Scientific software often has a lifetime of decades. There has been little research into what supports successful long-term evolution of this type of software. Domain specific software engineering research is needed (Vessey and Glass, 1998). The first step in domain specific research is to characterize the software and its development environment. As part of that characterization, this paper describes a research method developed to examine the environmental impact on long-term evolution of an example of industrial scientific software. The method allows analysis of the software without complete change records being available. The method required deep understanding of the development environment and the source code itself, but provided an objective tool to view the impact of different environmental factors. The research method itself is not specific to the type of software in our case study, but the results from our research provides further understanding of the development environment of scientific type software.

For the scientific software examined, the analysis suggests the original design of the software, a data-driven architecture, successfully supported changes over the long-term without major effort to maintain the design. This concurs with findings by Godfrey and Tu (2000) in their research on Linux, which also has a data-driven design. Software designed as a collection of engines driven by data input appears to support successful long-term evolution without extensive effort to offset increasing complexity. This is counter to Lehman's Second Law of Software Evolution (Lehman, 1996).

The data-driven software architecture used in our case study software was appropriate for the successful evolution of the software over the 20-year period studied, in that changes were generally absorbed or minimized by the design of the software.

The effects of associated population appears to be dominant over the 20 years, both in the activity of the user groups and in the skill level and experience of developers. This may be a key contributor to the success of this software, and may be an important factor for all scientific software.

External policy played the smallest role in affecting changes to the software. Whether this is typical of this class of scientific application software would have to be confirmed by further studies.

The analysis method used in this study combined two fundamental concepts, that of five knowledge domains being sources of change, and that of change filters impacting changes that are manifest as modifications in the source code. This allowed the examination of filters of interest, in this case, the architectural design of the software, the hands-on population of users and developers, and management policy. This also allowed the flexibility of assessing the individual response of each filter for each knowledge domain.

In addition, there are few descriptions available of the development environment of an example of industrial scientific software. This case study provides a detailed description of one such environment and a data point for future research.

Acknowledgements

This work is funded by NSERC (National Science and Engineering Research Council) and ARP (Canadian Department of National Defense Academic Research Program). My thanks to the anonymous reviewers for providing helpful comments on a previous version.

Appendix A. Example of a set of changes between base version T_0 and version T_1

A.1. Example data groupings from T_0

/FLAGS/ LDFOUT, LCFOUT, LDEMIF, LCKPTF, LPLOTS, LSFILE

LOGICAL LDFOUT, LCFOUT, LDEMIF, LCKPTF, LPLOTS, LSFILE

/PARMS/ ITMAX, ISLPRC, ISLPTS, ISLPSS, IFMDL, DELT, EPSH, EPSP, EPSW, EPSQ, PFAC, IPRINT, TBLPOP, TBSPOP, TTMAX, NQUADS, LSURGE, IMFLX0, IPDMSD, IPDMLD, IFPRFX, JPRINT, IMODES, IAOVRL, MSFLAG, OO(6)

LOGICAL LSURGE

A.2. Example data groupings from T_1

LOGICAL LDFOUT, LCFOUT, LDEMIF, LCKPTF, LSFILE, LSSRUN, LTRRUN, LTREAD, LTDUMP, LTRED2, LFORCE, LBNDOU

/FLAGS/ LDFOUT, LCFOUT, LDEMIF, LCKPTF, LSFILE, LSSRUN, LTRRUN, LTREAD, LTDUMP, LTRED2, LFORCE, IDTSTP, NOWDPF, TTMAX, TBLPOP, TBSPOP, TBPPOP, TBFPOP, LBNDOU, LDUMMY(5)

LOGICAL LDRIFT, LTFNPW, LBNDMV

/PARMS/ ITMAX, ISLPRC, ISLPTS, ISLPSS, IFMDL, IPRINT, JPRINT, NQUADS, IMFLX0, IPDMSD, IPDMLD, IMODES, IAOVRL, MSFLAG, LDRIFT, NOPSRH, IPFFGC(3), IPFFGB(3), LTFNPW, DELT, EPSH, EPSP, EPSW, EPSQ, PFAC, HPFMLT, BPFMLT, PPFMLT, EPSGUS, LBNDMV, OO(4)

A.3. Changes identified between T_0 and T_1

FLAGS: 17 changes

C4: syntax changed at grouping level (LOGICAL statement moved to preceding position);
V1: 1 variable deleted (LPLOTS);
V6: 3 variables moved here from PARMS (TTMAX, TBLPOP, TBSPOP);
V2: 12 new variables.

PARMS: 14 changes

C4: syntax changed at grouping level (LOGICAL statement moved to preceding position);
C5: existing variables in grouping reordered (other than syntax requirement);
V1: 1 variable deleted (LSURGE);
(one variable moved elsewhere from here (IFPRFX) – not counted here);
V2: 9 new variables;
V6: 1 variable moved here from another grouping (LDRIFT);
V7: declaration of OO changed (see Tables A.1 and A.2).

Table A.1
Natural language descriptions to identify changes that drove the modifications.

Grouping name	Reason for modification identified at grouping level	Reason for modification identified at variable level
FLAGS	C4: compiler requirements for a new version of FORTRAN	V1: (LPLOTS deleted) functionality replaced by counting number of plot variables submitted through input V6: (TTMAX, TBLPOP, TBSPOP moved here from PARMS) logical regrouping of variables V2: (12 new variables) 4 variables for time function capability: new functionality for user 3 variables for force calculation: increased capability for existing function 2 variables for checks on run status: new functionality for user 1 variable for plot function: new functionality for user 1 variable for bundle movement: modeling changes at the station 1 variable added to pad end of grouping and support more flags being added
PARMS	C4: compiler requirements for a new version of FORTRAN	V6: 1 variable moved to this grouping from (DRIFT): regrouping decision by programmer
	C5: declaration of logical variables moved to start of data grouping: requirement for new compiler	(one variable moved elsewhere – not counted here)
	C5: reordering required for new compiler: integers first, logicals next, reals last	V1: 1 variable deleted; surge tank model changed
		V4: size of buffer variable reduced; making use of facility for planned change V2: (9 new variables) 1 new variable for modeling bundle movement 1 counter for warning messages to user 1 counter for time function facility for user 2 flags for peaking factor calculations 3 quantities calculated from reactor power 1 variable for user to control size of minimum entry in Gaussian matrix

Table A.2
Assignment of knowledge domain that drove the change.

Grouping name	Reason for modification identified at grouping level	Reason for modification identified at variable level
FLAGS	C4: compiler requirements for a new version of FORTRAN (execution domain)	V1: (LPLOTS deleted) functionality replaced by counting number of plot variables submitted through input; not requested by user (software domain) V6: (TTMAX, TBLPOP, TBSPOP moved here from PARMS) logical regrouping of variables by programmer 3 × (software domain) V2: (12 new variables) 4 variables for time function capability: new functionality for user 4 × (operational domain) 3 variables for force calculation: increased capability for existing function 3 × (theory-based domain) 2 variables for checks on run status: new functionality for user 2 × (operational domain) 1 variable for plot function: new functionality for user (operational domain) 1 variable for bundle movement: modeling changes at the station (physical domain) 1 variable added to pad end of grouping and support more flags being added (software domain)
PARMS	C4: compiler requirements for a new version of FORTRAN (execution domain) C5: declaration of logical variables moved to start of data grouping: requirement for new compiler (execution domain) C5: reordering required for new compiler: integers first, logicals next, reals last (execution domain)	V6: 1 variable moved to this grouping from (DRIFT): regrouping decision by programmer (software domain) (one variable moved elsewhere – not counted here) V1: 1 variable deleted; surge tank model changed (theory-based domain) V4: size of buffer variable reduced; making use of software facility for planned change (software domain) V2: (9 new variables) 1 new variable for modeling bundle movement (physical domain) 1 counter for warning messages to user (operational domain) 1 counter for time function facility for user (operational domain) 2 flags for user to control peaking factor calculations (operational domain) 3 quantities calculated from reactor power 3 × (theory-based domain) 1 variable for user to control size of minimum entry in Gaussian matrix (operational domain)

References

Anton, Annie I., Potts, Colin, 2003. Functional paleontology: the evolution of user-visible system services. IEEE Transactions on Software Engineering 29 (2), 151–165.

Arnold, Dorian C., Dongarra, Jack J., 2000. Developing an architecture to support the implementation and development of scientific computing applications. In: IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software, October 2000.

Belady, L.A., Lehman, M.M., 1976. A model of large program development. IBM Systems Journal 15 (3), 225–252.

Boisvert, Ronald F., Tang, Ping Tak Peter (Eds.), 2001. The Architecture of Scientific Software. Kluwer Academic Publishers.

Brooks, Fred, 1987. No silver bullet. Computer, 9–19.

Eick, Stephen G., Graves, Todd L., Kerr, Alan F., Marron, J.S., Mockus, Audris, 2001. Does code decay? Assessing the evidence from change management data. IEEE Transactions on Software Engineering 27 (1), 1–12.

Godfrey, Michael W., German, Daniel M., 2008. The past, present, and future of software evolution. In: Invited Paper, International Conference in Software Maintenance, September 2008, 10pp.

Godfrey, Michael W., Tu, Qiang, 2000. Evolution in open source software: a case study. In: Proceedings of the International Conference on Software Maintenance, pp. 131–142.

Guindon, Raymonde, 1990. Knowledge exploited by experts during software system design. International Journal of Man–Machine Studies 33, 279–304.

Kelly, Diane, 2006. A study of design characteristics in evolving software using stability as a criterion. IEEE Transactions on Software Engineering, 315–329.

Kelly, Diane, 2004. An exploration of evolutionary change in an example of scientific software. Ph.D. Thesis, Royal Military College of Canada.

Kemerer, Chris, Slaughter, Sandra, 1999. An empirical approach to studying software evolution. IEEE Transactions on Software Engineering 25 (4), 493–509.

Lehman, Meir M., Ramil, Juan F., 2003. Software evolution – background, theory, practice. In: Integrated Design and Process Technology, IDPT-2003, USA, October 2003.

Lehman, M.M., 1996. Laws of software evolution revisited. Lecture Notes in Computer Science. <http://www-dse.doc.ic.ac.uk/~mml/feast/papers/pdf/556.pdf>.

Mens, Tom, Buckley, Jim, Zenger, Matthias, Rashid, Awais, 2005a. Towards a taxonomy of software evolution. Journal of Software Maintenance and Evolution: Research and Practice 17 (5), 307–322.

Mens, Tom, Wermelinger, Michel, Ducasse, Stephane, Demeyer, Serge, Hischfeld, Robert, Jazayeri, Mehdi, 2005b. Challenges in software evolution. In: Proceedings of the International Workshop on Software Evolution, 10pp.

Perry, D.E., 1994. Dimensions of software evolution. In: Proceedings of the Conference of Software Maintenance. <http://citeseer.nj.nec.com/82523.html>.

Sanders, Rebecca, Kelly, Diane, 2008. Scientific software: where's the risk and how do scientists deal with it? IEEE Software (Special Issue).

Segal, Judith, 2005. When software engineers met research scientists: a case study. Empirical Software Engineering 10, 517–536.

Swanson, E.B., 1976. The dimensions of maintenance. In: International Conference on Software Engineering, San Francisco, pp. 492–497.

Vessey, I., Glass, R., 1998. Strong vs. weak approaches to systems development. Communications of the ACM 41 (4), 99–102.

Weiss, M. David, Basili, R. Victor, 1985. Evaluating software development by analysis of changes: some data from the software engineering laboratory. IEEE Transactions on Software Engineering 11 (2), 157–168.

Yau, Stephen S., Collofello, James S., 1980. Some stability measures for software maintenance. IEEE Transactions on Software Engineering 6 (6), 545–552.

Diane Kelly is an Assistant Professor in the Department of Mathematics and Computer Science at the Royal Military College of Canada (RMC). She has worked in industry in a wide variety of roles in safety-related software environments, from programmer to project leader, trainer to QA advisor. She has a Ph.D. and M.Eng. in Software Engineering from RMC, a B.Sc. in Mathematics and a B.Ed. in Mathematics and Computer Science, both from the University of Toronto, Canada.