Information System Evolvability, Feedback and Pattern Languages

Stephen Cook[a], Rachel Harrison[b] & Paul Wernick[c]

21st June 2006

Abstract

Information systems for business are frequently heavily reliant on software. This paper identifies two important feedback-related effects of embedding software in a business process. Firstly, the system dynamics of the software maintenance process can become complex, particularly in the number and scope of the feedback loops. Secondly, responsiveness to feedback can have a big effect on the evolvability of the information system.

In this paper we explore ways to provide an effective mechanism for improving the quality of feedback between stakeholders durng software maintenance. The difficulty of devising modelling languages that cover the concerns of all stakeholders is illustrated and the kind of planning process needed to achieve consensus on the way ahead is also considered.

Understanding can be improved by using representations of Information Stystems that are both service-based and architectural in scope. The conflicting forces that encourage change or stability can be resolved using patterns and pattern languages. We describe a morphology of Information Systems pattern languages to facilitate the identification and reuse of patterns and pattern languages.

1 Introduction

Over the last 30 to 40 years, organisations of every kind and size have gradually become more dependent on information systems for their success, and it is now almost ubiquitous that those information systems

[a] Applied Software Engineering Research Group, School of Systems Engineering, University of Reading, UK

[b] Stratton Edge Consulting, Gloucestershire, UK

[c] Centre for Empirical Software Process Research, School of Computer Science, University of Hertfordshire, UK

depend, often critically, on software [1]. Over the same period, the performance and reliability of IT hardware have improved dramatically and the functionality that software can offer has also expanded. Consequently users' expectations of information systems have increased, particularly in terms of systems' quality.

The increased dependency of business processes on their supporting information systems raises particular issues when adaptive change is required. Regardless of whether a specific change originates within the business process, or is a consequence of technological developments or some exogenous cause such as government regulation, it is likely to have an impact on both the business process and the information system. However, these impacts will often be asymmetrical; i.e. a trivial change in one may require a major re-engineering of the other. These differential impacts may be difficult to anticipate.

Consequently, the issue of information system evolution is becoming increasingly important for many organisations. There is a widespread perception that software is difficult to adapt, particularly in the context of Internet-related changes in business methods. Organisations increasingly want agile information systems that can co-evolve with business systems [2].

These pressures are generating a growing need to understand how and why information systems become misaligned over time with the business processes that they were intended to support. However, this need is often neglected or even unrecognised. The immediate tasks of keeping existing systems running and delivering urgently required improvements often absorb all the available resources in an IT organisation — systems in this state are sometimes described as 'support-bound'. The medium- and long-term processes of software evolution are often ignored, even though they may be exacerbating day-to-day problems. In this paper we will show how the concepts of evolvability and feedback in information systems impinge on each other, both in theory and practise. We examine the problem of raising awareness of information system evolution within organisations, and suggest that people need:

- models to help them understand how evolution processes work.
- languages and processes for communicating the architecture of information systems.

In both cases, solutions are needed that are adaptable to both technology- and business-oriented perspectives.

This paper presents conjectures that indicate directions for further research. As these conjectures become more refined they should be tested in carefully designed experiments or studied in suitable case studies. Case studies and simulation models [3], [4] are invaluable for exploring the complex interactions that are common in feedback processes and for eliciting experiences from software engineers and other stakeholders. Some simple examples of these techniques are presented in later sections, taken from various case studies.

2 The FRS Case Study

The principal case study in this paper concerns a financial management information system (MIS) called the Financial Reports System (FRS). It is a specialised, bespoke, intranet-based MIS for scientists and engineers who are managing projects in a laboratory. The FRS produces online reports showing income and expenditure. Users, who are predominantly project managers, request and receive reports through their web browser (although originally (in 1984) the system produced reports centrally). The laboratory's many other requirements for financial enquiries and reports are handled by other systems. The objectives of the FRS case study were to:

- pilot the study of evolvability issues in real-world information systems
- collect examples of evolvability problems in a long-lived management information system
- explore concepts and techniques for helping designers to understand and improve system evolvability.

The main data inputs to the FRS concern budgets, income, expenditure, assets and resources. The main outputs are dynamically-generated reports that can be displayed in a web browser. The reports provide snapshots of the current financial state of a project. The aim of the FRS is to enable managers to compare projections with current balances, both in summary and in detail.

The system's growth has been modest but continual. There have been major changes in the input data due to changes in accounting systems, and major technological changes to keep pace with the organisation's computing strategy. Changes in the organisation's accounting policies have affected both input data and output reports.

The FRS is relevant because it has a long history of continual evolution and there have been recurrent difficulties in implementing the changes that were required. The current version of the FRS is the latest descendent in a family of information systems that dates back to 1984. Over this period the core concepts of these systems have remained intact but many technical and business details have changed and will continue to. FRS has, at different times, been written in various SQL dialects, glued together with various scripting languages such as REXX (when it ran on an IBM mainframe) and Perl (when it ran on other platforms). On average, a team of 2 full-time staff was needed to develop and maintain FRS. The major COTS component of FRS throughout its lifetime has always been its RDBMS.

There are three main sources of evolution in the FRS, arranged here in descending order of impact:

Source data : There have been several major changes in both the laboratory's choice of accounting system (which supplies the input data) and in accounting practices.

IT environment : An important success factor for the FRS is its compatibility with users' normal desktop computing environment, which has changed from a mainframe with text terminals to a network of graphical PCs and workstations. On the server side of the system there are also recurrent issues about retaining compatibility with the laboratory's evolving IT architecture.

Users' business process : As the laboratory's customers become more diverse there is an increasing need for flexibility and customisation in the way that project reports are compiled and presented.

These three sources of change are characteristic of MISs although clearly their relative importance might be different in other systems.

3 Primary Concepts

This section discusses the terms *evolution*, *information system* and *software evolvability.*

3.1 Software Evolution

The word *evolution* is commonly used in at least three senses in relation to software:

1. The term *evolution* is sometimes used to refer to the changes in a software product over its lifetime. This sense of evolution is closely associated with the work of Lehman *et al.*; [5] [6]. One of their fundamental conclusions is that software evolution is a complex feedback-driven process of change that affects a broad class of software products, and some of its effects are emergent properties (i.e. they can occur independently of the intentions of the system's stakeholders, as discussed in section 4.3).

2. The term *evolve* is sometimes used as a transitive verb to refer to the act of intentionally changing software [7]. For example, "we evolved the software to meet the users' needs".

3. The term *evolutionary* is sometimes used to characterise software that has been designed to automatically adapt to changing circumstances (such as autonomic software). The principal applications of evolutionary computation and genetic programming [8, 9] have been in optimisation and searching.

This paper is concerned with *evolution* in the first of these senses. That is, we view evolution as what happens to software over time, as seen from outside the technical process; it is the result of maintenance and other changes [10]. By contrast, the second sense is concerned with the people who do the work on the system actually changing the software to meet the users' changing needs (i.e. the acts of analysing, designing, programming, etc.).

3.2 Information System

In order to understand the role that feedback plays in software evolution, a holistic, systemic approach is needed. In addition to code, systems include:

- specifications and designs
- technical and managerial processes
- business processes

- viewpoints of the system's stakeholders

In other words, an information system is characterised by interactions between software, people and organisations [11]. Some authors also use the term *information system* in a broader sense that need not necessarily include software. Consequently, a more precise term would be *software-dependent information system*, which draws attention to the symbiotic effects that emerge when software is embedded in business processes. However, for brevity we use the less cumbersome term *information system.*

3.3 Information System Evolvability

Software varies in its capability for evolution; this quality is its *evolvability.* It may be a result of internal characteristics, or the balance between external pressures for change and stability, or a combination of both. For software products, there is a large overlap between the concept of evolvability and more general concepts of software quality, such as maintainability. The ISO 9126 standard [12] identifies five sub-characteristics of maintainability (analysability, changeability, stability, testability and compliance) These are relevant to evolvability, as are some of the metrics in the standard [13].

For information systems in the holistic sense, there is little existing work on identifying the constituent factors that influence evolvability. From theoretical work and case studies [13, 14] that we have done we suggest an extension to the definition of evolvability such that it can be refined into the factors listed below.

3.3.1 Adaptability

This is essentially the same quality as maintainability in ISO 9126 but scaled up from the product focus to encompass the processes of both software maintenance and information system evolution.

3.3.2 Responsiveness to feedback

In systems that have multiple levels of feedback counter-intuitive behaviour often occurs [15]. In other words, the system may exhibit behaviour that cannot be inferred from its constituent parts. A common cause of this is when feedback loops with different time delays contribute to the level of some system variable. This kind of *emergent behaviour* may reduce the system's evolvability because it does not respond to events in the ways that participants expect. Even in simple systems, feedback chains that are either very long or that include major delays can also have counter-intuitive effects. Emergent behaviour may result in unpredictable changes being necessary and the system structure deteriorates as these changes are made across the grain of the system architecture, reducing evolvability (and increasing the risk of a change having unexpected side-effects) as the elegance of the system structure worsens.

3.3.3 Compliance with constraints

Maintenance changes must not violate requirements or other constraints on a system. The main driver for compliance is the extent to which concerns have been separated. Concerns that can be expected to evolve at different speeds should be separated by clearly defined interfaces that remain unchanged.

This requires going beyond the immediate requirements and trying to anticipate which of its constraints are inherent and which are time-dependent. For example, most business information systems are likely to achieve better evolvability if there is a clear separation of concerns between unchanging domain concepts, volatile business rules and the details of data manipulations.

4 Towards an Improvement Process for Evolvability

The improvement of information system evolvability is essentially about finding sustainable resolutions of the conflicting forces that encourage either change or stability in the system.

4.1 Forces Affecting Evolvability

The forces that affect evolvability are both technical and organisational, and may be internal or external to a system. The following broad categories can be distinguished:

- Forces arising from the actions and aspirations of the system's stakeholders.
- Forces that are emergent properties of the evolution process.
- Exogenous forces, which can be particularly unpredictable in timing and effect.

Part of the process of discovering a balance between these forces involves understanding the characteristics of the various feedback paths between the different stakeholders. It is useful to assess the relative importance of different paths and to locate any significant delays or other timing properties. This will often reveal actual or potential conflicting forces at several levels of detail and in different domains.

Examples of tensions between stakeholders during maintenance of the FRS are given below.

- The Finance Department wanted to restrict the system's outputs to a small number of precisely defined reports (to minimise the problems of misinterpretation). However, some users were requesting more flexibility to meet the management needs of particular projects.

- The system's customer wanted continuity of staffing in the maintenance team (to maintain high levels of productivity). However, the professional development needs of staff and the demands from other projects put pressure on the project manager to divert development resources into re-engineering the system so that some maintenance tasks could be shared amongst a wider pool of staff.

- The maintenance team experienced pressure to short-cut design decisions due to urgent deadlines. For example, a decision was made to replace an unsatisfactory COTS report generator with a locally developed module with fewer proprietary dependencies. However, in order to meet delivery deadlines this module was released prematurely; some aspects of the report structure had been implemented without flexibility, increasing the cost of changes such as user customisation of reports.

If the stakeholders have a good, shared understanding of the system's dynamics, they will be more able to anticipate evolution issues before they become crises. However, it is also characteristic of information systems that unpredictable conflicts can occur from unexpected sources. Consequently, *agile evolution* requires information systems to have flexibility in both their architecture and their use of resources.

These kinds of tensions often result in organisational responses such as committee meetings, arbitration and other conflict-resolution procedures. However, such mechanisms may be less effective in resolving (rather than merely containing) tensions if they are not based on an understanding of the feedback dynamics of the evolving system or if the participants lack a shared language for describing the system.

The issue of feedback models is discussed in section 4.2.

4.2 Feedback Models and the Software Maintenance Process

Responsiveness to feedback is an important part of evolvability. This implies the existence of effective feedback channels between the system's stakeholders but many existing development and maintenance methodologies fail to deal with this effectively. They tend to concentrate attention on the feedback loops between stages of technical development, such as coding and testing, that are internal to the development team. When other stakeholders are considered, it is often assumed that feedback between them is straightforward. However, the most influential feedback loops in long term evolution have been found to be the longest, outermost ones, in terms of the number of identifiable steps in the feedback loop and its normal cycle time [16].

The FRS study confirms this. Concepts from system dynamics [15] were used to model the business process for maintaining this software product. Based on interviews with stakeholders, an influence diagram [17] was built that shows the processes and decisions that are involved in transforming resources (primarily staff time) and change requests into information system services and stakeholders' satisfaction. This diagram is complex (it contains more than 20 cause-and-effect loops) and some of its variables are specific to the FRS. However, it is possible to abstract from it some a more generic, qualitative model of the software maintenance business process which is shown in Figure 1.

4.2.1 Schematic model

Figure 1 shows a schematic model which was constructed using the Vensim[1] modelling tool. This can be interpreted as a generic description of software maintenance as a business process. The model is probably more valuable to business managers and process designers than to developers and maintainers. Users actions may well result in pressure in Opportunities for IS Improvements; emergent properties of process arise from the structure of the model itself and the inter-relationships of its variables, so are distributed over it as a whole. Exogenous forces are shown explicitly as an input in the diagram.

[1]See http://www.vensim.com/

Figure 1 shows the variables and cause-and-effect loops that are fundamental to the maintenance process. Because this model is very abstract, with some variables summarising complex phenomena, not all the influence arcs can be labelled with a simple polarity.

The *QoS gap* variable represents the size of the gap between stakeholders' perception of the current quality of the information system and the desired quality. The gap tends to increase with the quantity of *Outstanding change requests*; after some delay, the *Maintenance work done* reduces the QoS gap both directly (by improving the service) and indirectly (by shortening the queue of change requests).

Variations in the QoS gap cause a variety of *Stakeholder responses*, which are often complex and may be conflicting. For example, an increase in the gap might stimulate the maintenance team to bid for additional resources but the customer may conclude that the system was becoming unmaintainable.

The stakeholders' responses influence the *Opportunities for Information System improvement*. So, for example, if users are delighted with the system, they will be more likely to see opportunities to extend their use of it. Some of these opportunities will be converted into change requests, effectively raising the desired quality of the system. Other change requests will be generated by the impact of *Exogenous changes*.

The overall behaviour of the model depends upon the relative strength of the different causal influences and the length of any significant delays. Even in a comparatively simple model such as figure 1, this can be difficult to assess but simulation tools, such as Vensim, can be used to explore the effects of different parameters.

4.2.2 Intermediate model

Figure 2 shows an intermediate model of the software maintenance process that is less complex than a full model but more realistic than figure 1. We have found this to be a useful model for analysis. System dynamics models do not need to be fully quantified to produce results which stakeholders can use to understand evolution [18].

This model illustrates a number of features that are likely to be found in the evolution of similar information systems.

- Many of the feedback loops in other process models such as [19] have been subsumed into the causal influences between the *Domain expert's requirements*, *Work to do* and *Work done* variables, as evidence suggests that organisational forms of feedback are the most important.
- Most of the feedback loops that are important for explaining an evolving system's behaviour are not shown in process models such as the waterfall and spiral models, or in more recent models such as Extreme Programming [20].
- The external variables *Technical environment changes*, *Collaborating systems' changes* and *New user requirements* have an immediate effect of reducing the level of *Quality of service.* They also stimulate the processes for getting maintenance work done but there is an unavoidable delay, indicated by the bar on the arc from *Work to do* to *Work done*, before the level of *Quality of service* is restored. This kind of emergent behaviour has to be taken into account when, for example, a decrease in *User satisfaction* needs to be explained.
- The users' satisfaction and requirements influence the *Work done* variable only indirectly. The diagram shows that many other variables also influence it, for example, by revising, delaying or prioritising the requests for change.

4.3 Kinds of feedback

Several kinds of feedback are important to the evolvability of information systems. Some of these feedback loops are internal to the software maintenance team.

Architectural review : Reviewing past changes may uncover previous assumptions that have been broken by unexpected changes (and may therefore indicate a need for greater flexibility in the system architecture).

Design review : Reviewing maintenance activity to abstract implicit design patterns that have been revealed by unexpected changes. This builds on established mechanisms such as code inspections [21].

These forms of feedback can be thought of as meta-maintenance processes that internally review the technical quality of the team's work. As with all reviews, the developers should not be the only reviewers; this is a very effective way to reveal assumptions and undocumented knowledge. It obliges the team to explain the rationale of decisions and will often identify areas where code and design have drifted apart [21, 22].

Other kinds of feedback occur between groups of stakeholders; they provide the content for some of the causal influence arcs shown in figure 2 (for example, *Evolving usage* is related to *Customer satisfaction*).

Intra-subsystem : Developing a shared understanding of the semantics of subsystem/component interfaces so that future change impact assessments will become more accurate.

Evolving usage : Reviewing the co-evolution of the business process and information system to identify emerging usage patterns, expectations and needs.

These kinds of feedback are becoming increasingly important to the continuing effectiveness of many information systems. In particular, where the task of transferring information between loosely coupled business processes has been automated through information systems, there has to be a feedback process that reviews whether interfaces have changed significantly.

For example, consider a MIS, such as the FRS, that extracts data from various sources that are not under its direct control and generates reports. The systems share a data interface that has syntax, grammar, and semantics. If a data source changes this interface unilaterally at the syntactic or grammatical level, for example by supplying text in a field where a date was expected, then the non-conformance can usually be detected immediately by the data recipient and the problem can be addressed. However, even if the MIS incorporates an unusually rich conceptual model of its domain, it is much harder to automatically detect semantic non-conformance.

Another example from the FRS occurred when a rule in the accounting domain about the calculation of depreciation was changed. This altered the semantics of some data that the FRS imported from the accounting system. The FRS had to modify how it used this information to construct valid reports (even though the format and the physical datatype of the values remained unchanged).

Although some semantic problems can be detected using type and integrity checks, this process cannot be completely automated where an information system is embedded in an evolving real-world business process. In such a situation, the software becomes part of an open-ended domain and consequently its assumptions cannot be exhaustively specified; this is the defining characteristic of Lehman's category of evolving programs (E-type programs [23]). In practice some kinds of semantic problems in the FRS were detected more effectively by domain experts and users than by automatic checks.

In these complex situations, merely informing other stakeholders about changes is often ineffective. There has to be a process of dialogue and feedback using a shared language, so that a stakeholder can understand whether or not a change in a collaborating system or process invalidates current assumptions and if so, what the impact will be and how its effects can be mitigated.

4.4 Implications of Incomplete Feedback Models

Basing maintenance processes on incomplete knowledge of the processes of feedback contributes to several kinds of deterioration in information systems:

- Systems/components may become increasingly irrelevant to users because of divergence between evolving requirements and their implementation.
- Systems/components may become increasingly incomprehensible to their maintainers because of divergence between evolving implementation and specification/design documents that have been incompletely updated.
- The original architecture of the system may become inadequate as requirements evolve. This may put pressure on maintainers to make *ad hoc* changes that work around an immediate problem but often tend to increase the rigidity of the architecture for future changes.
- The interfaces between subsystems/components may become increasingly brittle if the assumptions and guarantees of each interface are unclear. This risk is particularly applicable when COTS components are involved; market forces or other events may cause a manufacturer to change a

product in ways that a customer did not expect and often cannot influence [24, 25]. In extreme cases, the product is withdrawn without warning.

Inability to anticipate changes, to know when they will arise and how long we have before these changes themselves are overtaken by events (all process issues) may contribute to what is experienced by its users as reduced evolvability of the system. All of these issues could have their effects mitigated by a greater understanding of the process of software evolution.

5 Using Patterns to Improve Feedback

5.1 Feedback requires a shared language

A central problem in improving the quality of feedback between system stakeholders is finding a shared framework for describing problems and solutions in information systems. This is a difficult problem to solve because different stakeholders have partially overlapping concerns about the system but may also have strong preferences for how they want their knowledge of the system to be structured and presented.

Many of the languages and models that are currently available express the viewpoint of a particular stakeholder (although this is not always made explicit); less attention has been given to developing languages that would enable different stakeholders to share information about their overlapping concerns. Pattern languages [26] and software services [27] may facilitate feedback, as discussed below.

It seems unlikely that there will be a simple, universal solution to this problem. If a single language had a broad enough spectrum to meet the needs of all the stakeholders in information systems, it would be impossibly cumbersome. Consequently, stakeholders will continue to use languages and modelling tools that meet their specialist needs. Translation between these languages will often be difficult. This notion of distinctive, describable viewpoints for different stakeholders has been recognised by the IEEE/ANSI Standard 1471-2000 [28].

5.2 Example of Languages for Viewpoints

Some of the issues that arise when a single problem is described from multiple viewpoints, can be illustrated by considering a very simple information system pattern. The following example describes the Gateway pattern, that can provide an interface between a local area network (LAN) and a leased-line connection to an ISP. The forces that it balances are the needs for control and security within a LAN, user access to resources within the LAN and via the Internet, and protection from Internet-based intruders. This pattern can be represented in various ways and each representation implies a viewpoint that expresses the concerns of particular stakeholders. Three different representations (a box-and-line diagram, an ADL component and a use case) are used to illustrate this.

5.2.1 Box-and-line diagram

The example in figure 3 is based on the IT architecture of a multi-site company.

This illustrates a viewpoint that could be characterised as an executive summary. It provides an overview of the main components of a system in a form that can be quickly and easily understood by non-experts. However, this view would clearly be insufficiently detailed for the specialists who procure, install, configure and maintain such systems; unfortunately, adding more detail to this sort of diagram would soon make it unreadable.

5.2.2 Architecture Description Language

Technical specialist stakeholders usually require much more detail and precision than a box-and-line diagram provides. One way of achieving this is to use an ADL such as Acme [29, 30]. Conceptually, Acme is intentionally very similar to box-and-line diagrams. Both languages focus on components and their configuration through connectors. More generally, an ADL can provide succinct and precise descriptions of what a system does and how it does it from an IT architect's viewpoint.

An example of an Acme specification for a component type based on the Gateway pattern is shown in figure 4. This illustrates several features that are shared by many ADLs:

- The provided types (Components, Connectors, Ports, Roles) can be combined to specify more complex types.

- Instances of types can be adapted with local extensions.
- The configuration can be specified precisely using the `Attachments` clause.
- The internal structure of objects can be specified recursively to any required depth.
- The mapping between the internal and external interfaces of an object can be specified using the `Bindings` clause.

5.2.3 Use cases

A component-centric ADL cannot meet the needs of all stakeholders. In particular, users of information systems are likely to find that it is difficult to deduce from an ADL description how a system interfaces with business processes. Users may find that use cases express a more helpful viewpoint. An example of a use case for the Gateway pattern is shown in figure 5 [31].

As a system description language, use cases provide several benefits for representing user viewpoints, as they:

- are focused on functionality, rather than configuration
- are structured into scenarios that are related to business processes
- use business, rather than technological, vocabularies
- minimise references to specific technologies

Nevertheless, use cases do not completely cover the concerns of users. Firstly, they usually concentrate on what users can do and only imply the limitations of the system's functionality; secondly, quality of service is usually excluded.

A significant disadvantage of use cases for other stakeholders, e.g. system architects, is that they often treat all constraints uniformly. However, architects need to know whether a constraint is, for example, a volatile business rule, an implication of a project's business case, a technological capability limit or an aspect of a domain's ontology. That is to say, architects need system description languages that can express not only the specification of constraints but also metadata about their provenance. This criticism is not specific to use cases, but is also true for a number of other representations of patterns and specification techniques in general.

5.3 Software as a service

Once an information system has become large and/or complex, stakeholders also need to understand how usage patterns interact with each other, i.e. a behavioural, rather than structural, model of the system. This requires a representation of the information system that is both service-based (i.e. expressed as *Requires/Provides* relationships) and architectural in scope. This viewpoint is implicit in the depiction of the software maintenance process in figure 2.

The building blocks of service-centric models of software-dependent information systems are agents, resources and human actors performing roles [32]. The agents and resources may be purely software or they may provide a software representation of some physical object (e.g. a database of stocks). The main feature that distinguishes agents from resources is that agents have some capability for autonomous behaviour, whereas resources are passive in their relationships with other objects. These relationships, the interfaces between objects, are defined in terms of what each object *provides* to and *requires* from the objects that it collaborates with.

Software services are an abstraction from this model that allows the behaviour of a collection of agents and resources to be encapsulated behind a single interface. Many existing ADLs do not provide built-in support for these behavioural concepts but the W3C initiative in web services[2] may lead in this direction by adding an architectural dimension to fine-grained service-oriented standards such as SOAP.

Bennett *et al.* [27] have proposed an advanced concept of software services as a means of achieving ultra-rapid evolution in information systems. Simpler forms of the same concept are already in everyday use, particularly in e-commerce for providing generic services such as shopping baskets and authenticated payments.

[2] `http://www.w3.org/2002/ws/`

5.4 Patterns and pattern languages

Patterns (in the sense described by Alexander [33]) may be characterised as reusable elegant solutions to recurring problems of conflicting forces. Alexander's ideas provide a conceptual framework for devising resolutions of the forces that are found in software maintenance processes. In the most general case, a pattern that improves software evolvability should provide an efficient and elegant resolution of some forces that encourage and inhibit change in a information system. The pattern must preserve the system's valued characteristics (which should themselves be expressed as patterns of design and usage), whilst allowing desired changes in the state of the system.

A pattern can capture in an explicit and reusable form a good solution to some recurring problem in software evolution. This makes it easier for the system's stakeholders to share their knowledge about problems and solutions. The evolvability of the system should improve if these patterns are used consistently over time to both diagnose and repair whatever problems arise. Alexander's use of this concept can be distinguished from the use popularised by Gamma *et al.* [34] in three important ways:

- patterns are arranged morphologically in a pattern language
- patterns are deployed sequentially in a design-and-repair process
- pattern languages evolve through a process of participatory planning

We suggest that the use of pattern languages has potential benefits over and above the use of patterns. These ideas are explained in the sub-sections below.

5.4.1 Pattern language morphology

Alexander believes that patterns can only be effective when they are arranged morphologically in a pattern language that covers all the relevant scales of detail and abstraction in the system. He provides patterns that cover a range from regional planning to the design of a window alcove. This is a more powerful concept than a catalogue that groups patterns thematically, as in [34]. In Alexander's view, arranging patterns in refinement sequences enables lay people to build up coherent mental models of how the patterns in a collection relate to each other.

In [35] Alexander provides several examples of this process, including a description of how he worked with staff to design a new clinic. The first pattern that they used was *Building Complex*, which determined the total floor area to be built, the number of separate buildings and their relative sizes. The next two patterns, *Main Gateway* and *Main Entrance*, were taken together. The site of the clinic had already been decided; these patterns produced decisions on how people would identify and reach it from neighbouring buildings and the street. These patterns needed to be considered early in the process because they would influence the orientation of the individual buildings. The next pattern in the language was *Circulation Realms*, which determines how a group of buildings will be arranged in relation to each other. It focuses on creating a coherent arrangement in which people will find it easy to understand where they are within the Building Complex and how to navigate to the building they require. This process of gradual refinement continued down to the design of individual rooms. The pattern language contained 42 patterns but only seven were specific to hospital environments, the remaining 35 were reused from Alexander's existing repertoire.

A similar concept of refinement can be found in the SADL [36]. This ADL includes a mechanism for refining descriptions by providing precise mappings between the style vocabularies used at different levels of refinement. For example, an architecture could be described in terms of a pipe-and-filter style at one level and an inter-communicating processes style at another, and the mapping between these can be expressed using SADL.

5.4.2 Morphology of information system patterns

Alexander arranged his patterns along essentially a single dimension from coarse-grained (e.g. Identifiable Neighborhood) to fine-grained (e.g. Alcove). A similar arrangement could be made for a narrow definition of software patterns. However, for information systems, in the holistic sense, it seems more realistic to arrange patterns along two orthogonal dimensions: *coarse-grained* to *fine-grained* and *Business process oriented* to *Technology oriented*. This pattern space is depicted in figure 6, which also shows the relative positions of some broad classes of interesting patterns.

If a pattern language is to fulfil the role of enhancing feedback between the stakeholders of an information system then it will need to include a very broad range of patterns, for example:

Ergonomic design: how to design tools, forms, screen layouts etc. that people can use efficiently.

IT architecture: how to configure IT components, connectors, services etc. to create IT architectural styles.

Program design: how to design program modules that collaborate to carry out computations.

Programming idiom: how to write well-crafted code in a specific programming language.

Software service: how to represent agents and resources in software and configure them to provide services.

Workflow: how to route work items between roles to carry out a business process.

This is not an exhaustive list. In particular, this does not imply that all business processes [37] should be modelled as workflows (note that some parts of figure 6 fall outside the scope of this paper).

The broad scope of information system patterns implies that the upper bound on the number of patterns to be developed could be high; for comparison, Alexander's catalogue [33] contains 253 generic patterns and his detailed examples of building projects typically use around 30-40 [35]. The development costs of patterns can be mitigated in two ways:

- Pattern languages can be developed gradually, using the processes described below.
- Well-designed patterns are easy to reuse and very few should be specific to a single project.

Eventually most of the patterns in pattern languages will be reused from a stock that has accumulated over time. As this stock increases, the task of designing a pattern language will become more concerned with ensuring that the most suitable patterns have been selected and arranged in a coherent way, rather than developing new patterns, in the same way that interest in frameworks [38] grew once catalogues of object-oriented patterns became available.

5.4.3 Examples of information system patterns

Three of the patterns observed in the FRS are described briefly here, and classified using our morphology.

Data Warehouse Over several versions of the FRS, its designers consciously and successfully reused the Data Warehouse pattern (figure 7). This *IT Architecture* pattern achieves a separation of concerns between one-at-a-time update transactions (which are handled by the data source components) and batched set report generation. This simplifies the management of the security, optimisation, dependability etc. characteristics of two different kinds of software service.

Intranet-based MIS Part of the business case for the FRS has been the notion that project managers need to be able to obtain online financial summaries of specific projects from within their usual desktop computing environment (rather than waiting for monthly standardised, centrally produced reports). This approach (an example of a *Software Service* pattern) is now widely used for delivering management information.

Materialised View This is an example of a *Programming Idiom* pattern, in this case for database languages such as SQL. This pattern builds on the concept of using a database view (i.e. a virtual table) to encapsulate a piece of business logic; the view is then materialised as a physical table in the database to improve the performance of queries. In SQL this can be achieved in a database schema by replacing statements of the form

`CREATE VIEW` *view-name* `AS SELECT` *query-spec*

with

`CREATE TABLE` *view-name* `AS SELECT` *query-spec*

This pattern has similarities with the Data Warehouse pattern. Both use additional storage (disk space) that is redundant in logical terms (and has non-zero cost) to separate otherwise conflicting concerns; however, they operate at very different scales, with the Materialised View pattern being a much finer grained pattern than 'Data Warehouse'.

The two patterns are also linked morphologically in an implicit pattern language, because the 'Materialised View' pattern conflicts with the 'Normalised Relations' pattern that database designers use to minimise update dependencies between data records. Consequently, 'Materialised View' is only a good solution to query performance problems in the context of 'Data Warehouse' or a similar pattern that delegates the updating of individual records to another component or service.

5.4.4 Pattern repair processes

Alexander describes a design-and-repair process in which patterns are deployed sequentially according to their position in the language. He believed that a satisfying, evolvable environment was more likely to emerge from the piecemeal outcome of large numbers of small concurrent projects than from the imposition of a master plan. However, he also recognised the risk that this approach would produce frustrating chaos rather than enjoyable diversity. He therefore retained significant top-down elements in his methodology and adapted them to a predominantly user-driven planning process.

The principal elements of Alexander's planning process [39] are summarised below.

1. The process begins by a community reaching agreement on a pattern language that it will use to solve recurrent design problems.

2. Gap analysis is undertaken to identify how each part of the current environment compares with each relevant pattern. This analysis identifies the "hot spots", i.e. the elements of the environment that are in greatest conflict with the most patterns.

3. Any member of the community can propose any change project and receive professional design assistance to present it for approval.

4. The project can only proceed if it is approved by the Planning Committee, which will take into account both the extent of support for the proposal and how much contribution it makes to closing the identified gaps. Alexander hoped that this perspective would encourage the committee to quickly approve many cheap but effective projects, rather than a small number of mega-projects.

5. Similarly, anyone can propose modifications to the pattern language to improve its capability for identifying problems and proposing elegant solutions. Changes are accepted if they achieve consensus.

The above techniques can be used in the software evolution process to reduce the delays and misunderstandings often found during feedback, particularly in the case of large-scale co-operative system's design.

5.4.5 Participatory planning for information systems

Some of the concepts of pattern languages and participatory planning described here may seem to introduce additional risks into the software maintenance process. Many, perhaps most, business change projects are organised using conventional, hierarchical management structures and it may seem implausible that a complex information system could be built and maintained by any other style of organisation. Furthermore, some attempts to use alternative approaches to system design, such as Alexander's, have produced disappointing results [40]); further research is needed to identify the critical success factors for using pattern languages to maintain information systems.

However, in favour of patterns, note that two of the most successful IT projects of recent decades have consciously used consensual, decentralised approaches and have become part of the infrastructure many information systems rely on. The organisation of the IETF [41] has many of the participatory elements described in the previous section. The architecture of the Internet is also a good example of a pattern language in practice [42]. A relatively small number of patterns is sufficient to describe how the Internet works. These patterns relate to each other coherently but also retain some independence. So, for example, email users can send messages and system administrators can add hosts without interfering with each other's activities, or needing to know the details of the underlying infrastructure, or needing to refer to a central authority.

A similar (but different) example can be found in the WWW. Its leading body, the W3C, has a more formal structure than the IETF but it has similar well-developed mechanisms for seeking rapid feedback on its proposals. Its design principles of interoperability, evolution and decentralisation[3] are consistent with a pattern language approach. Unlike the Internet, which could be said to be a network rather than an information system, the WWW provides information services and infrastructure that are critical to

[3] http://www.w3.org/Consortium/.

the success of many organisations. This suggests that basing the design of a software maintenance process on concepts of feedback and pattern languages can be successful in the right circumstances. Patterns may succeed because they make purpose and design explicit by the use of recurring abstractions and hence improve communication.

5.5 Feedback and Rates of Change in Information Systems

The characteristics of feedback in software maintenance processes have implications for the pace of change in information systems. The techniques that are advocated here for achieving better evolvability in information systems (e.g. improved quality of feedback, user-oriented pattern languages and service-oriented software architectures) imply a slower (but more stable) rate of change. This is because the system maintenance process is re-engineered to continuously respect the kinds of feedback loops identified here as critical for evolvability. Conversely, when ultra-rapid change is required, it may be more cost-effective to build light-weight, disposable systems. In this situation, the operation of feedback loops may be constrained to intensive bursts of consultation and decision-making at defined points in the system's life cycle.

Currently available technologies allow different kinds of software architecture and components to provide equivalent functionality. This means that there is a decreasing need for technological choices to dominate the design of the information system and more opportunities to consider business-oriented concepts, such as feedback models. The decision on what kind of architecture and feedback model to adopt for a system should be based on a business case covering its expected life, the relative priorities of non-functional qualities (e.g. correctness, security, agility), risk analysis of alternative development and maintenance approaches, and thus its predicted total cost of ownership.

6 Conclusions and Future Directions

The process of maintaining the software of an information system can be very complex. The concept of feedback provides powerful insights into this complexity. In particular, it draws attention to the communication issues that can arise when stakeholders with different viewpoints need to reach a shared understanding of the evolution of an information system. This is clearly important when the strategic evolution of an information system is being considered but it is also relevant at more mundane levels, such as prioritising bug fixes.

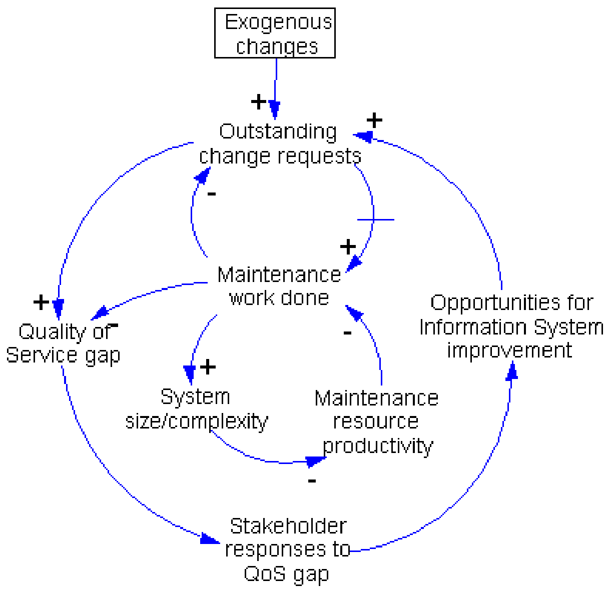
Although some of the issues discussed here might be considered to be outside a narrow definition of software engineering, in practice software engineers often become involved in trying to resolve them. One of the implications of information systems is that software engineers often have to consider the interactions between their specialised domain and the non-software parts of the system.

A significant contribution that software engineers can make is through improving the quality of feedback between stakeholders. The concept of patterns has already proved to be useful and popular within the software engineering community as a means of capturing and sharing design knowledge. This paper has shown how this concept can be broadened to encompass the whole scope of information systems evolution. This involves re-discovering some of Alexander's original concepts, notably pattern languages, considering carefully how these can be used and adapting them to the information systems domain. A pattern language approach can be consistent with participatory styles of organisation that have produced software engineering successes. A taxonomy of patterns and pattern languages constructed around the morphology of pattern languages suggested here could facilitate this. Guidelines as to what kind of architecture and feedback model to adopt would also facilitate systems' developement.

Finally, one of the benefits of focusing on the quality of feedback may well be to reduce the need for rapid evolution of information systems. If software engineers and other stakeholders share a richer understanding of the dynamics of an information system and its co-evolution with business processes, it may be easier to anticipate the software adaptations that will be required and achieve a closer alignment between software release schedules and business needs.

7 Acknowledgements

The authors would like to thank the University of Reading for supporting this research, and the anonymous referees for providing very helpful comments.


Exogenous changes
Outstanding change requests
Maintenance work done
Quality of Service gap
System size/complexity
Maintenance resource productivity
Opportunities for Information System improvement
Stakeholder responses to QoS gap

Key for figures 1 and 2

$\boxed{\text{boxed node}}$	: External variable
unboxed node	: System variable
$\longrightarrow$	: Cause-effect influence
$+\!\!\!\longrightarrow$	: Cause-effect influence with delay
$+\ -$	: Influence polarity

Figure 1: Schematic influence diagram for a generic software maintenance process

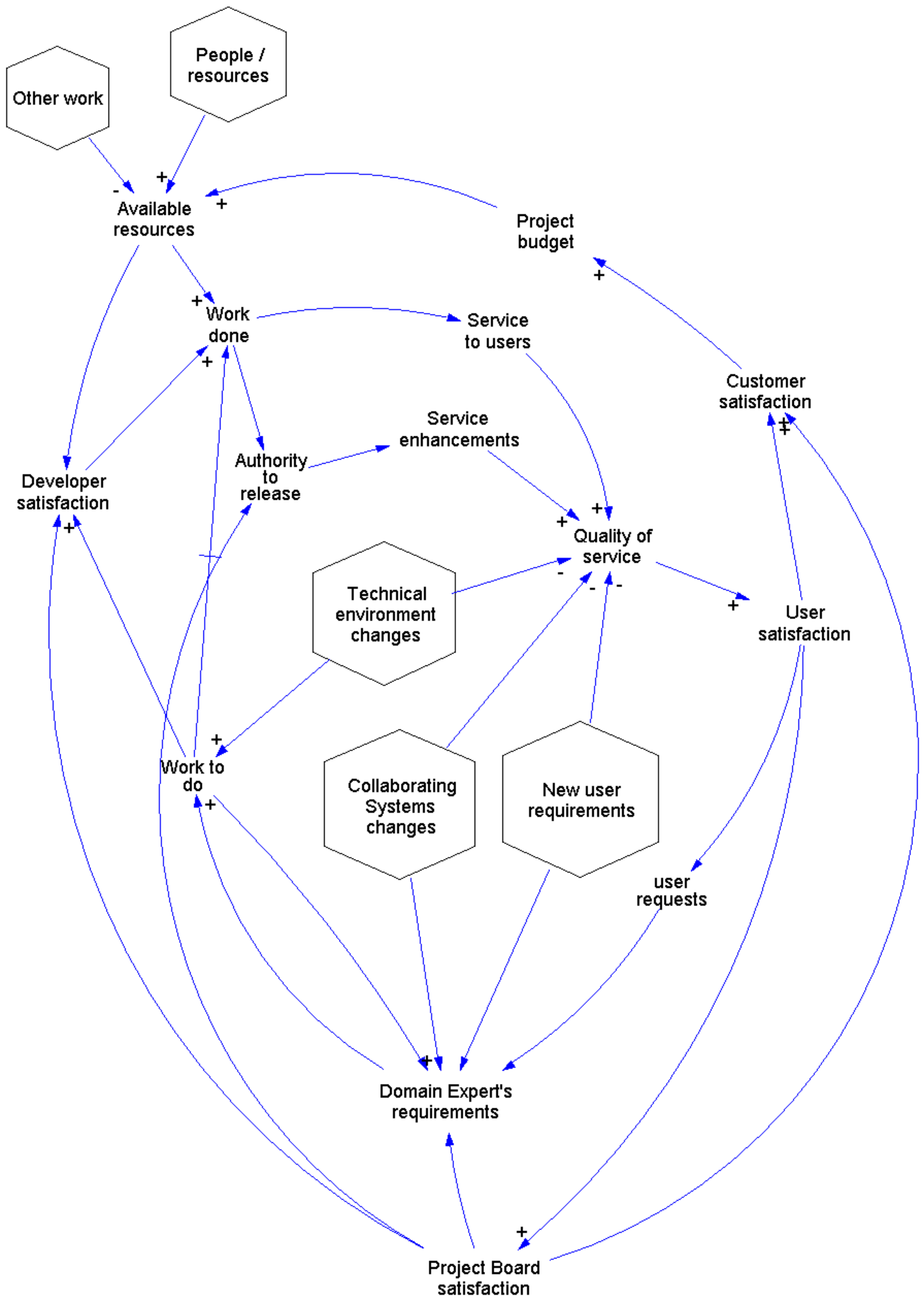


Figure 2: Influence diagram for a generic software maintenance process

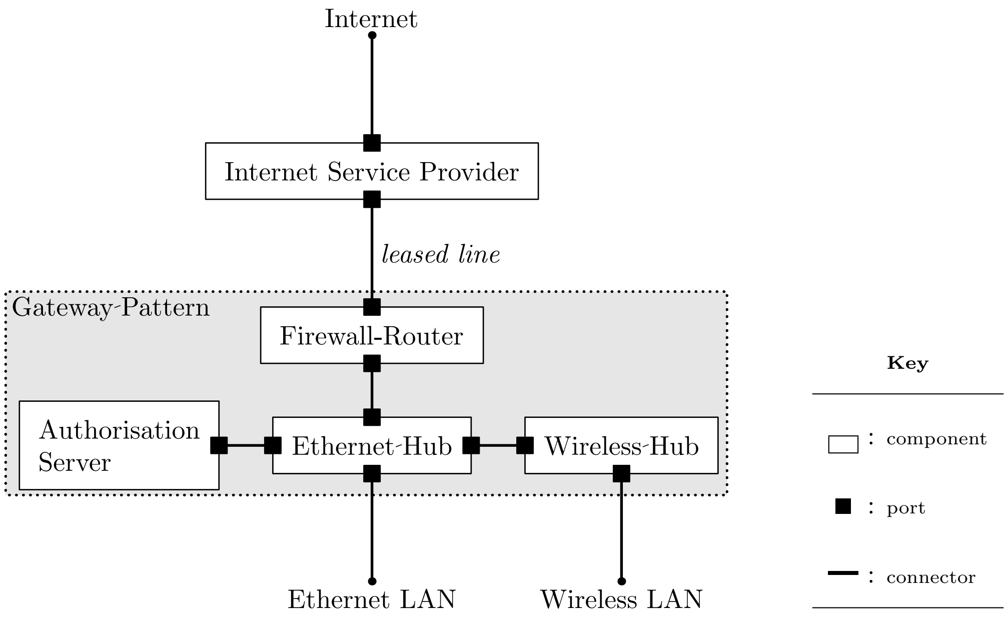


Figure 3: Gateway pattern – box-and-line diagram

```
Component Type Gateway = {
        Port inside : theEthernetPort = new EthernetPort;
        Port outside : theWANport = new WANport;
        Port wireless : theWirelessPort = new WirelessPort;
        Representation {
            System Structure : NetworkArchitecture = {
                Component theFirewallRouter : FirewallRouter = new FirewallRouter;
                Component theEthernetHub : EthernetHub = new EthernetHub;
                Component theWirelessHub : WirelessHub = new WirelessHub;
                    Component theAuthServer : AuthServer = new AuthServer;
                Connector c1 : EthernetConnection = new EthernetConnection
extended with {
                    Role r1 : EthernetSocket = new EthernetSocket;
                    Role r2 : EthernetSocket = new EthernetSocket;
                };
        /* similar specification for Connectors c2 and c3 omitted */
                Attachments {
                    theWirelessHub.EthernetPort1 to c2.r1;
                    theAuthServer.EthernetPort1 to c3.r1;
                    theEthernetHub.toWireless to c2.r2;
                    theEthernetHub.toRouter to c1.r2;
                    theEthernetHub.toAuth to c3.r2;
                    theFirewallRouter.EthernetPort1 to c1.r1;
                };
            }; /* end system */
            Bindings {
                outside to theFirewallRouter.outside;
                inside to theEthernetHub.EthernetPort1;
                wireless to theWirelessHub.WirelessPort1;
            };
        };
    };
```

Use case:	**Access Network Resources**
Actors:	User (initiator), ISP
Purpose:	Obtain access to local and remote network resources

Typical Course of Events :

Actor Action	**System Response**
1. The User uses a workstation connected to either the wired or the wireless LAN and attempts to log in.	2. If authentication of the User succeeds, open a session for the User.
3. The User requests one or more services, either sequentially or in parallel, from the following kinds of resource:	
(a) resources within the LAN	4(a). Add the resource to the User's session.
(b) Internet resources	4(b). Forward the request to the ISP.
5. The User logs out.	6. Close the User's session and release any resources attached to it.

Alternative courses:

Action 2: Authentication fails ⇒ indicate error.

Action 4(a): This User is not authorised to use this local resource ⇒ indicate error.

Action 4(b): Access to this Internet resource is barred ⇒ indicate error.

Figure 5: Example of a use case

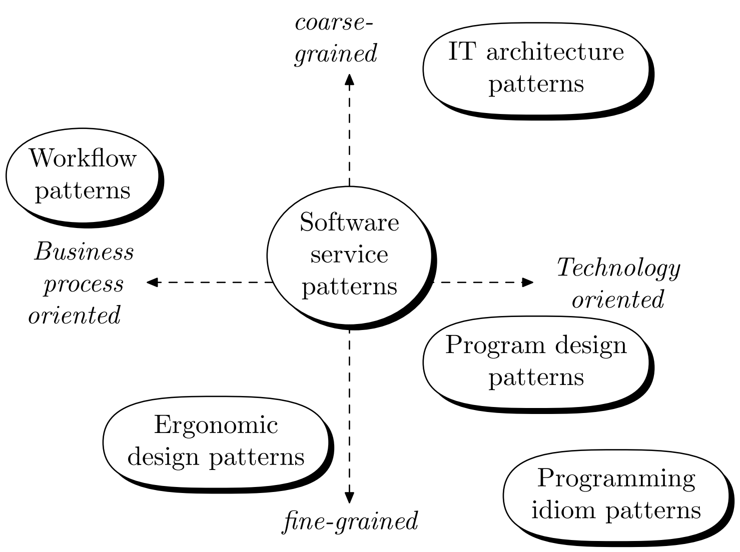


Figure 6: Morphological map of pattern languages for information systems

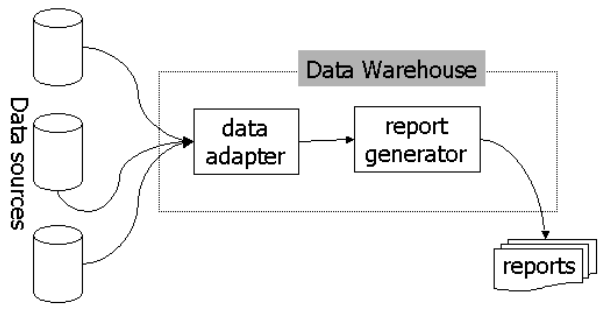


Figure 7: Data warehouse pattern

List of Figures

References

[1] Ward J, Griffiths P. Strategic Planning for Information Systems. John Wiley (Wiley Information Systems series): Chichester, UK; 1996.

[2] Warboys BC, Greenwood RM, Kawalek P. Modelling the co-evolution of business processes and IT systems. In: Henderson P, editor. Systems Engineering for Business Process Change: Collected Papers from the EPSRC Research Programme. Springer-Verlag; 2000. p. 10–23.

[3] Chatters BW, Lehman MM, Ramil JF, Wernick P. Modelling A Software Evolution Process. Software Process: Improvement and Practice. 2000;5:91–192.

[4] Wernick P, Hall T. Simulating Global Software Evolution Processes by Combining Simple Models: An Initial Study. Software Process: Improvement and Practice. 2002;7:113–126.

[5] Lehman MM. Programs, life cycles, and laws of software evolution. Proceedings of the IEEE. 1980;68(9):1060–1076.

[6] Lehman MM, Ramil JF, et al. Metrics and laws of software evolution — the nineties view. In: Proceedings of the 4th International Symposium On Software Metrics (Metrics 97). Albuquerque, New Mexico: IEEE Computer Society; 1997. p. 20–32.

[7] Lehman MM, Ramil JF, Kahen G. Evolution as a noun and evolution as a verb. In: Proceedings of SOCE 2000. London, UK; 2000. .

[8] Banzhaf W, Nordin P, et al, editors. Genetic Programming: An Introduction. Morgan Kaufmann: San Francisco CA; 1998.

[9] Koza JR. Genetic Programming: On the Programming of Computers By Means of Natural Selection. MIT Press: Cambridge MA; 1992.

[10] Madhavji N, Ramil JF, Perry D, editors. Software Evolution. John Wiley: Chichester, UK; 2006.

[11] Avison D, Fitzgerald G, Wood-Harper T. The discipline of information systems: the interdisciplinary thing. Systemist. 1994;16(1):62–69.

[12] Organisation IS. Information Technology — Software Product Evaluation — Quality Characteristics and Guidelines for Their Use, 1st edn. ISO/IEC 9126; 1991.

[13] Cook SC, Ji H, Harrison R. Dynamic and static views of software evolution. In: Proceedings of the IEEE International Conference On Software Maintenance (ICSM 2001). Florence, Italy: IEEE Computer Society Press; 2001. p. 592–601.

[14] Mens T, Hassan G. 4th Workshop on Object-Oriented Architectural Evolution. In: Frohner, editor. Object-Oriented Technology ECOOP 2001 Workshop Reader: Proceedings of the ECOOP 2001 Workshops, Panel, and Posters. Budapest, Hungary: Springer Verlag; 2002. p. 150–164. Lecture Notes in Computer Science series, 2323.

[15] Forrester JW. Industrial Dynamics. MIT Press and John Wiley; 1961.

[16] Kahen G, Lehman MM, et al. Dynamic modelling in the investigation of policies for E-type software evolution. In: Proceedings of ProSim 2000. London, UK; 2000. .

[17] Wolstenholme EF. System Enquiry: A System Dynamics Approach. John Wiley: Chichester, UK; 1990.

[18] Coyle G. Qualitative and quantitative modelling in system dynamics: some research questions. System Dynamics Review. 2000;16(3):225–244.

[19] Royce WW. Managing the development of large software systems: concepts and techniques. In: WESCON Technical Papers, 14(A/1); 1970. p. 1–9.

[20] Beck K. Extreme Programming Explained: Embrace Change. Addison Wesley; 1999.

[21] Fagan ME. Design and code inspections to reduce errors in program development. IBM Systems Journal. 1976;15(3):182–211.

[22] Fagan ME. Advances in software inspections. IEEE Transactions on Software Engineering. 1986;12(7):744–751.

[23] Cook SC, Harrison R, Lehman M, Wernick P. Evolution in software systems: foundations of the SPE classification system. Journal of Software mainteance and Evolution. 2006;18(1):1–35.

[24] Lehman MM, Ramil JF. Software evolution in the age of component based software engineering. IEE Proceedings on Software Engineering. 2000;147(6):249–255.

[25] Lehman MM, Ramil JF. EPiCS: Evolution Phenomenology in Component-intensive Software. In: Proceedings of WESS 2001. Florence, Italy; 2001. .

[26] Harrison N, Foote B, Rohnert H, editors. Pattern Languages of Program Design 4. Addison-Wesley; 2000.

[27] Bennett K, Munro M, et al. An architectural model for service-based software with ultra rapid evolution. In: Proceedings of the IEEE International Conference On Software Maintenance (ICSM 2001). Florence, Italy: IEEE Computer Society; 2001. p. 292–300.

[28] Society IC. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE-Std-1471-2000. IEEE: New York; 2000.

[29] Garlan D, Monroe RT, Wile D. ACME: an architecture description interchange language. In: Proceedings of CASCON'97. Toronto, Canada; 1997. p. 169–183.

[30] Garlan D, Monroe RT, Wile D. ACME: architectural description of component-based systems. In: T LG, M S, editors. Foundations of Component-Based Systems. Cambridge University Press; 2000. p. 47–68.

[31] Larman C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design. Prentice Hall; 1998.

[32] Jennings NR. An agent-based approach for building complex software systems. Communications of the ACM. 2001;44(4):35–41.

[33] Alexander C, Ishikawa S, Silverstein M. A Pattern Language: Towns, Buildings, Construction. Oxford University Press; 1977.

[34] Gamma E, Helm R, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley; 1995.

[35] Alexander C. The Timeless Way of Building. Oxford University Press; 1979.

[36] Moriconi M, Riemenschneider RA. Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. SRI International; 1997. SRI-CSL-97–01.

[37] Ould MA. Business Processes: Modelling and Analysis for Re-Engineering and Improvement. John Wiley: Chichester, UK; 1995.

[38] Johnson R, Foote B. Designing reusable classes. JOOP. 1988;p. 22–35.

[39] Alexander C, Silverstein M, et al. The Oregon Experiment. Oxford University Press: New York; 1975.

[40] Gabriel RP. The failure of pattern languages. In: Rising L, editor. The Patterns Handbook: Techniques, Strategies, and Applications, (SIGS Reference Library series, 13. Cambridge University Press; 1998. p. 333–344.

[41] Bradner S. The Internet Engineering Task Force. In: DiBona C, Ockman S, M S, editors. Open Sources: Voices from the Open Source Revolution. O'Reilly; 1999. p. 47–52.

[42] Dyson P, Longshaw A. Patterns for Internet Architecture. In: Proceedings of EuroPLoP 2002. Illinois, USA; 2002. .