Architectural Reflection for Software Evolution

Stephen Rank

Department of Computing and Informatics,
University of Lincoln.
srank@lincoln.ac.uk

Abstract. Software evolution is expensive. Lehman identifies several problems associated with it: Continuous adaptation, increasing complexity, continuing growth, and declining quality. This paper proposes that a reflective software engineering environment will address these problems by employing languages and techniques from the software architecture community.

Creating a software system will involve manipulating a collection of views, including low-level code views and high-level architectural views which will be tied together using reflection. This coupling will allow the development environment to automatically identify inconsistencies between the views, and support software engineers in managing architectures during evolution.

This paper proposes a research programme which will result in a software engineering environment which addresses problems of software evolution and the maintenance of consistency between architectural views of a software system.

1 Introduction

Software evolution is expensive, with costs variously estimated as constituting 50–70% of the total lifecycle costs of software [1]. This paper proposes a software engineering environment which will enable software engineers to produce software that is easier and cheaper to evolve as its requirements change. The environment will use reflection to maintain a consistent set of views of a software system at several levels of abstraction, up to and including the architectural level.

The proposed environment will ensure that software engineers always have up-to-date knowledge of the architecture and design of a software system, enabling them to make informed decisions during its evolution, and to avoid some of the problems associated with degradation of structure during evolution. Architectural constraints, both within and across views, will be automatically monitored, using design critics [15] to inform software engineers of inconsistencies and potential problems with a software system.

2 Problems

There are several problems associated with the evolution of software. This paper proposes a software engineering environment which is designed to address five of these problems, selected from Lehman's laws of software evolution [2]:

Continuous Adaptation "E-type systems[1] must be continually adapted else they become progressively less satisfactory"

Increasing Complexity "As an E-type system evolves its complexity increases unless work is done to maintain or reduce it"

Continuing Growth "The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime"

Declining Quality "The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes."

This paper takes the position that loss of knowledge about architectural structure is a cause of several of these problems: knowledge is dispersed throughout the documentation, code, configuration management tools, build tools, and often only maintained tacitly by developers. Section 3 identifies the architectural principles used in this paper, and section 4 introduces reflection as used in this work.

Because knowledge of and constraints upon software architecture are not always made explicit by developers, these structures tend to degrade as software evolves. As the structure is not known, there are extra comprehension costs in maintenance. Worse, if the documentation is wrong/out of date, work can proceed on incorrect assumptions about the current and desired structure and content of the software system.

This paper proposes that a software engineering environment which makes use of reflection can be used to:

- Automatically maintain the consistency of our documentation with respect to the various different "views" of a system;
- Make the architecture of a system visible and manipulable at run-time;
- Allow intervention at a higher level than the source;
- Manage and maintain safety properties and other desirable features;
- Allow automatic analysis of architectural properties of software systems.

A programme of research, leading to a software system which can support the above activities is proposed in section 5, and further work is identified in section 6.

[1] Software systems that are *E*mbedded in a real-world environment, as contrasted to P-type software which solves approximations of real-world problems with well-defined input, such as weather forecasting, and S-type software, which is formally and completely specified as a function from its input to its output.

3 Software Architecture

Software architecture is the study of the structure of software systems, including inter-component relationships [3, 4]. One of the first definitions of architecture, still widely-used, is “Software Architecture = {Elements, Form, Rationale}” [5]. Interactions are considered first-class entities [6]. Architectural description languages model both components and connectors.

There are many important structures in a software system. Kruchten suggested a “4+1” view model, in which four structural views are bound together with a fifth “scenarios” view [7]:

Logical The object-oriented decomposition, commonly modelled using class diagrams.
Process Models “non-functional” requirements (*eg*, performance, availability). Often modelled using collaboration or interaction diagrams.
Development Modular decomposition, modelled with component diagrams.
Physical The mapping of the software to the hardware; usually modelled with deployment diagrams.
Scenarios The “+1” view; a set of scenarios used to motivate the development and assist in the verification of the system.

Documentation of architecture is a key issue: “The essence of the activity is writing down—and keeping current—the results of architectural decisions” [8]. Maintaining a correct and up-to-date architectural model assists with the problems identified in Lehman’s laws of Continuing Growth and Declining Quality (described in section 2). Additionally, knowledge and control of the architecture of a software system is required to support the multi-level feedback nature of evolutionary processes.

4 Reflection, Architecture, and Evolution

Software reflection has been used in many ways to support software evolution [9–12]. This work has usually focused on enabling evolution to take place on a particular system at a design level, rather than on using reflection to ensure consistency between multiple levels of view. It is possible to use reflective operations to map simple architectural changes down to the implementation [9, 13], allowing modifications to the system to be carried out a relatively high level.

Information obtained by reflection tends to be limited to structural and low-level behavioural information. There is a lot of architectural information generated by software engineers during the production and modification of a system. Kruchten’s views [7] can be considered to be bound together by reflection:

Logical Available as a result of reflection or introspection on source code (or intermediate representations such as Java bytecode), as it most closely corresponds to the source of the system.
Process Partially available from reflection and static analysis of code, but often only made explicit in requirements or specification documents.

Development Partially available from reflection or even SCM data.
Physical Partially related to the process view, and available in some cases from deployment descriptors.
Scenarios Not available from the code (except in very specialised cases).

Reflection can be used to bind these views together, ensuring the correspondences between them are maintained, and allowing changes made in one view to be automatically reflected in the others.

Reflective modelling of the *architecture* of software systems can support run-time software evolution. We can support multiple views in ways that make them consistent.

In order to accomplish the goal of knowing the architecture of a system at run-time, the following are required [14]:

Monitoring The ability to to see the architecture at run-time
Interpretation Making sense of the data from monitoring, looking for 'problems' (in any sense of the word)
Resolution Fixing problems: manually, semi-automatically, or automatically. Determining where problems are and what to do about them.
Adaptation The ability to make changes to the architecture of the system.

5 Proposal

This section proposes a software development environment which, using reflection, automatically maintains consistency between a collection of views of a software system. A programme of research and development leading to such an environment is proposed, and potential benefits are discussed.

5.1 A New Kind of Development Environment

It is proposed that a software development and analysis environment will be created. This environment will use reflection to support architectural (and other) approaches to software evolution. Reflection will be used to ensure that different views of the system are synchronised (and to enable highlighting of incompatibilities between these views).

Figure 1 shows the "Lingua Franca" approach: one single representation, with several views (a code view, a use-case view, an architectural view, *etc*) available by applying different 'lenses' to the central representation of the system. By creating appropriate 'lenses', other kinds of views can be created, such as a configuration management or deployment views.

Figure 2 shows the converse case: a specialist language for each view. Boxes in the diagram represent views, while arrows indicate that two views have a correspondence relationship. In this case, there is a specialist language for each kind of view, and correspondences between views are maintained pairwise.

The most obvious problem with the specialist language approach is the proliferation of languages, and the problem of maintaining consistency between the

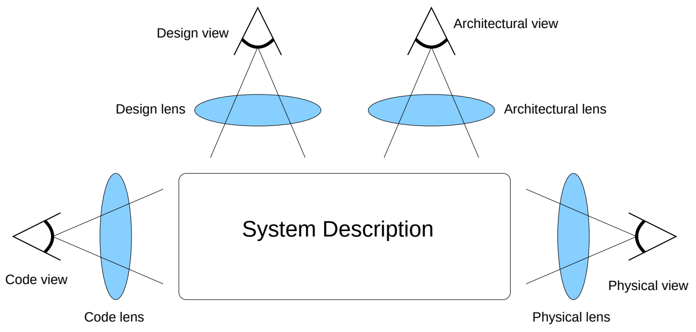

Fig. 1. Lingua Franca Approach

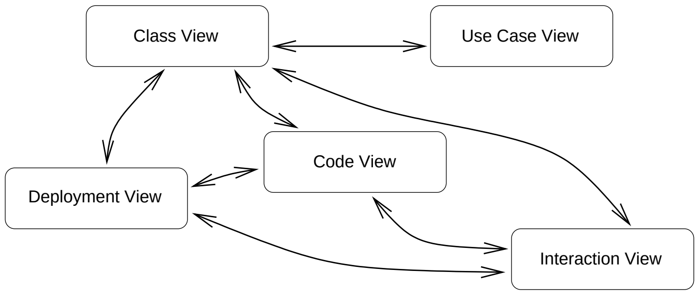

Fig. 2. Specialist Language Approach

views. With the 'lingua franca' approach, these problems are avoided. The most serious problem here is to create a suitable representation that is rich enough to record all information necessary for each potential view.

There are several potential approaches to building such an environment, including the following:

- Adapt current IDE technology (*eg*, Emacs, Eclipse). These are mainly code-level tools, with higher-level modelling considered as an extra facility.
- Adapt current design tools (such as ArgoUML). There is some level of traceability between the design (usually in a language such as UML) and the code, and support for multiple views (such as UML's different diagrams). ArgoUML also includes design critics [15], which go some way towards the analyses that are proposed here.
- Adapt current architectural tools (such as the Software Architect's Assistant). These have an architectural focus, with (currently) few code-level features.
- Start from scratch. This approach leads to an exact match of the system to our requirements, but is slow and inefficient in terms of development effort.

Modelling the architecture explicitly, with proper traceability from higher level constructs to the code will support evolution. Software will be constructed using the appropriate techniques for each kind of entity, and the development environment will manage traceability and identify inconsistencies between different artefacts. The environment will enable software engineers to:

Manage change better because we can interconnections and dependencies are visible;

Discover inconsistencies within and between views automatically in some cases;

Know that our views are correct as they will be automatically extracted from the actual system.

This will reduce costs because: 'comprehension' tasks will produce correct information by definition; some kinds of 'unsafe' changes will be warned against or disallowed; high-level reuse will be supported by high-level knowledge; structure will be made explicit and thus degrading changes will become more obvious.

In order to develop a suitable development environment, the following steps are proposed:

- Develop reflective modelling of each view of a system;
- Allow a software engineer to change each view and have the actual system automatically updated;
- Check and enforce consistency within views;
- Check and enforce consistency between views;
- Support automatic architectural analysis within views;
- Support automatic architectural analysis across views;
- Allow the modelling of patterns and architectural styles in each view;
- Allow the modelling of patterns and styles across views.

To support dynamic replacement of components (*eg*, upgrading a component), it is necessary to support the transfer of state between the old and the new version. This is possible in a semi-automatic fashion [16]. Enforcing consistency requires the use of a suitable logic for describing constraints and evaluating models against them. Support for automatic modelling of patterns and the enforcement of consistency across views will require extensions to this logic to provide suitable mechanism for describing patterns in terms of the architectural features they demand.

6 Further Work

There are several potential problems with the kind of development environment proposed in this paper. In this section, some of them are identified and discussed, and potential means of addressing them are identified.

A system which dynamically supports software evolution must itself be capable of evolving. If it is to remain useful, it must be capable of supported features not considered at the time of its first development [17]. In this example, it may become necessary to develop additional views, or to allow new kinds of constraints between views.

Some, especially in the extreme programming and agile methods communities, take the view that documentation (such as the architectural views proposed here) are superfluous and should be disposed of (for example: "There's this big assumption that diagrams, use cases and the like must be kept in synch with the code, and if they aren't they become completely useless. XP says to write them if you need them and then throw them away."[2]). This is (at least partly) due to the perceived effort involved with maintaining multiple views of the same system, the costs associated with inconsistencies, and the perception that the documentation is of little use anyway. Removing some sources of inconsistency will lessen the desire to discard documentation. Extreme programmers often take the view that the code and test cases together form the documentation, and resist any attempts to create other types of documentation (seen as "the tradeoff to get less functionality and more paper" [18]). On the other hand, there is much research and industrial effort expended in program comprehension and other reverse engineering tasks. This effort would be mitigated by the automatic generation and maintenance of the views proposed in this paper.

In order to carry this work forward, it is essential that a rigorous evaluation technique is devised, and objectively applied to the software and methods developed.

7 Conclusions

Software evolution is a hard problem, which is expensive to tackle. In this paper, a programme of research leading to a software engineering environment has been proposed. This environment will tackle some of the problems of software evolution identified by Lehman [2]. Using software reflection, multiple consistent views of the same system will be maintained, and problems will be identified (in some cases automatically). Reflection provides a means to maintain architectural models which are timely, correct, useful, and consistent.

The main problems to be tackled immediately are the creation of a suitable representation for software systems, definition of the properties which will be analysed, and creation of the mechanisms for ensuring consistency between multiple views.

References

1. Nosek, J.T., Palvia, P.: Software maintenance management: Changes in the last decade. Journal of Software Maintenance: Research and Practice **2**(3) (1990) 157–174
2. Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., Turski, W.M.: Metrics and laws of software evolution—The nineties view. In El Eman, K., Madhavji, N.H.,

[2] `http://c2.com/cgi/wiki?CritiqueOfUseCases`

eds.: Elements of Software Process Assessment and Improvement, Albuquerque, New Mexico, IEEE CS Press (1997) 20–32

3. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall (1996)
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. S.E.I. Series in Software Engineering. Addison-Wesley (1998)
5. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes **17**(4) (1992) 40–52
6. Shaw, M.: Procedure calls are the assembly language of software interconnection: Connectors deserve first class status. Technical Report CMU/SEI-94-TR-2, Software Engineering Institute, Carnegie Mellon University (1993) Presented at the Workshop of Software Design, 1994. Published in the proceedings: LNCS 1994.
7. Kruchten, P.: Architectural blueprints—The "4+1" view model of software architecture. IEEE Software **12**(6) (1995) 42–50
8. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R.: Documenting Software Architectures: Views and Beyond. Addison Wesley (2002)
9. Cazzola, W., Savigni, A., Sosio, A., Tisato, F.: Architectural reflection: Bridging the gap between a running system and its specification. In: Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering, Florence, Italy (1998)
10. Masuhara, H., Yonezawa, A.: A reflective approach to support software evolution. In: Proceedings of International Workshop on the Principles of Software Evolution. (1998) 135–139
11. Dowling, J., Cahill, V.: Dynamic software evolution and the k-component model. In: Proceedings of the OOPSLA 2001 Workshop on Software Evolution. (2001)
12. Cazzola, W., Pini, S., Ancona, M.: Evolving pointcut definition to get software evolution. In: Proceedings of RAM-SE'04, the ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution, Oslo, Norway (2004) 83–88
13. Rank, S.: A Reflective Architecture to Support Dynamic Software Evolution. PhD thesis, University of Durham (2002)
14. Garlan, D., Schmerl, B.: Using architectural models at runtime: Research challenges. In: Proceedings of the European Workshop on Software Architectures, St Andrews (2004)
15. Robbins, J.E., Redmiles, D.F.: Software architecture critics in the Argo design environment. Knowledge-Based Systems **5**(1) (1998) 47–60
16. Vandewoude, Y., Berbers, Y.: Component state mapping for runtime evolution. In: Proceedings of the 2005 International Conference on Programming Languages and Compilers, Las Vegas, Nevada, USA (2005) 230–236
17. Bennett, K., Rajlich, V.: Software maintenance and evolution. In: The Future of Software Engineering, ACM Press (2000) 75–87
18. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)