SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **41**:63–86
Published online 24 August 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.999

Assessing architectural drift in commercial software development: a case study

Jacek Rosik[1,*,†], Andrew Le Gear[2], Jim Buckley[3], Muhammad Ali Babar[4] and Dave Connolly[5]

[1]*Lero, University of Limerick, Ireland*
[2]*Nomura Plc., London, U.K.*
[3]*University of Limerick, Castletroy, Limerick, Ireland*
[4]*IT University of Copenhagen, Denmark*
[5]*IBM Dublin Software Lab, Ireland*

SUMMARY

Objectives: Software architecture is perceived as one of the most important artefacts created during a system's design. However, implementations often diverge from their intended architectures: a phenomenon called architectural drift. The objective of this research is to assess the occurrence of architectural drift in the context of *de novo* software development, to characterize it, and to evaluate whether its detection leads to inconsistency removal. *Method*: An *in vivo*, longitudinal case study was performed during the development of a commercial software system, where an approach based on Reflexion Modelling was employed to detect architectural drift. Observation and think-aloud data, captured during the system's development, were assessed for the presence and types of architectural drift. When divergences were identified, the data were further analysed to see if identification led to the removal of these divergences. *Results*: The analysed system diverged from the intended architecture, during the initial implementation of the system. Surprisingly however, this work showed that Reflexion Modelling served to conceal some of the inconsistencies, a finding that directly contradicts the high regard that this technique enjoys as an architectural evaluation tool. Finally, the analysis illustrated that detection of inconsistencies was insufficient to prompt their removal, in the small, informal team context studied. *Conclusions*: Although the utility of the approach for detecting inconsistencies was demonstrated in most cases, it also served to hide several inconsistencies and did not act as a trigger for their removal. Hence additional efforts must be taken to lessen architectural drift and several improvements in this regard are suggested. Copyright © 2010 John Wiley & Sons, Ltd.

Received 7 May 2009; Revised 10 June 2010; Accepted 15 June 2010

KEY WORDS: software/program verification; architecture consistency; architecture degeneration; reflection models

1. INTRODUCTION

The Software Architecture (SA) of large software systems is an important asset [1–3]. Design decisions made at the architectural level, also called architectural design decision, affect a system's ability to accept changes and to adapt to changing market requirements [4–7]. Architectural design decisions directly affect the system's maintenance and evolution; activities that consume a significant proportion of the effort expended over a software system's lifespan [8, 9].

However, as software systems evolve over time, they undergo changes, such as bug fixes, the addition of new features and porting to different hardware architectures. Such changes can

*Correspondence to: Jacek Rosik, Lero, University of Limerick, Ireland.
†E-mail: jacek.rosik@lero.ie

contribute to a situation where the system's implemented architecture diverges from the designed architecture [4, 7, 10–12]. In such situations, the advantages of designing an appropriate architecture may be lost [3].

Additionally, it is likely (although not yet empirically validated) that systems' architectures diverge during initial implementation, due to factors like development deadlines, or developers' limited familiarity with the architectural design. Indeed, inconsistencies introduced during development are of relatively greater importance than those introduced during software evolution, as they may render the designed architecture redundant, even before it is fully realized.

Such discrepancies between the designed and implemented architecture are collectively referred to as *architectural drift*, *architecture degeneration*, or *system degeneration*[‡], a well-documented phenomenon [4, 5, 7, 13]. Individual discrepancies are often referred to as violations or inconsistencies[‡].

Several techniques have been designed to analyse and evaluate the discrepancies that can arise between a designed system and its implementation [2, 4, 5, 7, 11, 13–15]. However, the empirical studies reporting on these approaches are mostly single-session experiments on a finished version of a system and seem to concentrate on inconsistency discovery only [4, 11, 12] (as opposed to prevention or subsequent removal). Moreover, most of these studies do not provide any feedback to the development team in order to enable them to act on the detected inconsistencies. These empirical characteristics make it difficult to evaluate the techniques as a means of controlling architectural drift during software development.

The focus of this research is to evaluate whether architectural drift can be identified during the *de novo*, *in vivo* development of a commercial software system and to characterize this drift. This research also evaluates whether the detection of these inconsistencies, allied with feedback to the development team, results in the removal of the detected inconsistencies. As a means to evaluate these issues, a 2-year case study [16] covering the development of a commercial software system (from initial design to product release) was performed at the IBM Dublin Software Lab. This longitudinal case study was in the tradition of action research in that it moved beyond reflective knowledge creation to on-going theorizing towards guiding genuinely well-informed practice [17].

The technique selected for the case study is based on Reflexion Modelling [18], Reflexion Modelling follows the generic model for architectural evaluation presented in the literature. In addition, the technique was also suggested for architectural evaluations by Hochstein [5] and later used by Knodel *et al.* [15, 19, 20]. As this case study was performed longitudinally, within the organization and team that developed the software, it provides useful empirically derived insights into the potential utility of *detecting* architectural drift as a means of *controlling* it during commercial system development.

The first paper on this research reported initial impressions of the technique's utility, and was published shortly after the commencement of the case study in [21]. The second report was on the case study itself [22] and was published shortly after the case study had finished. While that paper presented preliminary results, this paper presents a substantial revision of that work. Specifically it:

- Significantly extends the previously reported findings by characterizing the types of inconsistencies identified by the architecture evaluation technique, based on their rationale and severity.
- Identifies several new inadequacies (and thus potential improvements) existent within the current technique based on in-depth analysis of the data gathered during the reported case study. Moreover, this paper also considerably extends discussion of the 2 inadequacies/improvements discussed in the original paper.
- Bases these extensions to the conclusions on a much more rigorous data analysis phase. In this data analysis phase, content analysis was applied to the verbal data/architectural models obtained during the case study. Compared with the original paper, the new data analysis was performed at a finer level of granularity, which enabled us to report the findings quantitatively

[‡]A more precise term would be 'non-conformant architecture' but the literature tends to refer to this phenomenon using these three terms.

at a greater level of detail. These findings are expected to allow readers to gauge the size of effects observed with greater confidence and rigor.

- Reports additional data that were gathered specifically for this paper. The additional data took the form of a retrospective interview with one of the participants of the study. In this interview, his rationale was sought for some surprising findings uncovered by the study.
- Presents an extended review of the related literature in this area (Section 4) which discusses the alternative approaches adopted and empirical work carried out in this and closely related areas.

The remainder of this paper is structured as follows. Section 2 presents background information on the research problem. A general overview of the field is presented in Section 2.1, followed by the specifics of the technique used in Section 2.2. Section 3 contains the description of the case study. First, the motivation and goals for the case study are presented in Section 3.1, followed by a short overview of the subject system in Section 3.2 and the participants in Section 3.3. Details of the process and logistics of this case study are presented in Sections 3.4 and 3.5. The description of the study containing an overview of the sessions and evolution of the subject system's architecture is presented in Section 3.6, whereas the results are presented in Section 3.7. Section 4 presents a review of the related literature and Section 5 contains the discussion and conclusions drawn from this study.

2. BACKGROUND

2.1. Architecture consistency

Software understanding is said to constitute between 50 to 90% of the maintenance effort [23], and often requires a good understanding of its architecture [24]. Consequently, good architectural documentation is an important asset, helping to minimize the cost of software maintenance [6]. Unfortunately, numerous examples show [4, 11, 13] that, no matter how much effort is spent on designing an architecture, it is common for the resulting implementation to differ from the designed architecture, a phenomenon known as architectural drift.

Architectural drift results in a situation where the existing architectural documentation is of little use or even hinders developers' activities [6, 7, 11]. That means developers can become confused because they are faced with two inconsistent representations of a system (the source code and the architectural documentation), and consequently, they learn to distrust the documented architecture [25, 26]. In addition, these discrepancies may also lead to increased maintenance time and cost [4], as the original design aims have been lost. Sometimes, when a system's architecture deteriorates to a degree where further development is not feasible, these issues can even lead to a complete system re-implementation [4, 27]. Thus, we argue that control should be exercised over the architecture of a system during development and maintenance, with the aim of enforcing architecture consistency: that is, inhibiting divergences between the current, [26] implemented design and the original, as-intended, design.

Hence, for the purposes of this work, we define *Architecture Consistency* (AC) as the tasks of assessing and enforcing consistency between the *designed architecture* (DA) and the *implemented architecture* (IA) in an on-going fashion over the entire lifespan of a system. The designed architecture is represented by design documentation and is also referred to as the *high-level architecture* [10] or *hypothesized architecture* [28]. The implemented architecture, also referred to as the *source-code architecture* [10] and *concrete architecture* [28], is represented implicitly, in the implementation.

Architecture consistency assumes that implementation entities (elements of the IA) can be mapped onto design entities (elements of the DA). Relationships between elements of the IA which cannot be mapped on relations of the DA are called *architectural violations*. Likewise, relationships between elements of the IA (such as invocations) not present in the DA, or relationships in the DA, not present in the IA are called architectural violations. Finally, elements of the DA which have no representation in the IA can also be considered as architectural violations[§]. However, these latter violations, if found during development, would usually represent (as yet) missing functionality, and so are typically of lesser concern[§].

AC differs from *architecture evaluation* [13, 15] mainly in that architecture evaluation has been associated with detecting the divergences between DA and IA in a one-off exercise, sometime after system deployment [7, 11, 13, 14]. Instead Architecture Consistency implies a more continuous approach, during development and evolution, and aims to not just evaluate architectural drift, but to inhibit and recover from it.

2.2. *Maintaining consistency with reflexion models*

A suitable AC technique was required for the study and, as stated above, several architecture evaluation techniques have been proposed in the literature [11, 13, 14, 29] which may be suitable for this purpose. While little explicit detail is given on the individual processes adhered to, a general template does seem to emerge [13]:

1. Define the DA and realize it in any supporting tool.
2. Recover the IA from the system's current implementation assets.
3. Compare the two architectures and identify the inconsistencies, ideally with tool support.
4. Verify and analyse the inconsistencies.
5. Suggest changes to either the IA, the DA or both.
6. Repeat steps 3–6, after these changes have been implemented.

We chose to use Reflexion Modelling, a successful design recovery technique [7, 10, 30, 31] that closely adheres to the above schema. The original Reflexion Modelling technique is detailed in the following section.

2.2.1. Reflexion modelling. Reflexion Modelling is a semi-automated, diagram-based, structural summarization technique that programmers can use to aid their comprehension of software systems. Introduced by Murphy *et al.* [10], the technique follows a six step process:

1. The software engineer creates a hypothesized architectural model, the *High-level Model* (HLM)—or DA in the terminology of Section 2.1.
2. A dependency graph of the subject system's sources is extracted, creating the *Source Model* (SM)—or IA in the terminology of Section 2.1.
3. The software engineer creates a *mapping*, assigning IA entities to DA entities.
4. The relationships defined by the engineer in the DA are compared with those extracted from the implemented system in the IA. The results of comparison are presented to the user through a *Reflexion Model* (RM) (Figure 1). The following relationships are represented in this model:
   - A *solid edge* represents a relationship present in both the DA and the IA. (convergence)
   - A *dashed edge* represents a relationship present in the IA, but not in the DA. (divergence)
   - A *dotted edge* represents a relationship present in the DA, but not present in the IA. (absence)
5. By analysing particularly the inconsistent relationships in the RM, engineers choose to either alter the mappings, the DA, or the IA (the latter through re-factoring the source code) to address inconsistencies
6. Steps 4 and 5 are repeated until the recovered model is consistent with the system.

Reflexion's DA (an example DA can be seen in Figure 1) represents the system's architecture as logical modules and interactions between them. It is a simple 'boxes and edges' diagram, where boxes represent the logical clusterings of the system in the programmer's mind and edges represent any logical relationships that exist between these clusterings. As this model is independent of the

[§]To highlight the 'consistency-achieving' aim of our work, we will refer to all violations as inconsistencies.

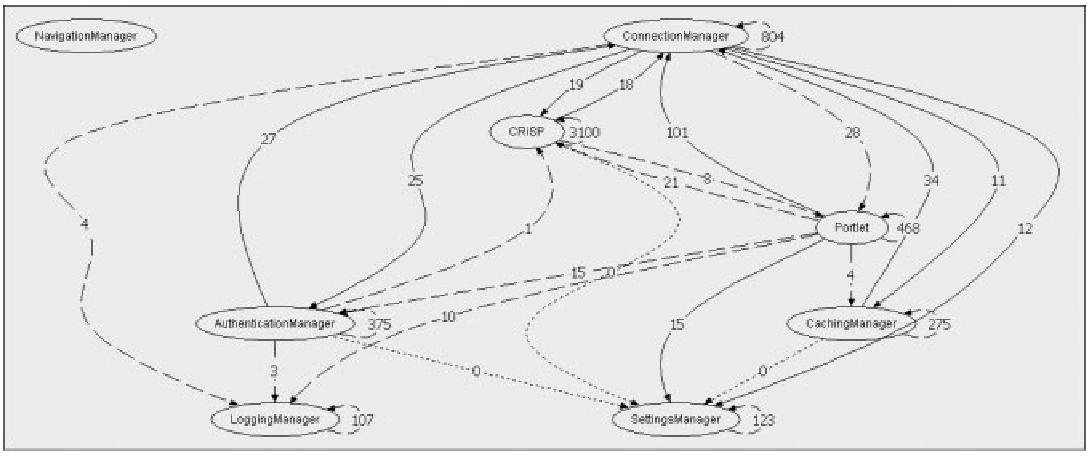


Figure 1. Reflexion model of the original DAP system.

'physical' structure of the system [10], it is similar to the *Logical View* introduced by Kruchten in his '4+1' View Model [32].

Gail Murphy anecdotally cites case studies that employ a variation of Reflexion Modelling for AC [7]. Again, her description of the process is implied only and the work seems to have been done as a one-off, re-engineering exercise. However, Hochstein re-enforces the impression that Reflexion Modelling would be useful by independently suggesting it as a means of enforcing architectural consistency in [5]. Given the successful empirical results associated with Reflexion Modelling in a re-engineering context [7, 15, 21, 28, 30, 33], we adopted this as the basis of our architecture consistency approach.

An additional benefit of choosing to tailor this existing technique is that it will allow us to reuse existing extensions to the approach [6, 28, 30, 31]. These include using CVS repositories to provide additional information on inconsistencies [6], hierarchical modelling of large architecture [28] and existing tool facilitation for the creation of initial models and mappings [34]. Although these approaches were created for design recovery processes, they should also prove useful when applied in an AC context.

However, only the last extension (that of having automated support when creating the model and mappings [34]) is used in this current work. As Tvedt mentioned in [13], a good tool and lightweight process are important factors in exercising control over a system's architecture. Reflexion Modelling's lightweight approach, as implemented by the jRMTool [34], makes it easy for interested organizations to adopt and integrate it into their existing tool-chains.

2.2.2. Adapting reflexion modelling for architecture consistency. Reflexion Modelling was selected based on its adherence to the schema outlined in Section 2.2, the freedom it allows the user in defining their architecture and its successful empirical evaluation in architecture recovery contexts [7, 18, 28]. However, an adaptation of the original Reflexion Modelling approach for this purpose is required. While this has been proposed by Hochstein as well as Murphy [5, 10] and later used by Knodel [15, 19], none of these authors have made the associated process explicit. This work builds on our earlier work [21] to provide an explicit description of the envisaged process (Figure 2):

1. Before implementation of the system commences, a designer creates an HLM, representing the system's DA.
2. During the implementation phase, developers and/or architects, update the code base and the associated mappings to the DA.
3. At any time, an RM can be generated from the DA and IA to assess the consistency of the implementation.

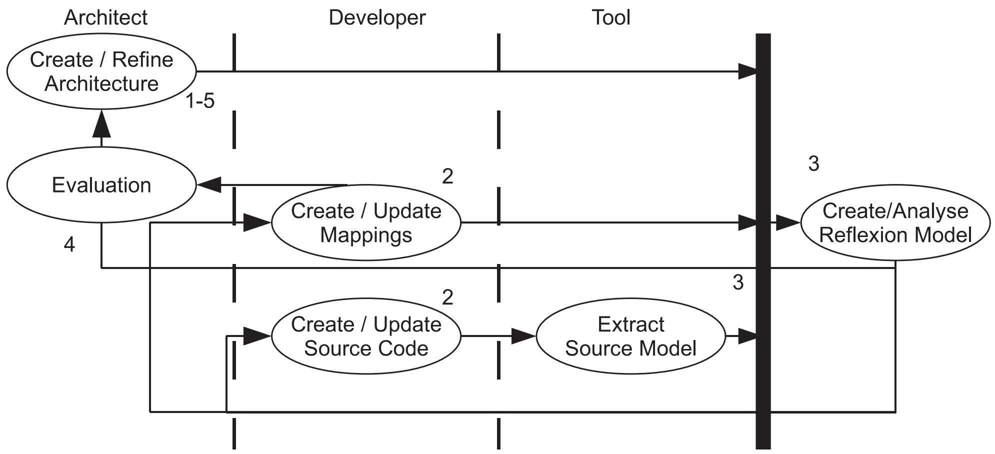

Architect
Developer
Tool
Create / Refine Architecture
1-5
Evaluation
2
3
Create / Update Mappings
Create/Analyse Reflexion Model
4
3
2
Create / Update Source Code
Extract Source Model

Figure 2. The Reflexion modelling process used in this case study.

4. Subsequently the resulting model is analysed to reveal the following:
   - Where the implementation is consistent with the design (convergences).
   - Where the implementation violates the design (inconsistencies and absences).
5. Of most interest are the inconsistencies, that is, the absence of an edge in the RM where one is expected, or the presence of an edge where one is not expected. Engineers may choose to take one of the following actions to correct the issue (derived from [11]):
   - The inconsistency may be corrected by updating the code base (changing the IA);
   - Mappings may be updated, reassigning an IA entity to a different DA entity;
   - The IA may be considered acceptable and the DA updated accordingly.
6. Steps 2, 3, 4 and 5 are repeated ensuring that the designed architecture is consistent with the implemented architecture. Indeed, these steps are repeated at intervals over the development and evolution of the software system, to alert developers and maintainers in a timely fashion as to architecture inconsistencies for removal.

As Reflexion is based on static analysis, this process enables repeated consistency assessments, before the system is fully developed. Indeed, this is the core difference between the process described above and previous attempts at design control, where evaluation was mostly one-off and post-deployment [12, 29]. The process was devised based on the opinions reported in [5, 7]. Such a process can allow a software designer to retain control over the architecture of the evolving application, thus improving its comprehensibility, availability and maintainability.

3. THE DAP CASE STUDY

Our review of the area suggests (Section 4) that, while there is evidence that inconsistencies are detected using existing approaches, there is little evidence that the identification of these architectural inconsistencies leads to their subsequent removal, in real-life scenarios. This section of the paper describes a case study which probes this issue. In this case study, the AC approach proposed in Section 2.2.2 was applied to the redevelopment of a commercial product in IBM's Dublin Software Laboratory, called Domino Application Portlet (DAP). The case study encompassed the entire redevelopment of this product over a 2-year period. During this time, (facilitated) architecture consistency sessions were held every 4–5 months with the system's developers and the effect of these sessions on the resulting architecture consistency was evaluated over time.

3.1. Objectives and methodology

The principal objective of this study was to assess the selected AC approach in a real-life software development scenario, over a realistic drift period. By assessing this AC approach, an additional objective of this study was to empirically derive requirements for an improved AC approach and associated tool.

Hence, the following research questions directed the reported case study:

1. *Does the process inform about architectural inconsistencies in implementation*?—That is, does the use of the technique described in Section 2.2.2 lead to the discovery of architectural inconsistencies during implementation and, if so, can we characterize these inconsistencies?
2. *Do the developers act on these inconsistencies to improve architecture consistency*?—This question asks if the discovered inconsistencies are ignored, removed or added to the architecture and probes why.
3. *If inadequacies are identified in this AC approach*, *what improvements can be suggested*?—Here the aim is to derive insights into additional requirements for an improved AC approach.

3.1.1. Methodological issues. At least three different methods can be used for method/tool evaluation [35, 36]: *a case study*, *a formal experiment* or *a survey*. A controlled experiment or a survey could possibly afford a larger population size and thus, findings of statistical significance. However, the constrained nature of controlled experiments would destroy the ecology validity of a study. In addition, a survey would require that participants were familiar with the technique under evaluation, which is infeasible in the case of a newly proposed technique.

The aim of this study was to characterize architectural drift in a real-life scenario. Given that the architectural drift usually manifests itself over a period of time [4, 5, 7], a longitudinal case study [16] was deemed an appropriate research methodology for this study. Hence, while the other methods were not deemed appropriate to gain insights into *in vivo* practices over the time-span required [35, 37], these methods may be considered in future for the purpose of buttressing our results and triangulation [36].

3.2. The subject system

DAP is a system that allows users to access HTML-enabled Domino applications via a WebSphere portal Server. A portlet is a pluggable user interface module that is managed and displayed inside a web portal, whereas a portal is a container for portlets which handles authentication and allows setting up of web pages. The DAP implementation relies on several framework technologies, including: the WebSphere Portal, the Portlet API, and the Apache Regular Expression Engine.

DAP 2.0 is a redevelopment of the original version of the system, DAP 1.x. The decision to re-implement the system was based on maintenance problems with the original version of DAP. Those problems originated from a practice where functionality was put in the wrong place, because there was not always enough time to locate it properly.

This led to a situation which was described by participants, as a ‘spaghetti of code’. The original DAP application consisted of 28 500 LOC, 16 packages, 95 classes and 509 files. It was expected that the complete, DAP 2.0, would be of a comparable size.

An RM reflecting the architecture of the original version of DAP was created during the first case study session. The resulting model, shown in Figure 1, showed how interdependent the components had become in DAP1.x. This, in part, led to the decision to redevelop DAP 1.x.

3.3. The participants

The case study began with only one participant. This participant, who for the remainder of this work will be called *Participant A*, undertook the combined role of architect and software developer for DAP. He was involved in the initial redesign of DAP, and the subsequent redevelopment of the system. He also had some experience in maintaining the previous versions of DAP, DAP 1.x. However, he was not one of the original developers or designers of that original system.

After 11 months, development of DAP 2.0, was handed over to a new developer. This developer, *Participant B1*, had some previous experience in working with DAP 1.x, but no experience with DAP 2.0 at that stage. He was later joined by another developer, *Participant B2*, who had no prior experience in working with any version of DAP.

Table I. Participants summary.

Participant	Development experience	Role	Projects
A	11 years	Architect/Developer	Quality engineer on early Lotus products; In-house tools for localization and process-automation; Lotus Domino Toolkit and DAP 1.x developer
B1	10 years	Architect/Developer	Localization tooling projects (C++); Websphere Portal based projects (J2EE); DAP 1.x and 2.0 developer; XPages developer (current)
B2	7 years	Developer	Java applications at different organization; Various projects at IBM; DAP 2.0 developer

A summary of participants' software engineering experience is presented in Table I. Sessions involving Participant A will be referred to as *Session Ax*, whereas session involving Participant B1 or Participant B2, will be referred to as *Session Bx*.

3.4. Process for the study

The case study started in November 2005 and concluded on October 2007, the entire development lifespan of DAP 2.0. It was initially envisaged that individual 'AC' sessions would run every 5–6 months, where mappings would be updated and Reflexion Models created and evaluated. In fact, sessions occurred at intervals of 4–5 months, several prompted by the developers themselves (Table II). Each of the sessions consisted of the following activities:

1. *Familiarization*: All new participants taking part in the study were to be set up and trained in the use of the jRMTool. Owing to the lightweight nature of the process and the tool, this activity was scheduled for 30 min.
2. *Preliminary architectural discussions*: After familiarization, all the sessions were to start with an analysis of the previous session's DA model, and mapping. This aimed at inducing architectural discussion. All proposed changes to the architecture were then discussed, applied to the DA model and applied to the mappings.
3. *Consistency check*: After the DA model and mappings were updated, the jRMTool was run against the current code base to create an RM. This RM reported on the consistency between the DA and IA. The resulting model was to be analysed in terms of any inconsistencies arising.
4. *Team analysis and discussion*: The discrepancies discovered were to be analysed by the participants. The seriousness of any inconsistencies was estimated and possible causes identified. Also corrective actions were to be discussed.

3.5. Logistics of the study

In the case study, the academic researchers took on the role of neutral observers, initially providing training to the participants in the use of the tool and subsequently being available in each session to provide technical help when required. An improved version of the jRMTool Reflexion Modelling plug-in [34], for the Eclipse Java IDE [38] was employed. This tool provides automated abilities for creating and viewing DAs, for mapping software elements to DA elements, for displaying the resulting RMs, for displaying summary information regarding the edges of the model and for displaying unmapped software elements[¶].

As can be seen in Figure 3, the mappings of the source code to the DA in the jRMTool are expressed in a basic XML language. It should also be noted that by clicking on an edge in the

[¶]It should be noted that the tool is a prototype only and suffers from both GUI and memory issues, when applied to large systems.

Table II. Session summary.

	Date	Session	Participants	Summary
1	November ’05	A0	A	Participant A was trained, created an initial DA of the envisaged architecture, and performed some architecture analysis of DAP 1.x
2	February ’06	A1	A	First assessment of DAP 2.0’s implementation and the first inconsistencies were discovered
3	July ’06	AB	A, B1	Participant A chose to use the technique to facilitate the handover to B1. Participant B1 was trained and an RM was computed. No new inconsistencies were discovered. However, old inconsistencies persisted, hidden in the DA
4	January ’07	B1	B1, B2	Participant B2 was trained. The DA created in session AB was analysed with respect to the persistent inconsistencies and a new inconsistency discovered
5	June ’07	B2	B1, B2	Participant B1 asked us to facilitate another session after a big system change occurred, but no new inconsistencies were discovered
6	October ’07	B3	B1, B2	Participants B1, B2 were asked to explore both expected and un-expected edges in the RM to ascertain if these edges hid unexpected source code relationships

Reflexion Model (the view at the top right), a user gets a list of the relationships that make up that edge (edge information). Additionally, the tool shows the source-code elements which are not included in the current mapping.

Each session was video recorded. In the video recording, the participants’ screens and remarks were captured, providing valuable data on the evolution of the models, the evolution of the mappings and the participants’ observations. After each session was complete, all the assets created during the session were collected for storage and analysis. These included:

- Audio recordings of participants think-aloud and their discussions.
- Video recordings of participants’ screens.
- Models (HLM), mappings and screenshots of the resulting RMs.
- Notes were also taken during the sessions, highlighting the important events.

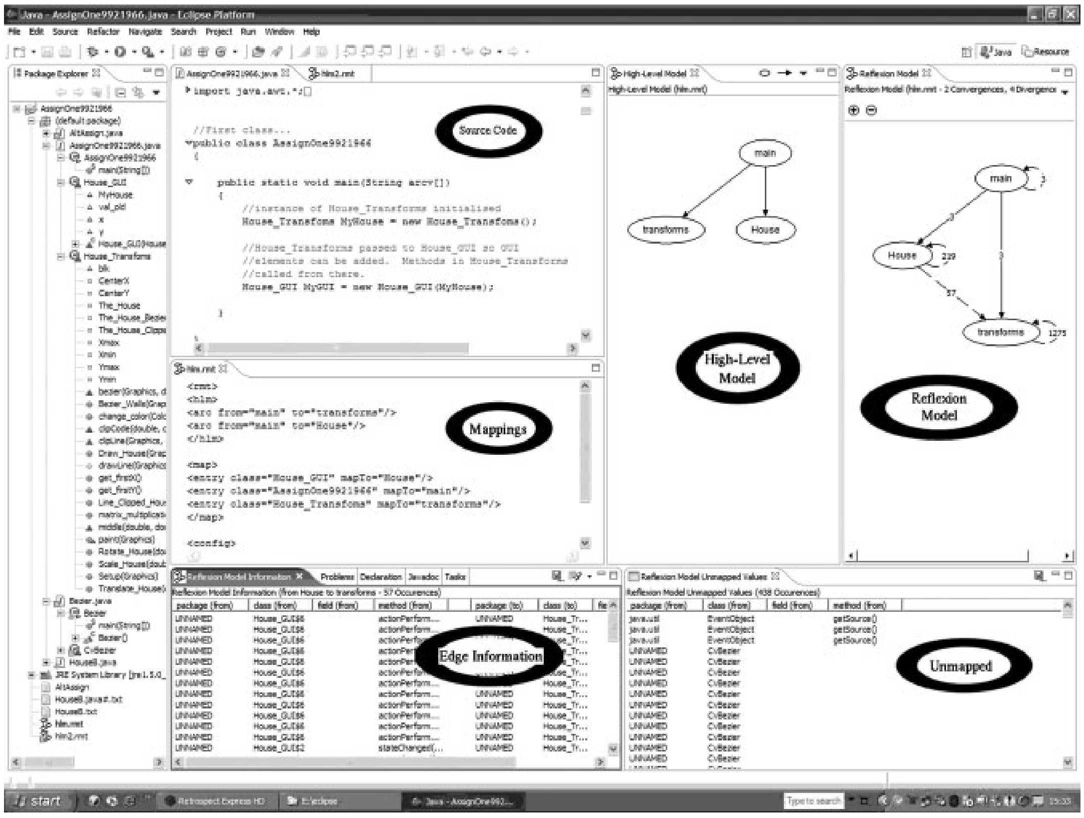
The gathered material amounted to over 6 h of audio/video recordings which was transcribed and, together with other material, thoroughly analysed and discussed to record important findings and prompt changes in future sessions. Content analysis [39] was performed on the think-aloud data and on the models gathered. In this analysis, individual phrases were categorized as discussing the architecture of the system in general, identifying an inconsistency, discussing an inconsistency, discussing the tool, and discussing the approach employed. Likewise the elements of the RM were characterized as consistent and inconsistent with the DA and, in several cases, the underpinning source code relationships were also characterized as consistent or inconsistent with the DA.

Finally, based on a review of our findings, a retrospective interview was carried out with one of the participants, in an attempt to understand the rationale behind some of our findings (detailed in Section 3.7.2).

3.6. Case study overview

At the start of the first session, *Session A0*, Participant A discussed the planned architecture for the new system and created a DA diagram on a flip chart (Figure 4). No inconsistencies for the new system were discovered during this session as no code existed at this stage.

Figure 4 illustrates all the main components originally envisaged for DAP 2.0. Before the next session, *Session A1*, further design insights occurred to Participant A, resulting in changes to the



```
import java.awt.*;

 //First class...
public class AssignOne9921966
 {

    public static void main(String arcv[])
    {
        //instance of House_Transforms initialised
        House_Transfoms MyHouse = new House_Transfoms();

        //House_Transforms passed to House_GUI so GUI
        //elements can be added.  Methods in House_Transforms
        //called from there.
        House_GUI MyGUI = new House_GUI(MyHouse);

    }
```

```
<rmt>
<hlm>
<arc from="main" to="transforms"/>
<arc from="main" to="House"/>
</hlm>

<map>
<entry class="House_GUI" mapTo="House"/>
<entry class="AssignOne9921966" mapTo="main"/>
<entry class="House_Transfoms" mapTo="transforms"/>
</map>

<config>
```

Reflexion Model Information (from House to transforms - 57 Occurences)

package (from)	class (from)	field (from)	method (from)		package (to)	class (to)	fie
UNNAMED	House_GUI$6		actionPerform...		UNNAMED	House_Tr...	
UNNAMED	House_GUI$6		actionPerform...		[illegible]	House_Tr...	
UNNAMED	House_GUI$6		[illegible]		[illegible]	[illegible]	
UNNAMED	House_GUI$6		[illegible]		[illegible]	[illegible]	
UNNAMED	House_GUI$6		[illegible]		[illegible]	[illegible]	
UNNAMED	House_GUI$6		[illegible]		[illegible]	[illegible]	
UNNAMED	House_GUI$6		actionPerform...		[illegible]	House_Tr...	
UNNAMED	House_GUI$6		actionPerform...		UNNAMED	House_Tr...	
UNNAMED	House_GUI$6		actionPerform...		UNNAMED	House_Tr...	
UNNAMED	House_GUI$6		actionPerform...		UNNAMED	House_Tr...	
UNNAMED	House_GUI$6		actionPerform...		UNNAMED	House_Tr...	
UNNAMED	House_GUI$6		actionPerform...		UNNAMED	House_Tr...	
UNNAMED	House_GUI$2		stateChanged(...		UNNAMED	House_Tr...	

Reflexion Model Unmapped Values (408 Occurences)

package (from)	class (from)	field (from)	method (from)
java.util	EventObject		getSource()
java.util	EventObject		getSource()
java.util	EventObject		getSource()
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		

Figure 3. A screenshot of jRMTool Eclipse plug-in.

architecture and a new DA. Specifically, he envisaged that the main components constituting this architecture (Portlet, CRiSP, CM and USM) should be implemented as separate servlets. This, in theory, would allow the system to distribute the work between different server nodes. Also, common code was separated from specific components, in order to reduce cloning, and thus new utility components were introduced into the architecture (Figure 5). Main components of DAP are described in Table III.

Ten months into the redevelopment of DAP 2.0, the participant working on the project was reassigned and development of the system was handed over to a new architect. Participant A chose to use the proposed approach in order to facilitate the handover to Participant B1, and asked us to facilitate this session, which can be considered an implicit recommendation of the approach.

Session AB started with an architectural discussion. The architecture had evolved into a third draft, as depicted in Figure 6[‖], where the numbers on the edges represent the high-level flow of control in the final version of the architecture. The main difference between this and previous drafts was the de-servletization of the system, a reversal of the architectural decision taken before session A1. All the components, except Portal and portal-related components, were placed in one servlet. Thus, while in the previous draft the communication was done through HTTP requests, in this draft, the communication was mainly done through method calls. Participant A stated that this overall architecture of DAP 2.0 was stable, at this point. After a discussion of the architecture, a Reflexion Model was computed in order to check architecture consistency of the current code base[‖]

[‖]Business Component and Property Broker Prototype are non-functional parts of DAP. They do not communicate with the rest of the system, and therefore, are not discussed here.

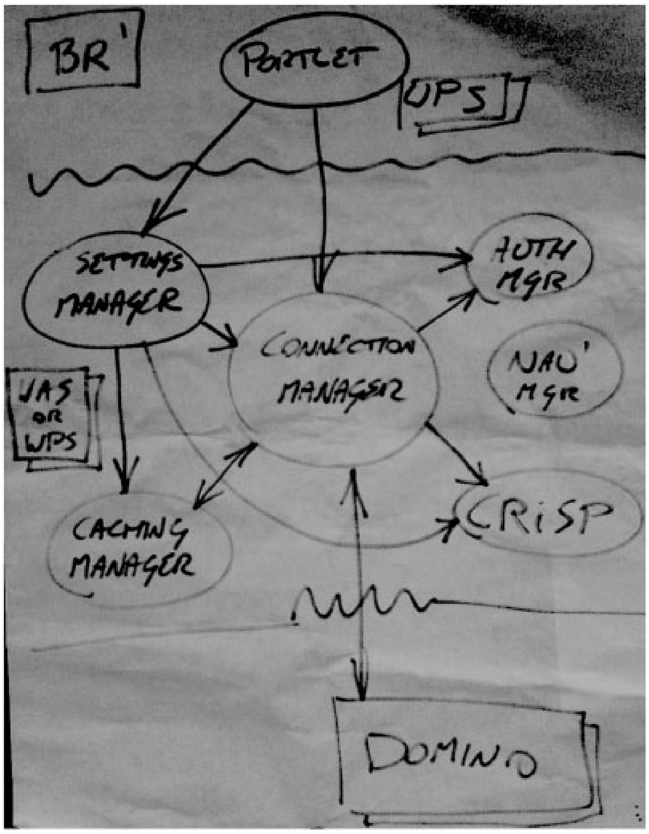


Figure 4. Initial architecture of DAP 2.0.

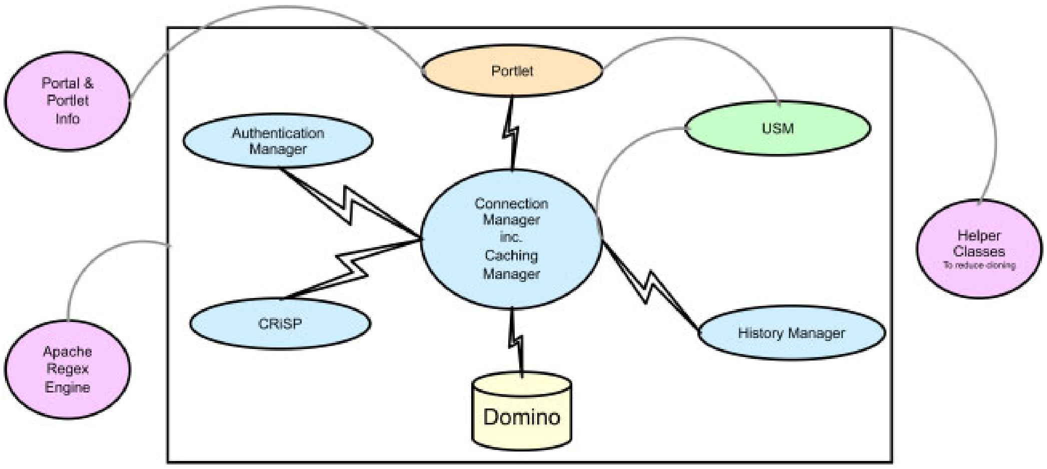


Figure 5. The architecture of DAP 2.0, as of Session A1.

After 5 months, we returned to IBM in order to run *Session B1* with the new development team of Participant B1 and Participant B2. As envisaged by Participant A, the architecture of DAP 2.0 had not altered, apart from the removal of the Business Component. The main goal of this session was to identify any new inconsistencies introduced and to analyse inconsistencies uncovered in Session A1 that were added to the DA by Participant A between sessions A1 and AB.

Table III. DAP 2.0 Components.

Component	Introduced	Description
Portlet	Session A0	Portlet represents the front-end of DAP 2.0. It is responsible for displaying the content, initial authentication and interaction with the 'outside world'. It receives requests from portal and forwards them further down the hierarchy, to the Connection Manager
Connection Manager (CM)	Session A0	CM is the core of DAP: the component that does most of the work. It passes information between all other major components and communicates with Domino to retrieve content
Authentication Manager (AM)	Session A0	AM provides authentication mechanisms for DAP. Thus, it stores user names and passwords. AM is implemented as a plug-in to allow users of DAP implement custom authentication modules
Caching Manager (CHM)	Session A0	CHM provides a cache for static Domino content (like images). Whenever Connection Manager receives a request for Domino content it will first check the cache, and only make a connection to Domino when the requested content is missing
Content Re-writing Service Provider (CRiSP)	Session A0	CriSP is the component which actually modifies Domino content to allow its use within the portlet. These modifications are done through the use of this Regular Expression Engine. This is the only legacy component: one which comes directly from DAP 1.x
User and Settings Manager (USM)	Session A0	USM stores all the settings and regular expressions used in DAP. The settings can be stored on several different levels: per application, per portal and per user. It is the only module apart from the CM which is extensively referred to by other modules
The Apache Reg-Ex Engine (REE)	Session A1	This module implements the regular expressions. At the beginning this module was envisaged as part of CRiSP, but the functionality provided proved to be useful to other components. Thus, it was separated and became a utility module
Helper Classes (HC)	Session A1	HC is a collection of classes and utility functions common to all the components. Its main purpose is to increase code re-usability
Portal and Portlet Info (PI)	Session A1	Similar to the HC, this component is a collection of utility functions specific to portal and portlet components
Locale Manager (LM)	Session AB	LM was introduced to handle different languages supported by DAP. Its role not only involved translation of GUI messages but also translation of internal exceptions and errors

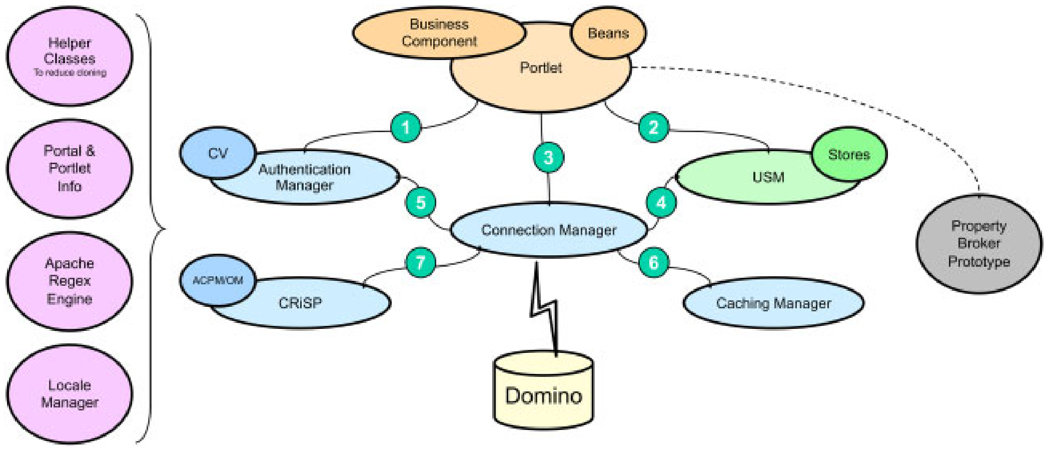


Figure 6. The final High-Level architecture and control-flow of DAP 2.0.

Another session, *Session B2* was run in response to a request from Participant B1, 5 months later. In the interim, the system had been ported to a new version of the underlying WebSphere Portlet API. However, although this change involved extensive modification at a low-level, there was no impact on the DA and no new inconsistencies were discovered in this session.

Table IV. Analysed Edges and inconsistencies Discovered*.

From	To	Type	S. C. relations	False Negatives	Comments
Session A1					
Portlet	USM	D	1	NA	Trivial, constants
CM	CRiSP	D	4	NA	Trivial, constants
USM Stores	USM	D	1	NA	Trivial, constants
USM	CRiSP	D	3	NA	Legacy
CRiSP	AC Matcher	D	2	NA	False positive
Session B1					
CM	CRiSP	C	6	NA	Divergence introduced into HLM
USM Stores	USM	C	1	NA	Divergence introduced into HLM
USM	CRiSP	C	9	NA	Divergence introduced into HLM
CM	HPI	D	1	NA	Trivial, misplaced functionality
Session B3					
CM	CRiSP	C	5	2	Divergence introduced into HLM
CRiSP	CM	C	3	3	Divergence introduced into HLM
USM	CRiSP	C	9	9	Divergence introduced into HLM
USM Stores	USM	C	1	1	Divergence introduced into HLM
AM	AM CV	C	44	41	Trivial, constants
Portlet Rules	Portlet	D	1	—	Trivial, misplaced functionality
Portlet	AM CV	D	2	—	Trivial, constants
CM	HPI	D	1	—	Trivial, misplaced functionality
LM	CM	D	1	—	Trivial, misplaced functionality

*Note that the same edges were sometimes analysed in different sessions. For example, CM-> HPI was studied in session B1 and B3.

The last session, *Session B3* was run after the active development of DAP 2.0 came to an end. The first version of the application had been delivered for testing before its final release and the code base was fixed. There had been no high-level architectural changes in the interim. The objective of this session was to assess a possibility that arose in session B1: namely that convergent edges in the DA may hide unexpected or undesired source code relationships (that they were, in fact, false negatives).

We targeted eight edges in the session B3 of this study. Some of these were edges artificially introduced into the DA by Participant A and suspicious edges like the cyclic link between the AM and the AMCV. These were supplemented by randomly selected edges.

3.7. Results and discussion

In this section, we present the findings from the analysis of the data gathered during this case study.

3.7.1. Characterizing architectural drift during implementation. During the first Reflexion Modelling exercise, i.e. Session A1, five inconsistencies were discovered. One of those inconsistencies was found to be mistakenly omitted from the design. Of the remaining four inconsistencies, three were considered trivial and one was considered a serious inconsistency resulting from the inclusion of legacy code (Table IV).

No new architecture inconsistencies were discovered during Session AB, although it is worth noting that the inconsistencies discovered during the previous session had been added to the DA as 'expected relationships'. This prompted us to more thoroughly re-analyse these inconsistencies in later sessions.

In Session B1, the Reflexion Modelling process was run against the current code base of DAP, resulting in a complicated diagram containing 23 nodes and 62 edges. One new inconsistency was discovered. This was a surprise for both participants, even though it was likely that one of them had introduced the inconsistency (Table IV). No new inconsistencies were discovered in session B2.

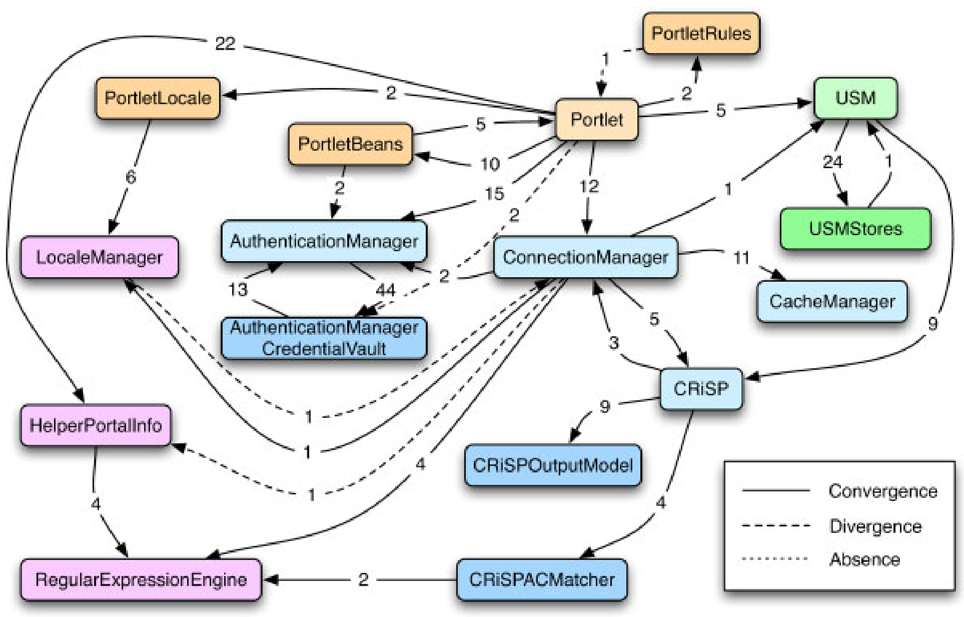


Figure 7. Reflexion model of the final DAP 2.0 architecture.

A recreation of the RM model generated during Session B3 is shown in Figure 7.[**] This session was designed to assess the possibility of false negatives and thus focussed on specific, convergent relationships in the HLM. Several of these edges were indeed desired dependencies, and as such, did not provide any interesting insight.

However, five out of the eight convergent edges analysed in session B3 provided interesting findings, suggesting that convergent edges often hide divergent source code relationships when using this technique. Admittedly four of these five were the converging edges introduced by Participant A into the DA in response to discovered inconsistencies. Even in these edges, though, the source code relationships underpinning them had doubled in number over time, from 9 initially to 18, showing increased actual inconsistency.

One of these four edges was between CRiSP and CM. This was introduced into the model after the first session to hide the accessing of (misplaced) constants. The edge also hid another call to a utility function, used to manipulate URLs. This function was mistakenly placed in the CM. While the participants were willing to accept the placement of constants, they stated that the utility function should be moved to the helper module and considered this violation more serious.

The one truly convergent edge found (with divergent source code relationships) was between AM and AM CV. It contained 44 source code relationships, 41 of which were identified as divergences in session B3. The participants stated that the relationships represented access to constants which should have been placed in the AM module. They stated that only the three remaining relationships represented legitimate calls, to initialize the AM CV component.

Four new inconsistencies (misplaced functionality or inappropriate constant location) were discovered during this session.

Table IV lists the interesting edges which were analysed over the course of the case study. As the 'Type' column shows, many of these were 'D'ivergences (probable architectural inconsistencies) but some, particularly in session B3, were 'C'onvergences, most of which were convergences introduced by Participant A in response to inconsistencies.

[**]The Helper component has been left out of this diagram as it is a collection of utility classes and all components are allowed to communicate with it.

Interestingly, in most cases the developers were genuinely surprised by the inconsistencies identified, indicating that they would probably have overlooked them without the intervention (see the quotations from the think-aloud below).

'…why! ? ...oh! that's constants again.' [Part. A]

'You've seen my surprise the last time, when we ran the tool against the existing code base.' [Part. A]

In total, 13 utterances reflecting surprise at architectural inconsistencies were made over the course of the case study. An additional measure of the approach's usefulness is the enthusiasm towards the approach, as expressed by the participants. The approach was recommended to the new development team by Participant A and they in turn, expressed an interest in continuing to use it. This is again illustrated by the participants' comments (seven such comments were discovered after the transcripts were analysed):

'I still think it would be useful to continue to use it on a semi-regular basis until DAP actually goes out the door.' [Part. A]

'When were you thinking of running that analysis thing-y again? We've had to make some API changes and need to check it' [Part. B1]

'…and do you thing you'd have documented all them without using the tool?' [Researcher] 'No! [laughs]…' [Part. A]

3.7.1.1. Discussion of results. The results show that architectural drift occurred during implementation and that Reflexion Modelling was useful in identifying it, in the majority of cases. During the initial stages of this case study, when DAP was still a one-man development effort, five architectural inconsistencies were highlighted by the tool, reflecting nine violating source code relationships and one oversight in the initial DA. This is an interesting finding as intuitively, one would expect few inconsistencies when the architect is also the only developer (as that developer should have a clear vision of how the system should be implemented).

This intuition is supported by the empirical findings of Murphy *et al.* [7] but is not supported by the findings from this case study. However, the empirical study presented in [7] was performed on a smaller system, which might have contributed to the differing results.

Later, when DAP was developed by a team of two developers, five new inconsistencies were discovered by the approach, reflecting six violating source code relationships. Hence overall, nine edges were identified by the technique as representing architectural inconsistencies over the 2-year lifetime of the project. These 9 edges reflected 15 violating source code relationships.

Further architectural drift could also be identified on more fine-grained examination of selected edges in session B3. This examination showed that, in falsely conformant edges (the edges introduced by Participant A after session A1) the number of source code relationships underpinning the edges tended to increase. For example, when the edge between USM and CRiSP was analysed in detail, it was discovered that the number of non-conformant (source code) relationships underpinning the edge grew over time, from 3 to 9.

In addition, and perhaps more significantly, in one of the truly consistent edges studied, the vast majority of source–code relationships underpinning that edge were inconsistent (41 out of 44). In this case, it could be stated that the RM 'encouraged' the participants to overlook that edge, based on its 'consistent' status: a status that reflected only 3 of the 44 source code relationships underpinning that edge. Indeed, the participants did overlook this edge until prompted otherwise by the researchers.

If such obfuscation is a common occurrence, it is a serious flaw with the technique, as it actively hides architectural drift and thus causes architectural inconsistency. This is reflected by the following quotation by Participant B2:

'Not a sin, but I'd say its a real problem... and what I mean it disguises, it may disguise real problem...' '...we have to go through that every time [analysing the dependency in detail].' [Part. B2]

Indeed, Murphy *et al.* [7] acknowledge this limitation of the approach implicitly (in an architecture recovery context) when they describe Reflexion Modelling as ‘approximate’.

Unfortunately, our current findings offer only preliminary insight into the prevalence of this issue. Specifically, in session B3, only eight convergent edges were identified for analysis. Of these, four were falsely conformant (introduced by Participant A after session A2). Hence, only four truly conformant edges were assessed and, of these four, only one demonstrated this issue. However, it is our intuition and contention that consistent edges would often hide inconsistent source code relationships. This would seem to be particularly true in larger systems where the DA abstracts away more from the source code (that is, where each DA edge represents a greater number of source–code relationships). Potential solutions for this issue are discussed in Section 3.7.3 (under the sub-heading ‘Approximation Issues’).

The discovered inconsistencies can be generally divided into three main categories:

- *Legacy Code*: One inconsistency was introduced because one of the components in DAP (CRiSP) is almost entirely taken from the previous version of the system. Developers were reluctant to change this code, due to its complexity and associated high risk. Consequently, there were several undesired relationships between *USM and CRiSP*.
- *Genuine Omissions from the Designed Architecture*: One inconsistency arose because the system designer simply forgot to add a relationship between elements of the DA at design time. This inconsistency is interesting because it demonstrates that inconsistencies arising in the source code can sometimes be correct and need to guide the DA, rather than guide correction to the code.
- *Trivial*: Most of the remaining inconsistencies were considered trivial by the participants. That means they did not consider them as a serious threat to the architecture. These could be further subdivided into two categories:
    - *Misplaced Functionality*: This type of inconsistency indicated that new functionality had been placed in the wrong location, and usually referred to a single method. Our audio data suggested that this was done for convenience or because developers simply lacked knowledge of the designed architecture.
    - *Constant Access*: Typically, in these inconsistencies, an offending constant was placed in the wrong module.

3.7.2. Does inconsistency identification lead to removal? While architectural inconsistencies were identified, the approach was not good at enforcing architectural consistency. Most of the discovered inconsistencies in session A1 remained until the final session, session B3. The inconsistencies that were removed were removed as a side-effect of development, rather than an explicit action aimed at inconsistency removal. Likewise, an inconsistency introduced by team B in session B1 persisted until the end. This is an interesting finding, as all the evaluation studies presented in this area (Section 2.1) to date, limit themselves to identification of inconsistencies only. In contrast, this research suggests that *inconsistency identification is not of itself sufficient to ensure architectural consistency*.

However, it should be noted that the architects often proposed rationale for not eliminating the inconsistencies: their trivial nature, performance savings and the difficulty associated with re-factoring legacy components (coupled to the time constraints imposed on the developers). While the first rationale suggests that architectural inconsistencies are at play, the latter 2 rationales are ambiguous: ‘Performance’ suggests that the DA should be changed to reflect this desirable quality attribute and ‘reuse of a legacy component’ suggests a conscious decision which justifies the inconsistencies introduced, again suggesting a change to the DA.

These interpretations were re-assessed during a short retrospective interview with Participant B1. When he was asked about the ‘trivial’ violations, he stated that they were so small that they ‘*didn*’*t impact enough on the architectural design to make the changes* [*worthwhile*]’. The use of the word ‘enough’ suggests that he perceived a trade-off with making these changes. Interestingly, his subsequent remarks suggest that this trade off appears to be in terms of the risk of ripple effects, rather than the time and effort expended:

'*… could see that they fell outside the expected design but were not major working parts of the overall product and so could be left in. Maybe trying to fix these 'minor' issues would've possibly caused larger issues to appear and so made them not worth exploring...*'

An obvious alternative open to the development team was to employ Eclipse's refactoring tools to address the inconsistencies. However, the participant again referred to the risk associated with this approach:

'*as powerful as they are, you can really alter a lot of code very quickly and so introduce unseen problems also very quickly*'

Here the implied risk is two-fold: causing ripple effects in the code base and causing the programmer to become disorientated, due to a new (refactored) system structuring:

'*Losing track of where everything is when you are already used to a design structure is also a factor…once a developer who inherits code gets used to a system* (*even with architectural quirks*) *they can be loath to change it to such a degree*'

Thus, the participant suggested that refactoring the legacy code was too risky and that the DA should be updated to reflect the IA. Likewise the participant suggested that the DA should be updated to allow for the architectural shortcuts made to improve performance, when time constraints prohibit a more designed solution:

'*if performance can be gained by breaking an architectural design*, *then this can sometimes be acceptable…time pressure is probably the main factor in some of the decision making that goes on during such a development*.'

3.7.3. Improving the approach

3.7.3.1. Persistence of inconsistencies. The persistence of the inconsistencies could be attributed to the batch-processing nature of the proposed approach. In designing the process for this study, periodic evaluations were deemed sufficient to maintain control over the architecture. This position was supported by the literature [2, 13]. However, in the delay between each session, it is likely that the primary focus of development switched to other parts of the code base, and that the developers might have been reluctant to revisit already 'closed' code.

Another interpretation is that the developers might have been unaware of parts of the designed architecture as they developed the system and they might have unintentionally introduced inconsistencies. If warned in real time, the developers might have introduced fewer inconsistencies and removed more [40]. This possibility is elevated in the case of trivial inconsistencies which do not significantly affect functionality, and where the perceived need to revisit for consistency is less. The majority of inconsistencies found in this case study were of this type. Thus, these findings argue for real-time notification of architecture inconsistency to developers, possibly through some sort of Intelli-Sense type prompt.

3.7.3.2. Developer-Hidden Inconsistencies. Participant A, in session A1, chose to introduce several inconsistencies into his mapping between the IA and the DA. These artificial dependencies exacerbated architecture drift in several instances and caused new participants difficulty in determining that they were inconsistencies, let alone determining their raison d'etre.

This argues for the separation of architect and developer roles: a separation that was not evident in our case study. This approach would allow the developer-introduction of inconsistencies while alerting the architect and forcing the developer who introduced the inconsistency to annotate the offending elements. Thus, work could progress while also allowing architecture-improving code-driven changes to emerge (as was observed in our study). However, it would make the inconsistencies explicit to the architect, thus focusing their attention on the updates of the system relevant to them: incorrect code changes that need to be rectified and code changes that suggest a change in the DA.

3.7.3.3. Approximation issues. When the developers were presented with the 'corrupted' architecture during session B1 (the RM which was based on partially incorrect mappings) they failed to identify any of the falsely-convergent inconsistencies, although one edge did raise their suspicion. These findings made us aware that the developers were highly led by the RM. That is, edges that were shown as consistent with the DA, were assumed correct. In fact, they could be incorrect due to incorrect mappings generated by the developer. Although, the iterative nature of the approach increases the chances of detection of such incorrect mappings (in fact participants did refine their mappings in response to discovered violations), these 'approximate' issues [7] of the original Reflexion technique, should be empirically assessed by having more than one participant map the source code onto the designed architecture. Then some measure of inter-rater reliability (Cohen's Kappa [41]) could be used to check the consistency of their mappings, thus assessing this approximation issue. Unfortunately, in this study there was only one participant at the start and so this approach could not be used. Additionally when two participants were involved, the 'heavy-weight' nature of such an approach was deemed prohibitive in this in-vivo context.

More seriously, in the current Reflexion Modelling technique, each edge in the RM typically represents many source–code relationships. Hence, it is possible that a 'consistent' edge can represent several expected source–code relationships (consistencies) and also a number of unexpected source–code relationships (inconsistencies). These unexpected source–code relationships are effectively obscured for the developer by the 'expected' edge status, leading to false negatives.

This obfuscation issue could be remedied by refining the edge types currently used by Reflexion. This enhancement could, for example, allow for edge annotation, enabling description of the source–code relationships allowed in the context of that edge. Alternatively, the enhancement could allow for sub-typing of edges between specific sub-sets of the elements in each DA node, allowing for certain public interfaces and disallowing inappropriate accesses.

3.7.3.4. General tool related issues. The automatic layout algorithm that is used by jRMTool to construct the RM is problematic. The topology achieved by this layout algorithm differs considerably from the topology of the participant's original DA. This makes the models hard to compare, particularly for larger software systems. In the case study, we observed that participants had substantial problems tracing the dependencies on the DAP RM (which is a relatively small system). To address this problem, the current presentation layer would have to be updated, to achieve consistency between the topology of the DA and the RM.

Other tool-related problems were also discovered during the case study: The prototype nature of the supporting tool meant that it sporadically crashed or performed very badly. Additionally, some of the files had to be edited by hand (text editor), as opposed to being edited through the supported model editor. Although the tool-related issues described here may be considered technical only, it is our belief that they are an important factor contributing to whether the particular technique will be adopted or not. Similar conclusions are drawn from different case studies performed by our team: [31, 42].

3.8. Validity threats

Important for any empirical study is an assessment of its validity. Validity refers to the meaning of empirical results [43] and can be categorized under three headings [44]:

External validity is the degree to which the conclusions of the study are applicable to the wider population of software development contexts. Our study was performed on a commercial system where real programmers and architects performed *in vivo* development tasks. Hence, the study had high ecological validity, a subset of external validity that refers to the degree to which the study is representative of reality.

In terms of generalizing the finding to a wide population of systems, this study focussed on only one system, of relatively small size. Possibly, when bigger systems are being developed, more formal architecture-modelling approaches may be utilized, resulting in less architectural drift. Indeed, this suggestion seems to be supported by Knodel *et al.* [19] where a similar technique was applied in a commercial organization within a more formal framework (see Section 4).

With regard to the generality of the results across developers, this study employed only three participants. In this small development team, inconsistencies were identified using the technique but not removed. In fact, inconsistencies were identified when only one developer was responsible for the system's development. Again, a larger development team may utilize a more formal process that serves to inhibit architecture inconsistencies. (Alternatively though, a greater number of developers, and the greater communication requirement this implies, might serve to increase the presence of architecture drift.) Thus, this work can be seen as one data point, specifically addressing small software development teams, and should be buttressed by further studies that test its findings' generality over different software development contexts.

Internal validity. Internal validity refers to the degree to which independent variables (and only independent variables) affect the dependent variables in a controlled experiment. Hence, it is of lesser relevance in industrial case studies, where control over variables is impossible to achieve. However, it should be noted that the architectural inconsistencies uncovered by the approach were not found outside of the sessions, suggesting that the sessions were largely responsible for their identification.

This does not necessarily imply that the Reflexion process itself was responsible. It should be noted that participants' typically did not dedicate time to search for violations outside of our empirical sessions. Indeed, their vigour in searching for violations during the sessions may have been increased by our presence and so, part of our findings may be attributed to this Hawthorne-like effect [45].

If this effect was entirely responsible though, our presence should have prompted the developers to remove architectural inconsistencies. Instead, this was not the case, suggesting that we observed more typical behaviour, not moderated positively by our interventions.

Construct validity refers to the degree to which the structure of the experiment affords the measurement of the researchers' focus. In this regard, the experiments in this publication demonstrate high construct validity. The longitudinal, *in vivo* case study allowed architectural drift be observed over a realistic time-span and also allowed time for inconsistency removal. Developers' identification of inconsistencies, failure to remove inconsistencies and failure to identify source code inconsistencies hidden in convergent edges were directly observable through the video recordings and through the successive Reflexion Models obtained.

4. RELATED WORK

Given the importance of maintaining the consistency between the designed architecture and the actual implementation, SA evaluation techniques have gained popularity [4, 5, 7, 11, 13, 14]. Most of these approaches deal with already implemented systems, assessing the level of degeneration and pointing out the problematic areas. Thus, they are designed to be applied during system maintenance, where the implemented system already exists. The techniques typically require the software engineer to produce their model of the designed architecture of the system and to check that model against the, subsequently recovered, implemented architecture. The models can be represented diagrammatically or textually, with the implemented architecture typically captured automatically by a parse of the existing system.

In [11], for example, researchers generated hierarchical, diagrammatic representations of the designed architectures of two open-source systems (The Linux Kernel and the VIM editor). They then compared the representations with the implemented architectures of the systems. Their purpose was to retrospectively repair the architectures of these systems through an approach which compared (what they refer to as): the *Conceptual Architecture* (DA) with the *Concrete Architecture* (IA). Design inconsistencies were called *anomalies* and two types of anomalies were defined:

- *unexpected dependencies* (divergences)
- *gratuitous dependencies* (absences)

They outlined four possible courses of action to deal with the discrepancies identified:

- *Splitting* a architectural entity;

- *Kidnapping* the source code elements from one model entity to another;
- *Changing the source* code to conform to the architectural constraints;
- *Changing architectural dependencies*.

Thus, their approach aligns closely with the approach presented here. While the authors did carry out an evaluation on the Linux Kernel, it was performed without the involvement of the original developers and no source code was modified. Hence, it was purely a modelling exercise and no results were communicated back to the development team. In the case of their evaluation of the VIM editor, the results were communicated to the development team, but there was no explicit discussion of how these results affected further development of the system.

Another process, similar to Reflexion Modelling was defined by Tvedt, Lindvall and Costa in a series of works [5, 12, 13, 29]. The proposed process evolved over time and the final process seems to align with the process presented here (see Section 2.2), although more detail would be required to ascertain this. For example, while the authors claim that the process was designed with the intent of minimal disruption to the team's usual activities and would fit seamlessly into existing development environments, they also suggest that the process requires a large overhead [13]. In fact, they mention that one iteration of the evaluation took several weeks to perform. This is not characteristic of the approach proposed here.

Tvedt, Lindvall and Costa used their process to perform a set of architectural evaluations on a software system being re-implemented, called the Experience Management System [12, 29]. Their concrete (as implemented) view was useful for demonstrating to management that the project was succeeding. But, even with the team's best efforts, the desired architecture could not be fully achieved without proper tool support. Using their technique, without tool support, they were again only able to *document* architectural inconsistencies after deployment.

An improved version of their technique has been used in another case study of a proprietary Simulation and Analysis Tool in [13]. However, as in previous studies [12, 29], the technique was only applied to a system after the implementation, to retrospectively check for architectural inconsistencies in hindsight. Additionally, the results of the evaluation were not feed back into the development of the system. Thus, like the other evaluations described above, their approach is contextualized as being more evaluative, and reverse-engineering oriented, rather than being a facilitator of forward engineering.

An interesting approach is presented by Sefika *et al.* in [14]. The authors combine static analysis with dynamic visualization through the use of their tool, called Pattern Lint. Pattern Lint supports consistency checking at various levels of abstraction from low-level rules (Prolog statements), through architectural rules (design patterns, architectural styles) to heuristic rules (cohesion and coupling). The process is similar to the one represented by Reflexion Modelling and executes as follows:

1. The designer specifies program entities to be analysed in the design model.
2. A parser and database generator analyses the source model creating a database of the implementation-level relations.
3. Model rules (Prolog rules) generated by the designer are then matched with the information in the database using a Prolog engine.
4. The results are matched by a static correlator to produce the visualization.

The tool they developed was used by the authors to check the compliance of a multimedia operating system developed by the same team. As a starting point, adherence to the mediator pattern [46] was verified using static analysis. A mediator pattern specifies a central component mediating all the communication between other components: a component is allowed to only communicate with the mediator and no other subsystem. One inconsistency was discovered where a component skipped the mediator and communicated directly with another component. After discussion, the authors identified that this was due to the performance optimization. However, extensive dynamic analysis suggested that the code responsible for the inconsistency was never used. Thus, the authors suggested that this performance optimization was rarely used, and should be removed, but again no comment is made on whether the inconsistency was acted upon.

The combination of static and dynamic analysis represented by this approach is an interesting option, although it seems to be specifically aligned with re-engineering tasks, as it requires a running (i.e. implemented) system. In addition, the approach seems to support only pre-defined architectural styles and design patterns. It does not allow for a user-view of the system, representing any possible logical partitioning.

Knodel *et al.* [2, 15] used a Reflexion Modelling-based tool called SAVE for the purpose of architecture evaluation. It was used to analyse software systems in nine scenarios [15]. The systems analysed were either academic, Open-Source or commercial in nature and of varying sizes, ranging from 10 to 600 KLOC. One of the case studies, for example, analysed the architecture of the Apache Tomcat web server [15]. Although, the primary goal of the evaluation was to assess the feasibility of SAVE's visualization model, some architectural inconsistencies were discovered and some questions regarding the system's architecture were raised. However, similar to all the other case studies reviewed to date, the work was done in a re-engineering context, as one-off evaluations, with no comment on the system's subsequent evolution.

In their later works Kolb [47] and Knodel [19] report experiences from using SAVE with their industry partner as a part of their PuLSE-DSSA [47] method. Although initially offered as an additional service, consistency checking was adopted by the company involved as a standard instrument for ensuring high quality of products at the organization. In this case, identified inconsistencies, in already deployed systems, were fed back to the development team and a formal process adopted whereby these inconsistencies were removed. Hence, while this study cannot comment on inconsistencies introduced during the initial development stage of the systems' lifecycles, it can comment on inconsistency removal in the context of system evolution for large product teams in that company. The results presented show a promising trend: a decrease in the number of inconsistencies over the product's life time. It should be noted however, that due to the product line [48] nature of the products, formal architecture modelling approaches were employed (in contrast to the project presented in this work), possibly resulting in increased architectural awareness and decreased architectural drift.

In their recent works Knodel *et al.* [20] and Eichberg *et al.* [49] present a tool which provides real-time alerts when developers introduce architectural evaluations. As such, their works closely align with the requirements proposed in this work (see Section 3.7.3). However, neither Knodel or Eichberg provides evidence or a rationale as to why a continuous AC checking approach should be provided. In addition, while they both performed evaluations of this approach, Eichberg [49] evaluated only the performance aspects of performing the required real-time static analysis on a software system, and did not try to prove that continuous checking was better than a batch-processing approach. Knodel's [20] evaluation was more directed at this goal but he performed it on student subjects split into two groups: one which used the live version of this approach and one which did not use any AC approach at all. Thus, he did not prove that the real-time version of his approach is better than the 'batch processing' alternative.

5. DISCUSSION AND CONCLUSIONS

In this paper, we report the objectives, logistical details and findings of a longitudinal case study performed to study architectural consistency issues in a commercial software development context. The evidence gathered indicates that the architectural drift did occur during the redevelopment of the subject system, even when the developers had a clear vision of how the system should be implemented at the architectural level. The inconsistencies that arose were due to an oversight into the original DA, dependence on legacy code and largely trivial misplacing of constants and methods.

The approach adopted was successful in detecting several of the introduced architectural inconsistencies, but it should be noted that:

- It also served to obfuscate violating source–code relationships which underpinned consistent, or expected, Reflexion Model edges.

- Inconsistency identification did not imply inconsistency removal. In fact, in this case study, none of the inconsistencies identified were explicitly removed. This is a particularly interesting finding, as the other empirical studies in this area have tended to measure their success as the number of inconsistencies that they identified. Our findings suggest that such measures are incomplete in the context of enforcing architecture consistency.
- There were several related reasons why developers chose to avoid inconsistency removal, in this context. For example, the issue of introducing ripple effects into the code, in order to address trivial inconsistencies, seemed too big a risk for the developers. While refactoring tools seemed to be a possible alternative, the widespread structural changes that these tools can introduce into the code base, allied with the developers' familiarization with the existing code base, made this approach seem unattractive also.

These findings suggest that caution must be exercised when employing this, or a similar technique, for architecture consistency, as the technique in itself does not necessarily preserve the designed architecture.

Despite the deficiencies in the approach presented here, we encourage practitioners to consider applying a design control technique in their environments, as the use of such techniques will allow the engineers to better understand the relationship between the implemented architecture and the designed architecture. Such an analysis may also reveal potential problem areas in the implementation, the design or both. At the moment, Reflexion seems to be an appropriate, lightweight and open-source technique to identify many of the inconsistencies introduced, and was popular among the practitioners who used it here and in other studies we have performed [50]. However, to better suit the requirements of architecture consistency the above caveats must be addressed. This suggests the following improvement to the approach, which our future work will address:

- The obfuscation issue present in the approach could be remedied by refining the edge types currently used by Reflexion. This enhancement could, for example, allow for edge annotation, thus facilitating architects in communicating the rationale behind the existence of certain edges, particularly those edges which represent 'acceptable inconsistencies'. The enhancement should also allow sub-types of edges to exist between specific sub-sets of the elements in each DA node (allowing for certain public interfaces to a node while retaining encapsulation of other private methods and attributes).
  However, a key factor contributing to the popularity of Reflexion Modelling is its lightweight nature, and it is envisaged that these sub-typing enhancements would be optional and minimal. That is, depending on the project's needs, the architecture could be modelled as a simple box and edges diagram or could use the more sophisticated edge sub-typing suggested here.
- To better facilitate the removal of the inconsistencies and thus make the technique more suitable for architecture consistency, we propose employing continuous consistency checking, rather than the batch processing approach prototyped here. According to Layman *et al.* [40], the longer an issue persists in the source code, the harder it is to fix. Thus discovering the inconsistency at the very moment of introduction should increase the chances of the inconsistency's removal. These alerts (as proposed by Eichberg and Knodel [20, 49]) should also increase the real-time architectural awareness of developers who sometimes introduce inconsistencies because they are not fully cognisant of the architecture.
  Indeed, if this approach could be integrated with IDEs like Eclipse, the alert mechanism could actually occur in advance of inconsistency introduction. For example, IntelliSense could prompt (via colour coding) the developer on the non-violating and violating classes/methods/attributes, before they complete their references to these classes/methods/attributes. This extends the real-time approach of Eichberg and Knodel to a predictive approach and acts to increase architectural awareness, *as and when* required.
- At the process level, an ancillary support would be to segregate the architect from the developer and to alert the architect when a developer commits offending code. This would at least prompt discussion of violating changes in an architectural context and so should prompt architecture consistency.

Work is currently under way to create a tool that supports this set of enhancements. However, it should be noted that, due to the small population involved in the case study and the relatively small system size, the results of this work need buttressing by additional empirical work. For example, the results presented here may only be relevant for small systems and differ if larger developer populations are considered [35]. Hence, replication studies should be carried out for triangulation purposes.

ACKNOWLEDGEMENTS

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre (www.lero.ie).

We also thank the contribution of our funding and research partners, IBM and Enterprise Ireland.

REFERENCES

1. Bass L, Clements PC, Kazman R. *Software Architecture in Practice*. Addison-Wesley: Boston, MA, U.S.A., 2003; 19–47.
2. Knodel J, Muthig D, Naab M, Lindvall M. Static evaluation of software architectures. *Proceedings of the Conference on Software Maintenance and Reengineering*, Bari, Italy, 2006; 279–294. DOI: 45E7CE08-9DC8-4B88-933B-76B31CF61134.
3. Shaw M, Clements PC. The golden age of software architecture. *IEEE Software* 2006; **23**(2):31–39. DOI: 10.1109/MS.2006.58.
4. Eick SG, Graves TL, Karr AF, Marron JS, Mockus A. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering* 2001; **27**(1):1–12. DOI: 10.1109/32.895984.
5. Hochstein L, Lindvall M. Diagnosing architectural degeneration. *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop*, Greenbelt, Maryland, 2003; 137. DOI: 10.1109/SEW.2003.1270736.
6. Hassan AE, Holt RC. Using development history sticky notes to understand software architecture. *IPWC*, Bari, Italy, 2004; 183. DOI: 10.1109/WPC.2004.1311060.
7. Murphy GC, Notkin D, Sullivan KJ. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering* 2001; **27**(4):364–380. DOI: 10.1109/32.917525.
8. Glass RL. We have lost our way. *Journal of Systems and Software* 1992; **18**(2):111–112. DOI: 10.1016/0164-1212(92)90120-9.
9. Pressman RS. *Software Engineering*: *A Practitioner's Approach*. McGraw-Hill: New York, 2004.
10. Murphy GC, Notkin D, Sullivan KJ. Software reflexion models: Bridging the gap between source and high-level models. *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, Washington, DC, 1995; 18–23. DOI: 10.1145/222124.222136.
11. Tran JB, Godfrey MW, Lee EHS, Holt RC. Architectural repair of open source software. *Proceedings of the 8th International Workshop on Program Comprehension*, Limerick, Ireland, 2000; 48. DOI: 10.1109/WPC.2000.852479.
12. Tvedt RT, Costa P, Lindvall M. Does the code match the design? A process for architecture evaluation. *Proceedings of the IEEE International Conference on Software Maintenance*, Montreal, Canada, 2002; 393. DOI: 10.1109/ICSM.2002.1167796.
13. Tvedt RT, Costa P, Lindvall M. Evaluating software architectures. *Advances in Computers* 2004; **61**:2–43.
14. Sefika M, Sane A, Campbell RH. Monitoring compliance of a software system with its high-level design models. *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, 1996; 387–396.
15. Knodel J, Popescu D. A comparison of static architecture compliance checking approaches. *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture*, Mumbai, India, 2007; 12. DOI: 10.1109/WICSA.2007.1.
16. Yin RK. *Case Study Research*: *Design and Methods*. Sage Publications Inc.: Thousand Oaks, CA, 2003.
17. Torbert WR. The practice of action inquiry. *Handbook of Action Research*: *Participative Inquiry and Practice*. Sage Publications Inc.: Thousand Oaks, CA, 2001; 250–260.
18. Murphy GC, Notkin D. Reengineering with reflexion models: A case study. *IEEE Computer* 1997; **30**(8):29–36. DOI: 10.1109/2.607045.
19. Knodel J, Muthig D, Haury U, Meier G. Architecture compliance checking: Experiences from successful technology transfer to industry. *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, Athens, Greece, 2008; 43–52.
20. Knodel J, Muthig D, Rost D. Constructive architecture compliance checking—An experiment on support by live feedback. *Proceedings of the 24th IEEE International Conference on Software Maintenance*, Beijing, China, 2008; 287–296.
21. Le Gear A, Buckley J. Exercising control over the design of evolving software systems using an inverse application of reflexion modeling. *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, Dublin, Ireland, 2006; 376. DOI: 10.1145/1188966.1189016.

22. Rosik J, Le Gear A, Buckley J, Babar MA. An industrial case study of architecture conformance. *Proceedings of 2nd International Symposium on Empirical Software Engineering and Measurement*, Keiserslautern, Germany, 2008; 80–89. DOI: 10.1145/1414004.1414019.
23. Standish T. Essay on software reuse. *IEEE Transactions on Software Engineering* 1984; **10**(5):494–497.
24. Knodel J, Muthig D, Naab M. Understanding software architectures by visualization—An experiment with graphical elements. *Proceedings of the 13th Working Conference on Reverse Engineering*, Benevento, Italy, 2006; 39–50. DOI: 10.1109/WCRE.2006.54.
25. Seaman CB. The information gathering strategies of software maintainers. *International Conference on Software Maintenance*, Montreal, Canada, 2002; 141. DOI: 10.1109/ICSM.2002.1167761.
26. Singer J, Lethbridge T, Vinson N, Anquetil N. An examination of software engineering work practices. *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, Toronto, Canada, 1997; 21.
27. Godfrey MW, Eric H. Secrets from the Monster: Extracting Mozilla's software architecture. *Proceedings of International Symposium on Constructing Software Engineering Tools*, Limerick, Ireland, 2000.
28. Koschke R, Simon D. Hierarchical reflexion models. *Proceedings of the 10th Working Conference on Reverse Engineering*, Victoria, Canada, 2003.
29. Lindvall M, Tesoriero R, Costa P. Avoiding architectural degeneration: An evaluation process for software architecture. *Proceedings of the 8th International Symposium on Software Metrics*, Ottawa, Canada, 2002; 77. DOI: 10.1109/METRIC.2002.1011328.
30. Christl A, Koschke R, Storey MA. Equipping the reflexion method with automated clustering. *Proceedings 12th Working Conference on Reverse Engineering*, Pittsburgh, U.S.A., 2005; 89–98. DOI: 10.1109/WCRE.2005.17.
31. Le Gear A. Component Reconn-exion, University of Limerick, 2006.
32. Kruchten P. Architectural Blueprints—The 4+1 view model of software architecture. *IEEE Software* 1995; **12**(6):42–50. DOI: 10.1109/52.469759.
33. Frenzel P, Koschke R, Breu A, Angstmann K. Extending the reflexion method for consolidating software variants into product lines. *Proceedings of the 14th Working Conference on Reverse Engineering*, Vancouver, Canada, 2007; 160–169. DOI: 10.1109/WCRE.2007.28.
34. jRM Tool Eclipse Plug-In. Available at: http://jrmtool.sourceforge.net/ [March 2007].
35. Kitchenham B, Pickard L, Pfleeger SL. Case studies for method and tool evaluation. *IEEE Software* 1995; **12**(4):52–62. DOI: 10.1109/52.391832.
36. Wood M, Daly J, Miller J, Roper M. Multi-method research: An empirical investigation of object-oriented technology. *Journal of Systems and Software* 1999; **48**(1):13–26. DOI: 10.1016/S0164-1212(99)00042-4.
37. Oates BJ. *Researching Information Systems and Computing*. Sage Publications Ltd.: Thousand Oaks, 2006.
38. Eclipse Software Development Platform. Available at: http://www.eclipse.org; [March 2007].
39. Krippendorff K. *Content Analysis*: *An Introduction to its Methodology*. Sage Publications Ltd.: Beverley Hills, CA, 2004.
40. Layman L, Williams L, Amant RS. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain, 2007; 176–185. DOI: 10.1109/ESEM.2007.82.
41. Gwet K. *Handbook of Inter-rater Reliability*. STATAXIS Publishing Company: Gaithersburg, MD, 2001.
42. Exton C, Avram G, Buckley J, Le Gear A. An experimental report on the limitations of experimentation as a means of empirical investigation. *Proceedings of the 19th Annual Psychology of Programming Interest Group Conference*, Joensuu, Finland, 2007; 173–284.
43. O'Brien MP, Buckley J, Exton C. Empirically studying software practitioners: Bridging the gap between theory and practice. *Proceedings of the 21st IEEE International Conference on Software Maintenance*, Budapest, Hungary, 2005; 433–442. DOI: 10.1109/ICSM.2005.44.
44. Perry D, Porter A, Votta L. A primer on empirical studies. *Tutorial Presented at the International Conference on Software Maintenance*, Boston, U.S.A., 1997; 657.
45. Adair J. The Hawthorne effect: A reconsideration of the methodological artifact. *Journal of Applied Psychology* 1984; **69**(2):334–345. DOI: 10.1037/0021-9010.69.2.334.
46. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns*: *Elements of Reusable Object-oriented Software*. Addison-Wesley, Longman Publishing Co. Inc.: Boston, MA, U.S.A., 1995.
47. Kolb R, John I, Knodel J, Muthig D, Haury U, Meier G. Experiences with product line development of embeded systems at Testo AG. *Proceedings of the 10th International Software Product Line Conference*, Baltimore, U.S.A., 2006; 172–181. DOI: 10.1109/SPLINE.2006.1691589.
48. Pohl K, Boeckle G, van der Linden F. *Software Product Line Engineering*: *Foundations*, *Principles and Techniques*. Springer: Secaucus, NJ, U.S.A., 2005.
49. Eichberg M, Kloppenburg S, Klose K, Mezini M. Defining and continuous checking of structural program dependencies. *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, 2008; 391–400. DOI: 10.1145/1368088.1368142.
50. Buckley J, Le Gear A, Exton C, Cadogan R, Johnston T, Looby B, Koschke R. Encapsulating targeted component abstractions using software Reflexion Modelling. *Journal of Software Maintenance and Evolution* 2008; **20**(2):107–134. DOI: 10.1002/smr.v20:2.