Design Requirements for an Architecture Consistency Tool

Jacek Rosik
Lero
University of Limerick
jacek.rosik@lero.ie

Jim Buckley
Lero
CSIS Department
University of Limerick
jim.buckley@ul.ie

Muhammad Ali Babar
Lero
University of Limerick
muhammad.alibabar@lero.ie

Abstract

Software architecture and its related documentation are acknowledged as some of the most important artefacts created during system design. However, often the implemented system diverges, over time, from the designed architecture. This phenomenon is called architectural drift and is either a result of inconsistent evolution of the system, or a failure to keep the architectural documentation up to date.

A case study, performed at IBM, over two years showed how architectural drift can occur in small development teams over time. It suggested that even when approaches are in place to identify architectural drift, they may prove insufficient for subsequent removal of the drift, and some possible reasons for this were derived. Consequently, this document outlines the resultant design requirements for an approach to inhibit architectural drift, primarily by identifying it *as, or before*, it is introduced.

1. Introduction

Software Architecture has become an important asset in the design of modern software systems [1, 2]. Design decisions made at the architectural level affect systems' properties, such as, its ability to accept changes and to adapt to changing market requirements [3, 4]. That is, these decisions directly affect system maintenance and evolution, activities which have become a primary focus of software development [5]. Thus, a significant effort is spent on designing architecture to facilitate future evolution.

However all the effort put into this design may be lost, if the implementation diverges from the designed architecture [1]. Such discrepancies between the design and implementation are referred to collectively as *architectural drift* and this is a well documented phenomenon [3, 4, 6, 7]. Architectural drift means that the system's ability to accept change and adapt can be compromised, significantly contributing to the increased cost and difficulties in system maintenance [6, 8]. In addition, architectural drift often results in a situation where the existing architectural documentation is of less use or possibly even harmful for developers [6, 8, 9]. Maintainers can be confused because they are faced with two inconsistent representations of the system (the source code and the architectural documentation) and this may also lead to increased maintenance time and costs [3]. In fact, studies have shown that programmers tend to rely more on the source code because they distrust such design documents [10, 11].

Architectural drift is most often attributed to the side-effects of continuous change [3, 6, 7, 9, 12]; change which is inevitable in the software application's life-cycle [5]. However, the system may also deteriorate in the initial stages of its implementation because the developers are unfamiliar with the design, because they prioritise certain aspects of the system over adherence to the architecture or simply through negligence. This drift is of relatively greater importance, as it may render the designed architecture redundant, even before it is fully realised, wasting all the effort put into its creation.

Several techniques have been designed to analyse and evaluate discrepancies that can arise between design and the implemented system [2, 3, 4, 6, 13, 14, 15, 16]. The evaluations of these techniques are mostly performed as single-session experiments on a finished version of the system and seem to

concentrate on the discovery and evaluation of drift only. That is, they did not study the impact of drift identification on subsequent drift removal.

In earlier work the authors performed a longitudinal, *in-vivo* case study of a commercial software system's re-development using one such technique, to assess its effect on discrepancy removal [17]. It illustrated that identification of discrepancies does not necessarily lead to their removal. This research builds on the findings of this study to present several requirements for a tool which will heighten consistency between the implemented system and the designed architecture. It then discusses the design of this resultant tool.

Section 2 presents some relevant terms from the domain. Section 3 describes the initial approach we employed in the case study, with the aim of reducing architectural drift. This approach emerged from the literature, as the most established and successful architecture evaluation technique to date. Section 4 briefly discusses the *in-vivo* case study that used this technique and subsequent sections discuss the requirements and design that arose as a result of the study's findings.

2. Relevant Terms

Several reports show [3, 9, 15] that, no matter how much effort is spent on designing a system's architecture, it is common that the resulting implementation diverges from that architecture over time [6]. Indeed, a system's implementation can diverge from the intended architecture during the system's initial implementation [17]. This phenomenon is referred to as *architectural drift* [3, 4, 9]. It may be also referred to as *architecture degeneration*, or *system degeneration*[1].

Given the negative implications of such a situation (as described in the introduction), we argue that control should be exercised over the architecture of the system *during the system's development and subsequent maintenance*, with the aim of enforcing consistency: that is, trying to inhibit discrepancies between the current, as-implemented, design and the original, as-intended, design.

Hence, we define *Architecture Consistency* (AC) as the task of assessing and achieving consistency between the *designed architecture* (DA) and the *implemented architecture* (IA) in an on-going fashion over the entire life-span of a system. The designed architecture is represented by design documentation and can also be referred to as *high-level architecture* [12] or *hypothesised architecture* [18]. The as-implemented architecture, also referred to as the *source-code architecture* [12] and *concrete architecture* [18], is represented implicitly, in the implementation.

AC implies that implementation entities (elements of the IA) can be mapped on to design entities (elements of the DA). Elements of the IA which cannot be mapped on elements of the DA are called *architectural violations*. Likewise relationships between elements of the IA (such as invocations) not present in the DA are called architectural violations. Finally, elements of the DA which have no representation in the IA can also be thought of as architectural violations. However, this latter class of violations, if found during development, usually represent (as yet) missing functionality. Hence, these are typically of lesser concern.

AC differs from *architecture evaluation* [13, 15] mainly in that architecture evaluation has been associated with detecting the divergence between DAs and IAs as a one-off exercise, sometime after system deployment [6, 9, 15, 16]. Instead AC, in this work, refers to a more continuous approach during development and evolution, and aims not just to evaluate architectural drift, but to correct it.

3. Reflexion Modelling for AC

In order to be able to exercise control over the design of a software system, a suitable AC technique is required. As mentioned above, several techniques have been proposed in the literature [9, 14, 15, 16] for architecture evaluation. Typically, little explicit detail is given on the individual processes adhered to, but a general pattern does seem to emerge [15]:

---

1 A more precise term would be "non-conformant architecture" but the literature tends to refer to this phenomenon using these three terms.

1. Define the DA and realise it in a supporting tool;
2. Recover the IA from the system's current implementation assets;
3. Compare the two architectures and identify the violations, ideally with tool support;
4. Analyse and verify the violations;
5. Suggest changes to either the IA, the DA or both;
6. Repeat steps 4-6, after these changes have been implemented.

We chose to use Reflexion Modelling as an initial approach to AC, as it is a successful design recovery technique [6, 12, 18, 19, 20, 21] which also closely adheres to the above schema. In addition, it allows the developers to define their own personal perspective on the architecture. An adapted Reflexion Modelling process, for the purpose of limiting architectural drift, was first made explicit by [22]:

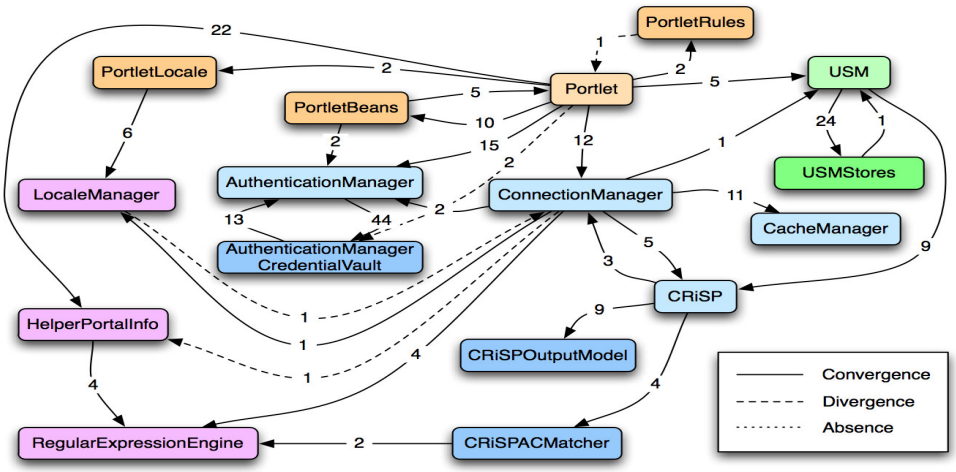

Figure 1: An Example Reflexion Model of the DAP system

1. Before implementation of the system commences, the designer creates a hypothesised architectural model, the *High-Level Model* (HLM) - or DA in the terminology of section 2.1;
2. During the implementation phase, developers and/or architects, update a set of mappings which assign newly implemented source code entities (IA entities) to HLM (DA) entities;
3. At any point during implementation or subsequent maintenance, a dependency graph of the system's sources can be extracted, creating the *Source Model* (SM), - or IA in the terminology of section 2.1;
4. The relationships defined by the engineer in the HLM are compared with those extracted from the implemented system in the SM. Results of that comparison are presented to the developer through the means of a *Reflexion Model* (RM) (see the stylized RM in figure 1). The following relationships are represented in this model:
    - A *solid edge* represents a relationship present in both, the HLM and the SM. (convergence);
    - A *dashed edge* represents a relationship present in the SM, but not present in the HLM. (divergence);

- A *dotted edge* represents a relationship present in the HLM, but not present in the SM. (absence) ;

By analysing particularly the inconsistent relationships in the RM, engineers can choose either to alter the mappings, the HLM, or the SM (the latter through re-factoring the source code).

There is already existing tool support for the Reflexion Modelling process in the form of a tool called the jRMTool [23], a lightweight plug-in for Eclipse that facilitates the creation of the HLM, the mappings and the resultant RM. It also allows users to explore the divergences by listing the source code relationships underpinning unexpected edges. Figure 2 presents a screen-shot of the jRMTool.

4. The Case Study

The principal objective of the case study was to evaluate the selected AC approach (see section 3) in a real-life scenario, over a realistic drift period. That is, the case study focused on determining if violation identification and feedback to developers through this tool would lead to violation removal and, if not, how the current approach could be improved towards this end.

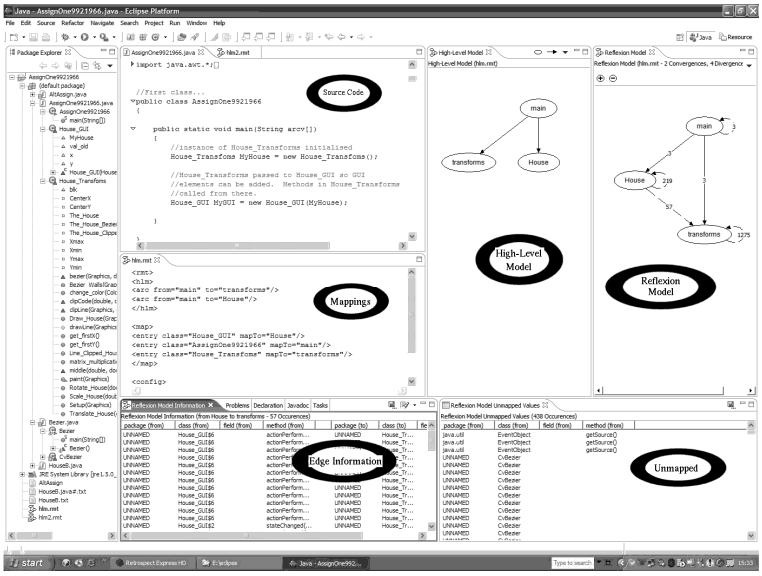

Figure 2: A screenshot of jRMTool, an Eclipse plug-in

Hence a two-year, longitudinal case study was performed in an *in-vivo* industrial setting (at IBM Dublin) [17]. During this period, at approximately four month intervals, three developers, who were re-developing a commercial software system, would perform the evaluation process outlined in section 3.1 using the JRMTool. Through these evaluations feedback was given on several of the architectural violations they had introduced into the system during redevelopment. However, while many architectural violations were identified in this fashion, the approach wasn't as good at achieving architectural consistency. Most of the violations discovered remained until, and beyond, the final session. Indeed, the violations which were removed, were removed as a side-effect of other actions, rather than by explicit developer action aimed at their removal. This suggests that identification of violations periodically is not, of itself, sufficient to ensure architectural consistency. Additional measures have to be employed to ensure violation removal with this approach.

Also in the case study, certain violations were added to the architecture based on specified and valid rationales. After some time however, the rationale for such changes was lost, due to a combination of personnel changes and a lack of annotation facilities in the jRMTool.

Another important issue which arose in the evaluation was that of false negatives. This is where expected edges in the RM represent both expected and unexpected relationships in the code-base. The most graphic example of this in our study was where an edge represented three desired source-code relationships, but 41 undesired source-code relationships. This edge was not explored by the developers because of its 'expected' nature, (even though the RM did show that there were 44 underpinning source-code relationships). Hence, rather than highlighting architectural inconsistencies, the RM served to obfuscate some of them. Murphy et al. [6] acknowledge this failing of the approach implicitly (in an architecture recovery context) when they describe Reflexion Modelling as "approximate".

Other tool-related problems were discovered during the case study. The provisional nature of the supporting tool proved to be problematic, as it sporadically crashed or performed very badly. Additionally some of the models had to be edited by hand using a plain text editor, as opposed to being edited through the supported model editor. Finally, the automatically generated RM differed in topology significantly from the developer-generated HLM, resulting in difficulties comparing the two. 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, [20, 24]. In support of this position, Tvedt et al. [15] claims that poor tool support was one of the limiting factors in the application of their technique (during one of their case studies).

There is no doubt [2, 3, 15, 16] that an AC technique would be a useful addition to existing design and development techniques. However, for such a technique to be successful the problems discovered during this first evaluation have to be addressed.

5. Requirements

The evidence gathered during the case study suggests that the initial "batch-processing" (four-monthly) approach should be refined and integrated more fully into the software implementation process. This section describes the overall approach that the tool should support and the various principles that the tool should embody.

5.1 Approach Overview

Consistent with other techniques used in evaluation contexts [2, 15, 16], it is envisaged that the proposed approach will use *static structural* analysis techniques. Static techniques are those which extract information only from 'static' system assets like source code and documentation. In contrast, dynamic techniques combine this information with the information extracted from the running system. Unfortunately, as the approach will concern itself with systems' initial development (as well as evolution) the approach will not always have access to a running system, and thus will have to rely on static information. In the future however, this could be extended to utilise both static and dynamic information - when dealing exclusively with the consistency of evolving systems (similar to the approach presented by Sefika [16]).

Structural analysis techniques are those which analyse the assets of the project and try to extract structural information about the system: its decomposition into modules and the dependencies / interconnections between these modules. As this information is extracted directly from the system and not based on documentation, it represents the IA.

5.2 Continuous Consistency Checking

In designing the approach for the initial case study, an assumption was made: that periodic evaluations of architecture compliance were sufficient to maintain control over architecture drift [17, 22]. This position was supported by the literature [2, 15]. However, according to other work [25, 26] the longer an issue persists in the source code, the harder it is to fix. Thus discovering the violation at the very moment of introduction should increase the chances of the violation's removal. This is supported in the findings from the case study where most of the discovered violations were considered trivial. This in turn implies that they would have been trivial to fix. Real-time alerts (as proposed by

Eichberg and Knodel [27, 28]) should serve to inhibit these trivially-avoidable violations while also increasing the architectural awareness of developers who sometimes introduce violations because they are not fully aware of the architectural constraints under which they should program [7].

In contrast, when using periodic evaluations, it is likely that the primary focus of development has switched to other parts of the code-base, increasing developers' reluctance to revisit already "closed" code. This argues for real-time notification of architecture violation to developers. Thus, a continuous checking of consistency between the DA and IA is envisaged. This should allow for discovering the violation at the very moment it is introduced. Such an approach should heighten the architectural awareness of the developers as they work on the code, not in retrospect. Consequently, the supporting tool should analyse the source code while it is being written or modified by the developer. When inconsistencies are detected, they should be highlighted to the developer immediately, through warnings or through highlighting of the problematic areas in the source-code editor. Such a process would allow for robust continuous monitoring of the consistency.

An extension of this ideal would be to alert programmers of potential architectural violations before they are introduced. Aids such as IntelliSense (which prompt developers as to the features - functions, data structures - available to them in given contexts) could highlight, or remove those items which would introduce architectural inconsistencies. Thus, such a facility would give programmers prior warning as to the appropriateness of forming relationships between specific methods, classes or packages in advance. However, it is also important that care should be taken to report such potential violations in a non-intrusive manner.

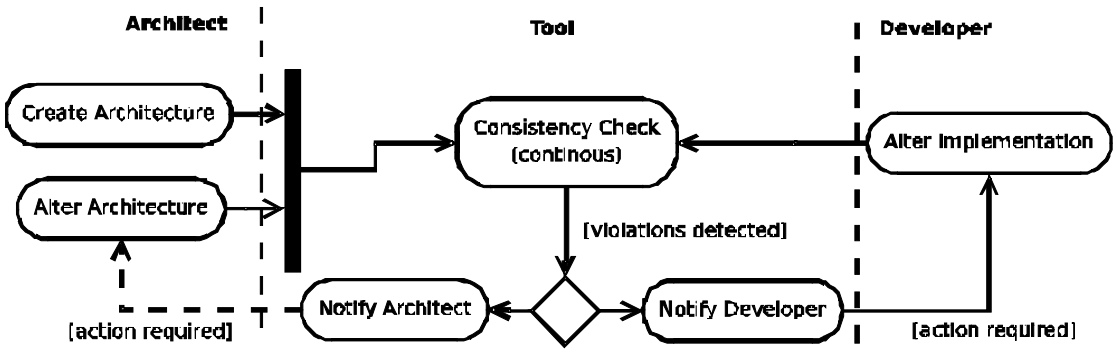

Figure 3: A workflow for AC tools that separates architect and develop roles

5.3 Role Separation and Change Validation

In the initial process there is no means of enforcing the removal of the violations, even if they are detected. When a developer finds an architectural violation he is not forced to remove it, or even comment on it. Indeed, he could 'hide' it in the RM by introducing it into the mapping between the IA and the DA. In three instances this caused architectural misunderstandings within the case-study team. This argues for the ability to annotate edges with rationale and for the separation of architect and developer roles; a separation that was not evidenced in the initial process. A workflow that clearly demonstrates this separation is presented in Figure 3.

However in some projects due to factors like small team size, no clear separation of architects and developers may be feasible. Thus, we envisage that individual team members may have different rights within the approach. For instance a junior developer may only be allowed to modify source code, a senior developer might also be allowed to modify mappings and an architect might additionally be allowed to modify the architecture (DA). Thus, rather than introducing a strict

architect/developer segregation, we envisage that access control could be permitted on an individual team member basis, where each team member may be given appropriate rights. This separation is introduced to prevent a situation where inappropriate team members can change the architecture unilaterally and without notifying the team.

5.4 Refined Architectural Representation

To address the approximation and obfuscation issue we plan to enhance the edge notation used by the initial technique when modelling systems' architectures. This enhancement is envisaged as the ability to introduce multiple edges and multiple categories of edges between the same nodes in the RM, probably on the basis of separating edges that connect different node 'interfaces'. This, in turn, will allow for better representation of real-life systems, while alleviating the problem of false negatives somewhat.

As mentioned in section 5.3, this increased granularity of edges should be connected with edge annotation. Annotating of edges in the model will allow other developers or architects in the team to understand the rationale for introduced edges quickly. If a seeming violation is introduced or allowed to persist in the source code, it should be annotated and clearly distinguished from other edges, so that it can be easily identified. In this way accepted violations wouldn't remain unnoticed when personnel changes occur, as was the case in this study. A similar method to this is described by Hassan and Holt [8] where developers could extract information about an edge from a version control system with the aim of being able to reason about them.

However, as a key factor contributing to the popularity of the technique is its lightweight nature, it is envisaged that the enhancements will be optional and minimal. That is, depending on the project's needs, the architecture can be modelled as a simple box and edges diagram or by using more sophisticated structures with edge sub-typing end explicit interfaces.

6. Design of the AC Tool

6.1 Implementation Overview

The Eclipse platform [29] has been selected as a platform for the implementation of the AC tool. Eclipse is a widely used, open source IDE for software developers and so provides software tool developers with the foundation of a familiar environment for these programmers, a favourable indicator for adoption [30]. But it is also a development platform comprising of extensible application frameworks, tools and a runtime library for add-on software development and management. These allow for easy implementation of additional features (in fact, the jRMTool used in the initial version of the technique is also implemented as an Eclipse plug-in). Additionally the 'continuous compilation' work-flow within Eclipse makes it an ideal candidate for our continuous, consistency-checking approach. Finally, as this IDE is open-source, any changes required in the core of the IDE can be freely applied, as the source code is freely available.

A rich set of generic extensions already exists for Eclipse which is designed as helpers in creating other, more complicated plug-ins. We envisage using at least two such extensions, namely the Graphical Editing Framework (GEF) and the Eclipse Modelling Framework (EMF). The GEF allows developers to create a rich graphical editor from an existing application model. The developer can take advantage of the many common operations provided in GEF and/or extend them for the specific domain. GEF employs a Model-View-Controller architecture which enables simple changes to be applied to the model from the view.

The EMF is an Eclipse-based modelling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XML, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

Models can be specified using annotated documents, or modelling tools, and then imported into the EMF.

The main components of Eclipse are implemented using the Java programming language and this enforces the implementation language of the AC tool. Even though the Eclipse IDE provides support for development using languages other than Java, like C++ and Python, the first version of the tool will only support Java as a target language for AC control. This is primarily a scoping consideration but Java was selected due to its widespread use and its interpreted nature, which makes it an ideal candidate for static analysis.

6.2 Integrating the Tool

The initial technique (employed in the case study [17]) used a separate view (the RM) to convey architectural information and warnings. This requires the developer to switch his attention from the main development view (source code) which involves undesirable overhead [25]. In addition, although a separate view which is specifically designed for such a task may convey architectural information better, it will be useless if a developer chooses not to disrupt his main development task to view it. Thus, along with this separate, holistic view it is planned to adopt two existing features of Eclipse already used by developers.

The first such feature is compiler warnings. Modern IDEs display compiler warnings in a window below the source code and also highlight problematic source code sections in the source code editor itself. Developers typically react to such warnings, as they indicate problems in the code. By introducing the notion of architectural *warnings,* we plan to reuse this feature (and the associated views). As the source is compiled, it will be also analysed for potential architectural problems and architectural warnings will be presented in the same manner as the regular compiler warnings.

The Eclipse IDE uses a notion of *incremental builders* to provide continuous compilation of the source code. When a developer writes a line of code, the affected code is recompiled and possible problems and errors are communicated to the developer. This incremental compilation uses delta computation to detect affected source code areas and compiles only modified fragments of code reusing already compiled chunks of unaffected code from previous compilations. This results in fast compilation time and real-time feedback. As this functionality is extensible, this provides the capability for real-time architectural warnings.

Another feature we plan to reuse is IntelliSense. This feature prompts the developer with the available methods and data structures that are available in a given, source code context. If real-time static analysis information is available and coupled to the RM, then the IntelliSense list can be filtered to remove violating code or, can highlight it in different colours. It is envisaged that this will prompt architectural awareness in the developers and discourage the introduction of architecture violations.

The reuse of these features should allow for an increased level of architectural control over the implementation of the software system in real-time while, at the same time, minimising familiarisation overhead, as analogue functionality is already commonly used by developers who use the Eclipse IDE.

6.3 Essential User Interface Components

The user interface of the proposed plug-in is composed of a set of interacting views. Through these views, users can interact with the underlying model and also receive architectural warnings. Some of the views will be new additions to the Eclipse IDE while other, standard views will be reused, and only adapted for the task of AC control.

The Architectural Editor / View is a combination of two views (models) from the jRMTool: the *High-Level Model* and the *Reflexion Model* (which is depicted in figure 1). In the original Reflexion Modelling technique, the High-Level Model is used to create and edit the DA and the Reflexion Model displays the results of the analysis in read-only mode. However, this Architectural Editor will

be used for both tasks, as this enforces consistency between the typology of the two models, facilitating comparison.

Mappings between the DA and IA can be created manually in a text editor but, the main mode of operation should be a drag-and-drop interface where elements of the IA are dropped into the DA elements. Eclipse already supports drag and drop for Java elements. Thus, this view only needs to be adapted to accommodate this functionality.

The Source Editor in Eclipse is used to edit source code. This view already has a feature that displays compilation errors and warnings by underlining the affected line of code and providing a pop-up explanation of the problem. This is shown in figure 4. This notification is connected with the **Problems View** where a full description of the error is available (see figure 5). These features of the IDE will be reused to show architectural errors. When a programmer accesses a source code entity which is architecturally forbidden in the current context, the offending line of code should be highlighted and the appropriate errors displayed in the pop-up explanation and the Problems View. In this way, developers do not have to switch attention to a separate view to check for architectural violations as they develop the system.

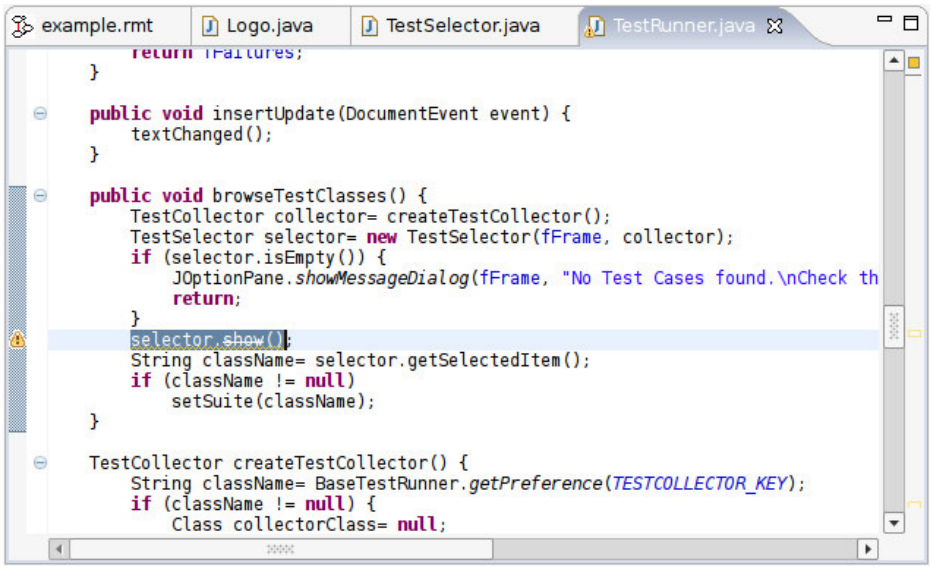

Figure 4: The Eclipse Source Editor showing a Compilation Warning

The Outline View is another, existing view of Eclipse. This view shows the hierarchical decomposition of Classes: their member variables and methods in the form of a tree. The contents of this view change with the file being edited. That is, when a user edits a file, the contents of this file are shown in the view.

When the developer switches and starts to edit a different file, the view will display the contents of that other file. Thus, when a developer edits the architecture, architectural elements should be displayed in this view, giving an outline of the area of code being worked on at that given moment. These elements will include nodes, node interconnections and Java elements mapped to given nodes.

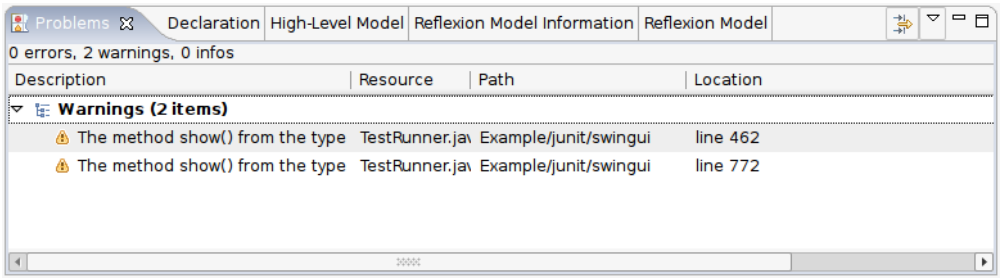

Figure 5: The Associated Problem View

7: Envisaged Execution of the approach

7.1 Continuous Consistency Checks

Our approach is envisaged as two-pronged: continuous consistency checks during the system's development and maintenance and commit-based consistency checks. To achieve the best results it is envisaged that the continuous consistency checks will be tightly integrated with normal development practice, ideally becoming a part of it. The consistency checks are performed in the background, concurrent with the main development task. Thus checking does not require any explicit action on the part of the software engineer. The control flow of a generic consistency checking approach, where a pre-defined DA exists, is shown in figure 6.

When a software engineer writes a line of code the new IA entities and dependencies will be extracted from the modified code base. These will be compared with the previous set of dependencies and entities, as not all changes to the source code will introduce/remove entities and dependencies. If a different set of entities and dependencies is detected, and/or the engineer alters the mappings, the IA model will need to be re-computed. This model will then be compared with the current DA and the results presented to the developer.

Note that, even though certain modifications may be prohibited for given user rights, no check as to whether that person is allowed to apply given changes is performed at this stage. This is to prevent excessive interruptions to the development process and also to allow for experimentation on the local copy. A final check will be performed when changes are committed to a central code repository which is described in section 7.2.

The results of the consistency check are presented to the developer in several ways, as discussed earlier:

- **The Architecture View** – this view is updated to show the current state of the architecture and also to highlight any discrepancies between DA and IA, on-demand;
- **The Source-Code View** – this view provides non-prompted, immediate feedback;
- **Problems View** – likewise, this view will provide non-prompted, immediate feedback;

An additional, helpful view during the development may be that of IntelliSense. Based on the internal IA and DA models, a list of available choices in IntelliSense pop-ups may be filtered or colour-coded to highlight only architecturally correct choices. This way, architectural awareness is heightened in advance of decisions and premature commitment is avoided [31].

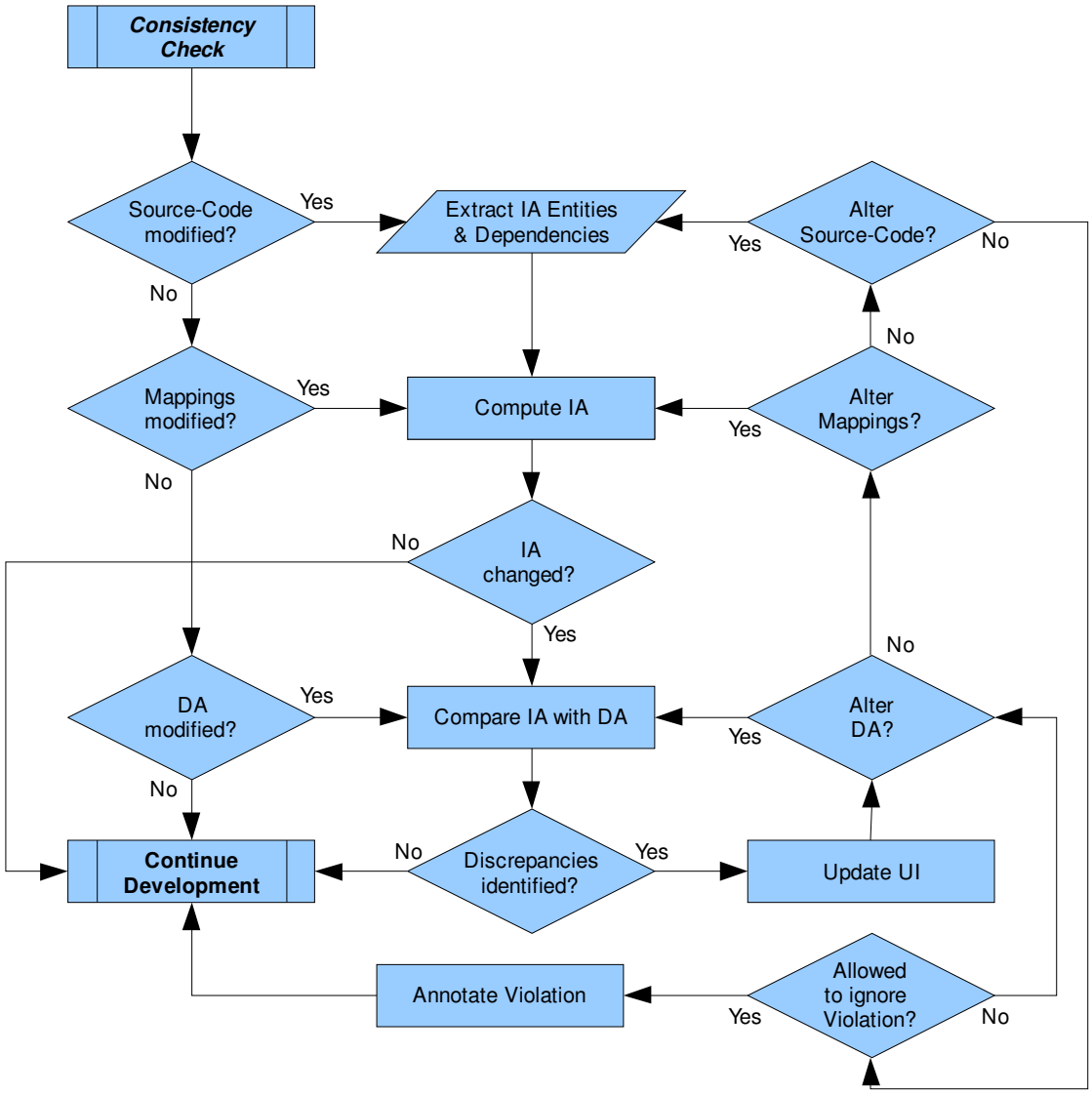

Figure 6: Control flow for envisaged continuous checking

When the developers get this feedback, they can choose to ignore the violation (defer the fix) and finish their current task or try to fix the violation immediately. If they choose to act on the violation, one of the following situations may apply:

- **They may Alter the Source-Code** – here the developer attempts to remove the offending code or re-factor it. This will alter the IA which will trigger another consistency check. When the new check confirms that the architecture is consistent, the development can continue;
- **They may Alter the Mappings** – another cause of violation may be that certain IA entities have been mapped to the wrong DA entities. By altering the mappings the architecture may be made consistent again. Not all developers may be allowed to alter the mappings and commit their changes without the validation of a system architect;
- **They may Alter the Architecture** – during the development, further implementation-driven insights into the design may occur. These may in turn call for a change to the architecture. Thus, one of the ways of fixing a violation is to alter the DA to accommodate these insights.

As with the mappings, only selected developers should be allowed to modify and commit the violations without an architect's consent;

- **They may Annotate the Violation** – sometimes, a developer may wish to preserve a violation for pragmatic reasons like performance. Such violations have to be annotated to preserve the rationale for allowing them to persist.

This concludes the "non-commit phase" of the approach. As mentioned before, during this phase no checks are performed to validate whether the developer is allowed to make certain changes. These changes are allowed in the user's working copy of the project. However, ideally these should be resolved and validated before the developer commits the code to the central repository.

7.2 Committing the Changes

When a developer decides to share their changes with the other team members, their code base should be free of violations. In addition, if that developer is not allowed to change mappings or the DA, no such changes should be allowed. Thus, another part of the approach has to be integrated with a version control system and consistency checks should be performed before the commit is allowed. The commit will only be allowed if the code base is free of any offending changes (see fig 7).

Apart from the conventional consistency check, run before allowing a commit to proceed, several additional checks have to be performed. If the consistency check or any of the additional checks fail, the commit will be cancelled and the developer will have to resolve the problems before attempting another commit.

If any violations persist when the developer attempts to commit the source code, provided that violations are allowed (which in turn depends on the project's configuration) all of these violations would need to be annotated. It is forbidden to commit an un-annotated violation, as they may confuse other team members and trigger more serious problems in the future [17].

Then, if the developer has applied any changes to the mappings or the architecture, a check is performed to validate their authority to do so. If such a check fails, the commit will be aborted and the developer should talk to the person allowed to introduce such changes; usually the team's architect. Otherwise such changes will not be committed. Even if the developer is allowed to commit such changes, they will not become effective before they are validated by the architect. An architect can validate such changes and re-commit them to the repository. Then these changes will become available to other team members.

8. Conclusion

This paper has presented the empirically derived requirements for an architecture consistency tool. These include continuous architecture consistency monitoring, non-disruptive alerts, user-role segregation and optional edge sub-typing. It builds on these requirements to present a proposed design for the tool, which aims to minimise cognitive disruption and premature commitment. But these proposals require further evaluation from the cognitive community. Evaluative issues include:

- what, if any, is the cognitive overhead in switching from developing with a source-code view, to evaluating an architecture through a separate graphical view?
- does the claim that 'advance indications of architectural violations' are desirable over 'retrospective error alerts' stand up to cognitive scrutiny?
- are there any other cognitive issues of which we should be aware in our proposed design?

In presenting our work at the PPIG forum we hope to get feedback on these questions.

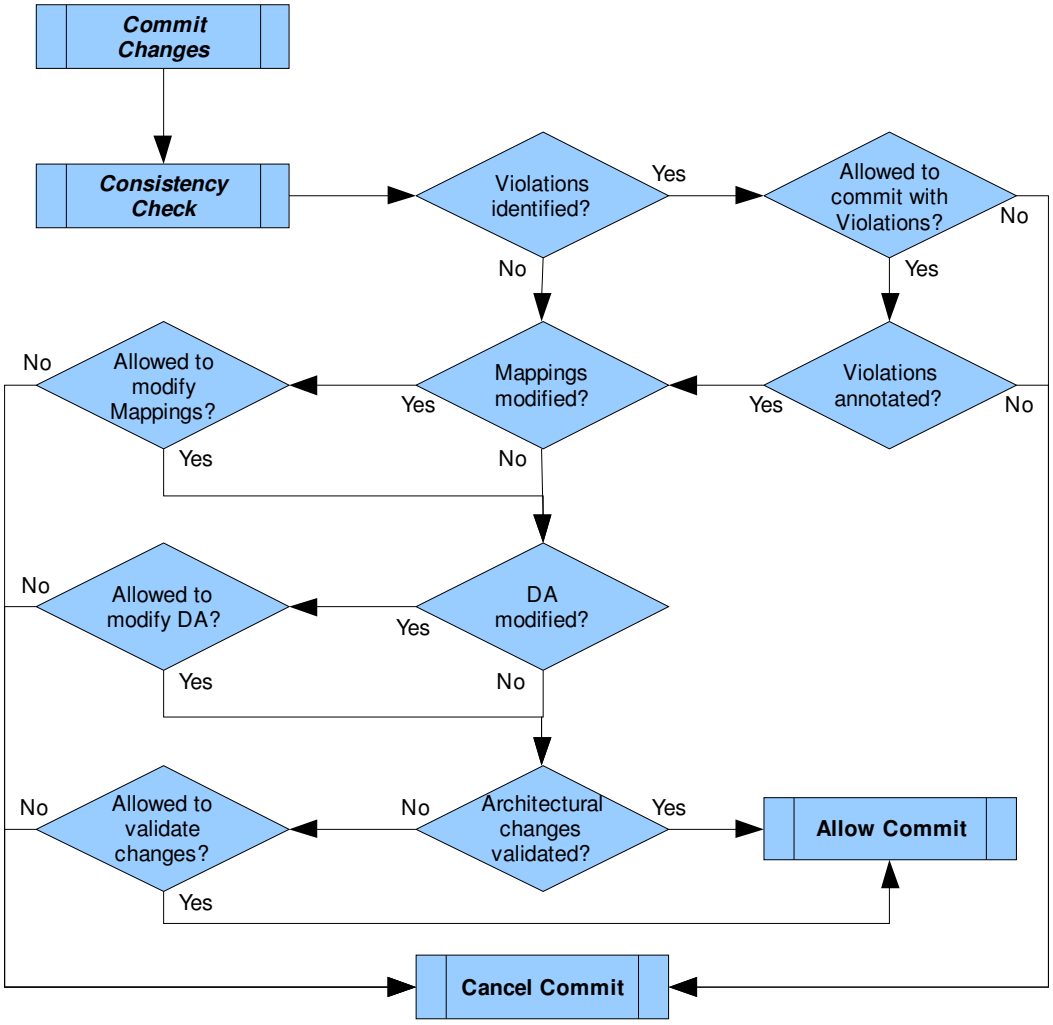

Figure 7: Envisaged Control Flow for Commits

Acknowledgement

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

References

[1] M. Shaw and P. C. Clements. The golden age of software architecture. *IEEE Software*, 23(2):31–39, 2006.

[2] J. Knodel, D. Muthig, and M. Naab. Understanding software architectures by visualization–an experiment with graphical elements. *Proceedings of the 13th Working Conference on Reverse Engineering*, 39–50, 2006.

[3] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering,* 27(1):1–12, 2001.

[4] L. Hochstein and M. Lindvall. Diagnosing architectural degeneration. *Proceedings of the SEW*, 137-142, 2003.

[5] R. L. Glass. We have lost our way? *Journal of System Software*, 18(2):111–112, 1992.

[6] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. On Software Engineering*, 27(4):364–380, 2001.

[7] R. T. Tvedt, P. Costa, and M. Lindvall. Does the code match the design? A process for architecture evaluation. *Proceedings of ICSM*, 393-401, 2002.

[8] A. E. Hassan and R. C. Holt. Using development history sticky notes to understand software architecture. *Proceedings of IWPC*, 183-192, 2004.

[9] J. B. Tran, M. W. Godfrey, E. H. S. Lee, and R. C. Holt. Architectural repair of open source software. *Proceedings of the 8th International Workshop on Program Comprehension*, 48-57, 2000.

[10] C.B. Seamen. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions of Software Engineering.* (25) 4. 557-572, 1999.

[11] J. Singer J., T. Lethbridge, N. Vinson and N. Anquetil. An Examination of Software Engineering Work Practices. *In Proceedings of CASCON,* 209-223, 1997

[12] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In SIGSOFT, 18–28, 1995.

[13] J. Knodel, D. Muthig, M. Naab, and M. Lindvall. Static evaluation of software architectures. *Proceedings of the Conference on Software Maintenance and Reengineering*, 279–294, 2006.

[14] M. Lindvall, R. Tesoriero, and P. Costa. Avoiding architectural degeneration: An evaluation process for software architecture. *Proceedings of the 8th International Symposium on Software Metrics*, 77-86, 2002.

[15] R. T. Tvedt, P. Costa, and M. Lindvall. Evaluating Software Architectures. *Architectural Issues*, 61:2–43, 2004.

[16] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. *Proceedings of the 18th International Conference on Software Engineering*, 387–396, 1996.

[17] J. Rosik, A. Le Gear, J. Buckley and M. Ali Babar. An Industrial Case Study of Architecture Conformance. *Proceedings of ESEM,* 80-89, 2008

[18] R. Koschke and D. Simon. Hierarchical reflexion models. *Proceedings of WCRE* 36-45, 2003.

[19] A. Christl, R. Koschke, and M. A. Storey. Equipping the reflexion method with automated clustering. *Proceedings of WCRE*, pages 89–98, 2005.

[20] A. Le Gear. *Component Reconn-exion*. PhD thesis, University of Limerick, 2006.

[21] G. C. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *Computer*, 30(8):29–36, 1997.

[22] A. Le Gear and J. Buckley. Exercising control over the design of evolving software systems using an inverse application of reflexion modeling. In *CASCON '06: Proceedings of the 2006 Conference for the Centre for Advanced Studies on Collaborative research*, 37-45 2006

[23] jRMTool eclipse plug-in. http://jrmtool.sourceforge.net/, 2008.

[24] C. Exton, G. Avram, J. Buckley, and A. Le Gear. An experimental report on the limitations of experimentation as a means of empirical investigation. *Proceedings of PPIG*, 173-184, 2007.

[25] L. Layman, L. Williams, and R. S. Amant. Toward reducing fault fix time: Understanding developer behaviour for the design of automated fault detection tools. *Proceedings of ESEM* 176–185, 2007.

[26] M.A. Storey. Source code comments: Graffiti or Information? (Keynote) PPIG 2008.

[27] M. Eichberg, S. Kloppenburg, K. Klose and M. Mezini. Defining and continuous checking of structural program dependencies. *Proceedings of ICSE,* 391-400, 2008.

[28] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. *Proceedings of WICSA*, 2007.

[29] Eclipse. http://www.eclipse.org, Jan. 2008.

[30] J. Buckley. Requirements-Based Visualization Tools for Software Maintenance and Evolution. IEEE Computer, April 100-102. 2009.

[53] T.R.G. Green. Cognitive dimensions of notations. I*n People and Computers V, A Sutcliffe and L Macaulay (Ed.) Cambridge University Press* 443-460, 1989.

[32] J. Buckley, A. Le Gear, C. Exton, R. Cadogan, T. Johnston, B. Looby, and R. Koschke. Encapsulating targeted component abstractions using software reflexion modelling. J. Softw. Maint. Evol., 20(2):107–134, 2008.