Research

Encapsulating Targeted Component Abstractions Using Software Reflexion Modelling

Jim Buckley[1*A], Andrew LeGear[1*B], Chris Exton[1], Ross Cadogan[2], Trevor Johnston[2], Bill Looby[2], Rainer Koschke[3]

[1]*Department of Computer Science & Information Systems, University of Limerick, Ireland*
[2]*IBM, Dublin, Ireland*
[3]*University of Bremen, Germany*

SUMMARY

Design abstractions such as components, modules, subsystems or packages are often not made explicit in the implementation of legacy systems. Indeed, often the abstractions that are made explicit turn out to be inappropriate for future evolution agendas. This can make the maintenance, evolution and refactoring of these systems difficult. In this publication we carry out a fine-grained evaluation of Reflexion Modelling as a technique for encapsulating user-targeted components. This process is a prelude to component recovery, reuse and refactoring. The evaluation takes the form of two in-vivo case studies, where two professional software developers encapsulate components in a large, commercial software system. The studies demonstrate the validity of this approach and offer several best-use guidelines. Specifically, they argue that users benefit from having a strong mental model of the system in advance of Reflexion Modelling, even if that model is flawed, and that users should expend effort exploring the expected relationships present in Reflexion Models. Copyright © John Wiley & Sons, Ltd.

J. Softw. Maint:

No. of Figures: 7. No. of Tables: 1. No. of References: 71.

KEY WORDS: Software Maintenance, Reengineering, Architecture Recovery, Component Recovery, Reflexion,

*Correspondence to: Jim Buckley, Department of Computer Science & Information Systems, University of Limerick, Ireland: or to: Andrew LeGear, Department of Computer Science & Information Systems, University of Limerick, Ireland:
[A]E-mail: Jim.Buckley@ul.ie
[B]E-mail: Andrew.LeGear@ul.ie

Grant Sponsors: QAD Ireland, IBM Centre for Advanced Studies Ireland, Enterprise Ireland.

1. INTRODUCTION

Just as object-oriented programming facilitated the maintenance and reuse of software at its level of granularity [71], the principles of component-based development have been championed as a means of promoting the same qualities at yet higher levels of abstraction [61], [7]. Unfortunately, programming languages that provide first class constructs for software components were not used in the implementation of the majority of the world's existing applications. As a result, software designers and architects have often conceptually designed software systems in terms of large components, even though the subsequent implementation could not make these components explicit. The problem in such cases is to make explicit these implicitly-implemented component abstractions for the purposes of eased maintenance, component recovery, refactoring or reuse [1], [29], [7]. Indeed, given that anticipating software evolution is a difficult task [27], it is likely that, even if the original design was made explicit, it would be inappropriate or inadequate for all of the system's subsequent evolutions.

This paper examines the problem of making component abstractions, as targeted by the designer or maintainer of non-component-based systems, explicit using a small variation of the software Reflexion Modelling technique. We evaluate the technique's usefulness using two large, industrial case studies.

The remainder of this publication is structured as follows. First we explore related work and position our approach within the state of the art. This is followed by a detailed description of the existing software Reflexion Modelling technique and its underlying cognitive basis. To ground our evaluation, we explicitly describe our instantiation of the technique for targeted component encapsulation. Next we carry out a fine-grained evaluation of this Reflexion Modelling process using two large, industrial case studies in an ecologically valid setting. Finally we discuss our findings and describe future directions for this work, based on this evaluation.

2. RELATED WORK

Closely related to the process of component encapsulation described in this publication is componentisation, which is a type of software restructuring technique [8]. Componentisation is the process of taking an existing software system that is not component-based and augmenting ('wrapping') it to conform to the principles of component-based development using explicit language constructs [52], [24], [10]. The goal of componentisation is, typically, modernisation of the system to conform to a new development paradigm and little consideration is given to the agendas of maintenance, component reuse or component recovery per-se. That is, the emphasis is placed on wrapping modules once identified and not on the means for identifying them, as explored in this publication.

Johnson in [24] describes the necessary requirements for a toolkit architecture to support the componentisation of software systems and bases a proposed process upon black-box reengineering. Black-box reengineering is any reengineering approach that only requires the maintainer to understand the system down to the functionality level and not the detail of implementation. Johnson, however, neither creates the proposed tool kit nor evaluates his proposed process. Also this componentization work differs from the subject matter of this paper in that it does not address the user-targeted recovery of a specific component for a specific agenda.

Instead, our proposed technique can be positioned in the research field of partial *design recovery* [4], a term which covers research regarding the reconstitution of existing software systems' macro-designs. A plethora of work in this field focuses on component identification and recovery [15], [37], [41], [36], [20], [44], [64], [69], [26], [39], [10], dating back to as early as 1985 [23]. The most comprehensive review of the subfield is found in [29], where existing recovery techniques are categorized into four groups:

1. Dataflow-based approaches - The movement and use of data is examined to cluster software elements together. For example, in [21], abstract data types are identified in systems by grouping global variables and the procedures that use them together.
2. Structure-based approaches - These approaches examine the structure (usually the call tree) of the subject system and look for relationships in this structure. The cardinality and strength of these relationships are used to cluster the related entities. Koschke [29] subdivides these into Connection-based approaches [9], Metric-based approaches [50], Graph-theoretic-based approaches [13] and Concept-based approaches [36].
3. AI-based approaches - These approaches exploit techniques from the artificial intelligence community to recover components from existing systems. For example, in [15], the authors use a genetic algorithm in an attempt to automatically recover the architecture of a system.
4. Domain-based techniques - These techniques exploit existing knowledge regarding the application domain, during the recovery of design artefacts. By application domain, we refer to the context of the problem that is addressed by a piece of software [56]. These approaches tend to be semi-automatic, requiring significant user involvement in the process to capture the domain knowledge. This is in line with a conclusion from Koschke's review that future techniques for component recovery should include the human more in the process [29]. The Reflexion Modelling technique, which serves as the foundation of the approach proposed by this paper, may be placed in this category [38].

The Reflexion Modelling technique has primarily been used for software comprehension [38], [30] and is described in detail in the following section. Other approaches similar to software Reflexion Modelling include the concern mapping approaches of FEAT [48] and CME [12] and architectural conformance techniques like PatternLint [55] and Hedgehog [6]. These approaches allow software developers to partition source code into (macro) categories of interest. These categories can reflect a mapping to the application domain but alternatively, the partitioning may reflect any specific tasks software engineers face when working on their software systems. In the case of Reflexion Modelling, the categorization is presented back to the user as an informal architectural diagram of the system.

We are aware of only one paper that describes Reflexion Modeling's use in the extraction of specific, user-targeted components [39] (with [38] alluding to this study in summary). In this paper, there is little explicit description of the Reflexion Modelling process used by the participant, little evaluation as to how supportive the process was and no objective assessment of the resultant components. This lack of empirical evaluation is unfortunate because the preliminary evaluation provided is promising and because user-targeted component isolation would seem to be a common maintenance scenario. For example, in porting legacy systems to a new platform, all system calls or elements of the GUI layer may need to be isolated. Likewise, in other scenarios, specific end-user functionality may need to be isolated for omission from or for reuse in future products (when moving a set of existing systems to a Software Product Line for example).

This paper aims to address this concern by proposing an explicit description of Reflexion Modelling, as applied to targeted component extraction, and by providing a fine-grained evaluation of the process and the resultant product.

3. REFLEXION MODELLING

The following sections describe the principles of software Reflexion Modelling, and briefly discusses previous evaluations of the technique. Subsequently, we explicitly describe our variation on Reflexion Modelling for the purpose of encapsulating a user-specified component from a system.

3.1 The Reflexion Modelling Process

Software Reflexion Modelling is a semi-automated, diagram-based, structural summarisation technique. Introduced by Murphy et al. [38], the technique is primarily aimed towards aiding software understanding. Software Reflexion Modelling follows a six step process, illustrated in Figure 1:

1. The software engineer who wishes to understand the subject system hypothesises a *high-level* conceptual model (HLM) of the system.
2. The computer extracts, using a program analysis tool, a dependency graph of the subject system's source code called the *source model*.
3. The software engineer then creates a map which maps the elements of the source model onto individual nodes of the high-level model.
4. The computer then assesses the call relationships and data accesses in the source code to generate its own high-level model (called the Reflexion Model). This model shows the relationships between the source code elements mapped to different nodes in the software engineer's high-level model. This allows comparisons between the computer's model and the software engineer's model and the tool can report this comparison in three ways:
    - A dashed edge in the Reflexion Model represents dependencies between elements of the software engineer's high-level model that do exist in the source model, but were not placed in the software engineer's high-level model.
    - A dotted edge in the Reflexion Model represents a hypothesised dependency edge of the software engineer's high-level model that does not actually exist in the source model.
    - A solid edge in the Reflexion Model represents a hypothesised edge of the software engineer's high-level model that was validated by the source model.
5. By targeting and studying the inconsistencies highlighted by the Reflexion Model the software engineer can either alter their hypothesised map or the high-level model to produce a better recovered model of the system.
6. The previous two steps are repeated until the software engineer is satisfied that the recovered model is consistent with their high-level model.

Figure 1, here please

Figure 1 –The Software Reflexion Modelling Process

Further expansions on the core Reflexion Modelling technique include the creation of hierarchical Reflexion Models [30], the use of automated clustering techniques to augment the creation of the map [11] and the use of source-code-control derived sticky-notes to inform users of the rationale and software engineer responsible for the introduction of unexpected relationships in the Reflexion Model [22].

Reflexion Modelling differs from previously proposed recovery techniques, in its user-prompted partitioning of the system and its on-going, intensive user involvement, thus adhering to best practice as espoused by [29]. Indeed, Reflexion Modelling is also strongly supported by educational psychology literature. Piaget [46], for example, postulated that cognitive conflict, a situation where a person is discontented with their current understanding (or mental model) as it no longer fits what they observe, is a driving force for learning and development. Likewise, Strike and Posner [59] contend that cognitive conflict is required for conceptual change to occur in the mind of a student. It is this dissatisfaction (or cognitive conflict) that motivates the student to consider alternative conceptual views that may result in a number of amendments or expansions to their understanding.

In Reflexion Modelling, cognitive conflict is embodied in the inconsistencies between the software engineers' stated HLM and the de-facto, parsed Reflexion Model. This cognitive conflict is a positive force, as it provides an impetus for the software engineer to either correct their understanding of the system or to adapt the system to comply with their own understanding, in order to re-achieve a state of equilibrium.

An opposing perspective is provided by 'Fit-to-Theory' [51] literature (also known as 'cherry-picking'). This literature suggests that people tend to pay attention to evidence that confirms their theories and disregard evidence that contradicts their theories. However, as Reflexion Modeling forces software engineers to explicitly state their model of the system in advance, and the subsequent Reflexion Model emphasizes the differences, it would seem difficult for the engineers to ignore the inconsistencies that arise. Thus the Reflexion Modeling scheme would seem to lessen the impact of 'fit-to-theory'.

Reflexion Modeling can also be couched in terms of 'Premature Commitment', part of Green's well-established Cognitive Dimension [5] framework for evaluating notations. Traditionally 'Premature Commitment' was seen as affording lesser utility in notations as it refers to the degree to which a notation forces its user to prematurely embody an assumption (in that notation) and possibly have to revise it later. In this instance, premature commitment presents itself in the form of forcing software engineers to explicitly state their system model, before this model is (possibly) contradicted by the Reflexion Model. This contradiction may force the iterative reformulation of the software engineer's initial model and thus, subsequent reformulation of their HLM should not be difficult (or 'viscous' in Cognitive Dimension vocabulary [5]). As Reflexion Modeling allows for the easy reformulation of the software engineer's HLM, 'Premature Commitment' seems to be a positive attribute of this modeling approach.

3.2 Evaluating Reflexion Modelling

Reflexion Modelling is accompanied by promising results when used to understand large software systems [39], [38], [30], [35]. For example, in two experiments, detailed in [30], users are described as gaining an encompassing understanding of 100KLOC and 500KLOC compilers in 6 and 8 hours respectively.

An evaluation more closely related to the work described here appeared in [39]. Here Reflexion Modelling allowed a software engineer to state that he gained a level of understanding of Microsoft Excel^{TM} (1 million KLOC+) within one month that would normally have taken two years. This was achieved in the context of extracting components from the Excel code-base for re-use. In this instance, the authors found that the approach was useful and they commented on several characteristics of the software engineer's behaviour in terms of his creating a detailed mapping up front, and studying both expected and unexpected edges. Surprisingly, the paper suggests that a strong mental model of the domain or system is not an important pre-requisite for users of Reflexion Modelling, in apparent contradiction of the psychology literature and [29]'s recommendation for the inclusion of users' domain knowledge. Additionally, they make no mention of identifying interfaces in this evaluation, a core element of resultant components [7]. Finally, it is unclear if the Reflexion Modelling resulted in extracted components or just documentation towards that goal. If components were extracted, the evaluation of the extracted components was not presented. These issues are addressed in this paper.

They are addressed by means of a qualitative [54] empirical study where verbal and textual data produced by software engineers is assessed in an open manner. That is, the findings are driven by the data in a more bottom-up, interpretive fashion [40]. Such a protocol has been used many times in the study of software engineers, notable examples including Ko et al. [28], Seaman and Basili [53], O Brien [42] and Von Mayrhauser and Vans [66]. The data produced here is used to assess if the Reflexion Modelling process is perceived as useful by software engineer practitioners and if so, to derive guidelines for best usage of the approach.

3.3 Reflexion Modelling for Targeted Component Encapsulation

Due to the successes of software Reflexion Modelling as a means of partitioning a system into higher level abstractions, the authors argue that software Reflexion Modelling is suitable for unambiguously encapsulating targeted components of existing systems and can aid in the definition of their interfaces [34]. Here, the term 'component interface' refers to the declaration 'of a set of behaviours that can be offered by a Component Object' [7]. This behavioural declaration usually specifies the required parameters and the returning values [60]. Encapsulating the targeted components suggests a small variation on the initial Reflexion Modelling process as follows:

1. The software engineer targets a component by creating a high-level model that contains only two nodes:
   - A high-level node representing a first attempt at the component that the software engineer wishes to encapsulate.
   - A second high-level node that represents the remainder of the system (see Figure 2).

   The selective targeting of one specific component in this way allows the software engineer to map the relevant software elements to this component in their model and thus assess its completeness, and its interactions with the rest of the system.
2. The software engineer maps the appropriate elements of the software system to one of the two nodes in the high-level model
3. They then iterate through the traditional software Reflexion Modelling process, allowing the tool to build the Reflexion Model and the software engineer to study the edges between the two nodes for expected and unexpected dependencies, as illustrated in Figure 2. At this stage, it is less likely that unexpected dependencies will exist to confront the software engineer with (although it is still possible). Rather the discovered edges serve to identify source code elements that are inconsistently placed in this High-Level model. Refinements at this stage are limited to changing the map, to better reflect the component's desired contents. This process iterates until the software engineer is reasonable confident that the component has been encapsulated.
4. The edge incident on the component node (as in Figure 2) identifies an approximation of the interface of the desired component. It is an approximation in that it represents *all* the calls that the remainder of the system currently makes on the component, but the component may offer other behaviours that the system does not currently use, and these will not appear in the edge. Also, in Reflexion Modelling the edges include data-access relationships between nodes, accesses which would not traditionally be considered part of a component's interface[7], [60].
5. The software engineer then proceeds to divide up the rest of the system into its major constituent parts. These divisions will be neighbours of the component being encapsulated, allowing the software engineer to finesse the contents of the component, and guiding them towards the individual interfaces of the component. By 'individual' interfaces, we refer to Szyperski's observation that a component's interface usually consists of a set of interfaces to the behaviours / services a component provides [60]. We argue that Reflexion Modelling can identify approximations of individual elements of this set by capturing the distinct sets of services being offered to (or more accurately 'called by') different parts of the 'Remainder of the System'.
6. Again, the software engineer will continue with several iterations through step four and the subsequent creation of the Reflexion Model until he/she is satisfied with the new breakdown of the system.
7. The resultant model will potentially show the contents of the encapsulated component and its dependencies with the rest of the system. We suggest that this may serve to identify the roles the component plays in the system and thus provide guidance on the interfaces that the component presents to it (See Figure 3).

Figure 2, here please

Figure 2 –Encapsulating a component and making its interface explicit

Figure 3, here please

Figure 3 –Identifying multiple interfaces on a component using Reflexion Modelling

4. CASE STUDIES

In the previous section we described a plausible technique for component encapsulation. This section takes the proposed technique and actuates it in an industrial setting under realistic circumstances, on a large commercial system. Two software engineers, who currently maintain that system volunteered as participants after hearing an introductory presentation on the technique. They were shown the Reflexion Modelling tool (see section 4.2) and were allowed use it on a small application to familiarize themselves with its facilities. Additionally, the component encapsulation process, as we envisaged it, was presented to them. Overall, this training lasted less than 40 minutes.

They performed their component encapsulation task in their normal working environment, with the exception that the Reflexion tool plug-in was installed on their IDE (Eclipse [16]). The task of one participant was to arrive at a better architectural understanding of one of the components of the system, while the other participant had a specific maintenance task that the support team had tried before, but had cancelled on realising the associated difficulty. Both the participants' processes and products were subsequently evaluated to assess:

- *If the process supported software engineers in component encapsulation?* While previous studies have suggested Reflexion Modelling is useful in software understanding, this study focuses on the specific utility of Reflexion Modelling for user-targeted component encapsulation. Even though preliminary reports indicate that the technique would be promising in this context, there has been little empirical work to support this assertion.
- *If so, how did the process support component encapsulation?* The empirical work carried out to assess the utility of Reflexion Modelling has shown that it is useful, but not how or why it is useful. If the reasons for its success can be ascertained, they may be leveraged in other visualization efforts. Likewise, if we can identify factors that lessen the utility of the technique, studies can concentrate on eliminating or mediating these factors
- *If the participants adhered to or deviated from the process?* It is possible that these case studies could reveal refinements to the Reflexion Modelling process in this context. If so, these should be documented as guidance for other developers using the approach.

4.1 The Learning Management System

The subject system in this case study is the Learning Management System (LMS), currently being developed and maintained at IBM's Dublin Software Laboratory, Ireland. It provides services for administering eLearning to the employees of a company. Specifically, it makes learning content available to employees as courses, allowing them to enrol in those courses, tracking their progress through courses and reporting on mandatory training compliance.

The LMS is part of a larger application called "Workplace Collaborative Learning" (WCL). WCL adds collaborative features to the LMS that allows employees and instructors to discuss and share information and resources around their courses. It also allows them access to skills-management features which can be used to assist in the automatic provisioning of eLearning to employees.

The current LMS was originally developed from a legacy stand-alone product that was altered to become part of WCL. The WCL in turn also forms part of yet a larger application called "Workplace Collaborative Services" or simply Workplace. Workplace aims to provide employees with a new way to perform their work, where all their applications (email, calendaring, document management) are integrated with each other and with the other employees in the company through collaborative applications like instant messaging (IM), Web Conferencing and presence awareness. WCL is intended to be the part of Workplace that administers learning to the company's employees through Workplace. The sub application in-total consists of approximately half a million lines of code.

4.2 Tool Support

To enable the case study to proceed, we used the jRMTool, a Reflexion Modelling plug-in provided by Gail Murphy [38], for the Eclipse Java IDE [16]. This tool provides automated abilities for creating and viewing high-level models, for mapping software elements to high-level model elements, for displaying a Reflexion Model, for displaying summary information regarding the edges of that model and for displaying the source code entities that are unmapped.

A screenshot of the tool can be seen in Figure 4. As can be seen in the Figure, the mappings of the source code to the High Level Model are expressed in a basic XML language. Also note that, by clicking on an edge, the software engineer gets a list of the relationships that make up that list (edge information). These relationships include details of each calling method and its associated called method. In terms of a component's interfaces, the list of called methods implicitly declares (at least a subset of) the behaviour that the component makes available to the system. However this information is gleaned using static analysis of the code-base, and is subject to the normal constraints of static analysis [67].

Figure 4, here please

Figure 4 –A screenshot of the jRMTool eclipse plug-in

4.3 The Participants

The two volunteer participants of this case study had similar attributes. They were both software engineers with between one and a half and two years commercial Java^{TM} development experience. They both had a broad understanding of the LMS at a conceptual level, having worked on it for more than a year each and were both currently involved in evolving the system into the Workplace suite. Importantly, one of their assigned work tasks, as part of this exercise, was to gain a greater understanding of the LMS as a whole. The Reflexion Modelling exercise was seen as part of this agenda and this heightened the ecological validity of this in-vivo study [2].

4.4 Recounting the Process

4.4.1 Participant 1

The first participant chose to encapsulate a *Helpers Data Transfer Objects* component in the LMS. The participant was aware of this logical entity as a component and its use but was not certain of its makeup or how it was connected in the system. Helper Data Transfer objects are data transfer objects (DTOs, also known as Value Objects) that get passed between different layers of the system and are a combination of two standard J2EE design patterns[1]. They attempt to encapsulate some business object, such as a course or educational offering, retrieved from or persisted to the underlying database or GUI. For instance, when a HTML form is submitted to the presentation layer, the java Struts components build a helper DTO and fills it with all the data from the form, before passing it on to the service layer for further processing. From there, the helper DTO may be passed to the persistence layer, or back up the stack again, as required. It is a tidier method of passing data, than having interfaces with many individual parameters, but it does incur some overhead.

Using the Eclipse plug-in, described in the *tool support* section, the participant spent 2 hours, in a single sitting, encapsulating the Helpers DTO's component using our prescribed framework of guidelines. Elements were added to the high-level model through ten model iterations, in the following sequence[2] :

- Iteration 1: *Added* RestOfLearning and Helpers. [Steps 1 and 2]
- Iteration 2: *Added* Struts. [Step 3,4,5]
- Iteration 3: *Replaced* Struts with StrutsActionHanders, StrutsActionForms. [Step 3,4,5]
- Iteration 4: *Added* Services. [Step 3,4,5]
- Iteration 5: *Replaced* Services with ServiceInterfaces, ServiceImpls. [Step 3,4,5]
- Iteration 6: *Added* PersistenceMgrs. [Step 3,4,5]
- Iteration 7: *Added* JSPTags and WebserviceAPIs. [Step 3,4,5]
- Iteration 8: *Added* DSPersistenceBeans and RemoteProcessingCommands. [Step 3,4,5]
- Iteration 9: *Added* LMMPersistenceBeans. [Step 3,4,5]
- Iteration 10: No further additions - Finalised Map. [Step 6]

In creating his first Reflexion Model (see point 1 above) the participant followed step *one* and *two* of the process by separating out the targeted helper DTO component from the remainder of the system. He felt, after this first model, that he had an approximation as to the contents of the *helper DTO* component. Figure 5 shows the initial Reflexion Model and corresponding map. However, he chose not to iterate through the 2-node Reflexion Model again, as suggested by the process, instead commenting that he would only be absolutely certain of the contents of the component when he separated out the "RestOfLearning" node.

Figure 5, here please

Figure 5 –First Iteration by the first participant

Over the course of a further nine iterations 12 high-level elements were added to the model and two replacements were made, each affecting the 'Remainder of System' node, as anticipated. Thus the participant iterated through steps three and four of the process repeatedly. Replacements were made in situations where a high-level element needed to be

[1] An internal IBM pattern that combines the purposes of the Transfer Object pattern [62] ('Data' in this instance because it deals with data transfer as opposed to other transfers such as session transfer [14]) and Transfer Object Assembler [63]

[2]Note that the steps of the proposed encapsulation process are placed in square brackets.

subdivided into further sub-architectures. For example, in iteration five above, the participant replaced the services component with two components - one which provided interfaces to the services and another which provided the implementation of the services.

By the ninth iteration in the process the participant was satisfied that the 'Helper DTO Component' contained the correct source code entities and that the system's important interactions with the component had been isolated. This was confirmed when the participant finalised his map in the tenth Reflexion Model, thus completing step five of the process.

4.4.2 Participant 2

The second participant chose to encapsulate the *web administration user interface (UI)* of the LMS. This component forms part of a larger web interface component. The web component is the collection of Java classes that comprises the front end of the legacy web application of LMS, prior to its integration with Workplace. As the LMS is now being integrated into Workplace, it was important to partition the system into its legacy web administration element (for exclusion from workplace) and the rest of the system (for inclusion into Workplace). Hence this was an outstanding task that the participant had been assigned.

The participant did this task over several sittings, taking approximately three hours and thirty five minutes. He encapsulated the *web administration UI* component over the course of 11 iterations. Elements were added to the high-level model in the following sequence:

- Iteration 1: [Steps 1 and 2]
    - The web administration UI component - *Added* Delivery, Servelet, Actions.
    - Remainder of system - *Added* RestOfSystem.
- Iteration 2: The web administration UI component - *Added* View. [Steps 3,4,5]
- Iteration 3: The web administration UI component - *Added* OddBalls. [Steps 3,4,5]
- Iteration 4: The web administration UI component - *Removed* Taglib, OddBalls, *Added* Navigation. [Steps 3,4,5]
- Iteration 5: The web administration UI component - *Removed* Navigation, *Added* WebUtil. [Steps 3,4,5]
- Iteration 6-9: The high-level model remained the same for these iterations. The participant only made alterations to the map over the course of several refactorings to the code base. [Steps 3,4,5]
- Iteration 10: The web administration UI component - *Removed* Delivery. [Steps 3,4,5]
- Iteration 11: The web administration UI component - *Removed* WebUtil. [Step 6]

Figure 6, here please

Figure 6 –Web administration user interface component. (Note the bolded black box is not part of the tool's visual output)

The resultant Reflexion Model is shown in Figure 6. Here the map states that all of the source code in the package 'action' is to be mapped to the Reflexion Model node 'Actions', all of the source code in the package delivery is to be mapped to the node 'Delivery' and so forth.

Surprisingly, the participant immediately subdivided the web administration UI component (in his first iteration) into four major sub components, consistent with the package structure of the GUI. He also created a 'Remainder of System' component. This took five minutes. Based on previous domain knowledge, and other packages, several more sub components of the web administration UI component were added and deleted over the course of the 11 subsequent iterations. The high-level model stabilised in iterations six through nine where only changes to the map were made, as refactorings to the actual system were implemented. After the eleventh iteration the participant was confident that he had encapsulated the web administration UI component and successfully decoupled it from the system, both in the logical Reflexion Model and, through associated re-factorings, in the code of the system.

4.5 Evaluation

In evaluating the support provided by this approach we were interested in the process itself and the product of the process. In evaluating the process we used as our data:

- Concurrent talk-aloud [TA] generated by the participants as they performed their tasks. The participants were asked to state everything that came into their mind as they proceeded. They were also prompted when they fell silent, in line with best practice use of the technique [17]. While speaking aloud is not the most natural way for a person to behave, it has been shown to provide the richest insight into the mental state of participants at a point in time [49] when used in adherence with these guidelines.
- Diaries [D] kept by the participants as they proceeded with the process, which contained their thoughts on the tool, the technique and details on each iteration of the process.
- Interviews [I] with the participants after the case study.
- A fourth stream of evidence is found in the form of contextual knowledge [CK]. We define contextual knowledge, in this instance, as knowledge we obtained from the LMS team about the process' subsequent impact in the organization.

In evaluating the product we relied primarily on data generated by an architect of the system under analysis by the participants. This architect was recorded, using an MP3 recorder, as he viewed and evaluated the models produced by both participants. In addition, a complimentary, if somewhat subjective, analysis of the encapsulated component's quality was achieved by obtaining the participants' opinions as to the coupling and cohesion of the resultant components.

Obviously the methods reported here will not be suited to quantitative, statistical analysis. Indeed, given the high ecological validity of this in-vivo study [2] and the range of uncontrolled variables that this suggests, this evaluation work is more in line with the approach suggested by Seaman [54] or O'Brien et al. [43] in providing qualitative evidence of high ecological validity to act as a precursor, and to complement, other studies of a more quantitative, but less ecologically-valid, nature.

4.5.1 Evaluating Process Support

A number of streams of evidence support the utility of the process. Segments from the participants' talk-aloud, diaries and interviews show that they viewed the process very favourably in identifying the component architecture:

"it does such a good job of helping you understand the architecture of the system." [I]

"A really good tool for getting a high-level idea of a big amount of source code." [I]

"That's fairly revealing actually." [TA]

"That's interesting, looks like this package should be catered for in another node." [TA]

"Reflexion: great for spotting where dependencies were nonetheless ..." [D]

"I certainly intend to continue to use it myself." [I]

As the last quote shows, the participants' enthusiasm for the process led to assertions that they would continue to use the approach after the case studies. Indeed, since these case studies, the participants have acted as champions for the process and, as a result, 18 other software engineers in the organization have used the approach, and the associated tool, to study the architecture of their systems. Again, this technology transfer suggests that, as per our first research question, the process is supportive during component encapsulation [CK].

Another piece of contextual evidence [CK] that supports the utility of the tool was obtained by comparing participant 2's session with his previous attempt to separate the web administration UI from the system. On that occasion, he had relied upon compiler errors generated as a result of refactoring changes he introduced into the source code when trying to isolate the component. Over the course of two weeks he had used this strategy, unsuccessfully, to encapsulate the web administration UI component and eventually, the project was cancelled due to the perceived difficulty of the task.

"I did try this same job about two months ago and gave up after two weeks." [TA]

This time around, the task was accomplished by the participant in under four hours (0.5 days). Using the original attempt as a baseline, our technique allowed the encapsulation and refactoring of the same component at least[3] 20 times

3 'at least' because the programmer backed out of the previous attempt without success.

quicker than the approach previously used in the company. Again, this strongly suggests that the process is supportive of component encapsulation.

Apart from identifying components' constituent parts, another proposed benefit of the process for component encapsulation was that it would facilitate the identification of components' interfaces. For participant 2, where the goal of the encapsulation process was to completely isolate the web administration UI of the system, interface identification was not an issue. In his own words, the component encapsulation was only a success when "the LMS doesn't even know the web component exists". Hence, he employed a strategy whereby he broke down the UI in a package-led fashion, exclusively to identify its constituent parts:

> "breaking down (the) GUI was package based" [I] .

The Helpers Data Transfer component encapsulated by Participant 1, in contrast, should have associated interfaces. Surprisingly however, Participant 1 stated that he also broke down the 'Remainder-Of-System' node predominantly to refine his understanding of the component's contents.

> " it was to figure out what was in and out" [I]

However, when asked specifically if 'the edges were of any help', he paused and said:

> "Yeah, yeah,... there's a lot of them, but each edge tells me how it works with a part of the system… helping with the interface, even though huge… there'd be some overlap, but it wouldn't be a bad starting point to explore these from".

The 'huge' adjective used in the quotation refers to the large number of relationships underpinning the edges incident on the Helpers component in the Reflexion Models (for example in Figure 5, the incident edge represents 2240 relationships). This is largely based on the size of the system, but there are also other factors at play. For example, each relationship counted could be unique in its calling method, its called method, or both. However, in obtaining an approximation of the component's interface, we are only interested in the distinct called methods. In addition the jRMTool captures data accesses in its edge relationships and there are not typically viewed as interface constituents. However, to conform to encapsulation principles, direct data access should probably be replaced by method-mediated data access and so, even these relationships inform on the possible constituents of the component's interface.

Later in the Reflexion Modelling process, when the 'Remainder of the System' is partitioned, we argued that approximations of individual interfaces could be obtained (section 3.3, bullet point 5). Again, referring back to the quotation by the participant, where he mentioned 'overlap', he was noting an issue with the identification of individual interfaces: namely that many of the individual interfaces contained the same called method and that some rationalization would be required to derive a set of non-redundant interfaces using this technique.

4.5.2 Evaluating Process Support through the Product

Of course the process is only truly supportive if it produces a high quality product. In order to evaluate this, we presented the resultant Reflexion Models to an architect of the LMS, for comment on the component abstractions produced. This evaluation step is unique to this study even though it defines, to a large degree, the success of the technique in this context.

The architect participant had two years experience working as an architect on the LMS system specifically and several years' prior experience as a systems architect. His evaluation concentrated on:

- The quality of the component.
- The correctness of the mappings to the component.

The architect agreed with the component encapsulated by participant 1 (the DTO Helper component) and the mapping made to it, with one exception; in Figure 7, we see a portion of participant 1's final model. It was the participant's hypothesis that communication from JSPTags to DBPersistence beans was a breach of the Helper DTO design pattern implemented in the system and that all communication should be routed through the Helpers component. The architect instead saw the participant's 'design-pattern flaw' as correctly implementing a View Helper [65] pattern which permits this type of communication. However, after further examination of the Reflexion Model the architect acknowledged that:

"helpers *are* being relied upon to do any manipulation."[I]

"Seems to be doing two patterns ...object assembler (DTO) and view helper pattern."[I]

He summed up his review of the component stating that the component and interfaces were,

"valid and interesting ...I would have never thought of this [helpers] as a view holder. Didn't know the extent of use."[I]

"This is not just [Participant 1's] perception ...it is quite a common pattern."[I]

In fact, the Reflexion Model showed that 90% of all JSPTag calls were through the helpers DTO component, showing a strong DTO pattern presence in the implemented system.

Figure 7, here please

Figure 7: The Helper DTO design pattern (lower 2 edges) and the View Helper Pattern (upper edge) in the LMS.

For Participant 2's component-decoupling task, the architect completely agreed with the high-level model and the contents of the participant's model. Not only could he appreciate the validity of the component encapsulation but also the motivation behind it:

"all communication could potentially happen over a webservice." [With the refactored system] [I]

,the ultimate intention of participant 2. With respect to the correctness of the model the architect stated,

"there is unlikely to be missing of extra edges here because we've done the work [actually refactored the code]...that is what he wanted it to be ...that's good, well done [Participant 2] ...there's not an awful lot else to say ...it's right." [I]

Indeed, probably the most convincing piece of data regarding the product produced by the process was a note made by participant 2 in his diary on the result of his Reflexion Modelling:

"Delivery is back with the 'RestOfSystem.' No unwanted links to RestOfSystem, job done basically ...Now I have just what I always wanted: the LMS doesn't even know the web component exists… There's been a few hacks along the way, but Reflexion allowed me to see the big picture, safe in the knowledge that this was, at least, the minimum number of hacks in the just the right places." [D]

A second, albeit more subjective, stream of evidence as to the product quality came from the participants themselves. They were asked to comment on the coupling and cohesion of the resultant component, where *Coupling* is the degree of interdependence between components or modules [58], and *cohesion* is the extent to which a component or module's individual parts are needed to perform the same task [70]. Using Fenton and Phfleeger's [19] 6 point scale of increasing coupling, and Yourdon and Constantine [70] seven point scale of decreasing cohesion we asked the participants to classify the intra-component coupling, the inter-component coupling (between the component and the rest-of-the-system) and the component cohesion. The results are presented in Table 1, where the actual levels of coupling and cohesion are presented in brackets.

Table 1, here please

Table 1 –Coupling and Cohesion Classification by Participants

Both participants thought that the intra-coupling exhibited by the resultant component was at level 4 in Fenton's classification schema. That is, the modules in each component referred to the same global data. [19] suggest that this classification is indicative of tight coupling and thus, based on the participants' classifications, there is tight coupling within each component.

When discussing the coupling between their components and the rest of the system, the two participants differed:

> "Certainly it is not 0; x and y do communicate…. In my opinion, it could be 1 or 2." [I] (Participant 1)

> "Coupling between the 'rest of system' (ROS) and the web module is either 2 or 3 ...probably 3, since there is, for example, sometimes some, very very basic, business logic contained in the web module" [I] (Participant 2)

Thus, we conservatively calculated the inter-component coupling of the components as 2 and 3 respectively. As the intra-component coupling is higher than inter-component coupling in both cases, this suggests that the components have some coupling merit. Likewise, the cohesion classification carried out by the participants suggests that the components have a cohesive merit, particularly in the case of Participant 2:

> "…Functional cohesiveness, yes 1 (functional cohesion) is appropriate I think." [I] (Participant 2)

4.5.3 Reservations on Process Support

One reservation about the approach was expressed by Participant 1. He highlighted the pre-requisite of mnemonic naming as a basis for undertaking the process:

> "You'd have a real problem doing this [process] if you didn't have a consistent naming scheme."

Two worthwhile points are highlighted by this comment. Firstly, without a consistent naming scheme for a subject system, the map for the source model would become very large and effort-intensive, as users of the jRMTool would not be able to catch a large set of software elements in a single lexical pattern. Secondly, if the naming scheme was inconsistent and the names held little clue as to meanings of the classes and methods, then the participants would find the creation of the map more difficult, having to rely instead on possibly incongruent existing super-structures like directories or packages when forming their component. Thus, the developer may have difficulty in selecting source code entities for their targeted component. These issues suggest that there is scope for combining techniques other than mnemonics in this part of the component encapsulation process. As mentioned above, [11] have already looked at using clustering techniques to augment this process with promising results. Another promising technique, under investigation in this regard, is to leverage software Reconnaissance [68], to identify source code elements, reusable across features. These re-useful elements could then act as a seed set for a reusable component. We are actively exploring this possibility [32], [33].

In addition the participants also suggested several improvements that would make the supporting jRMTool better. These included:

- Improved visualization for large-scale system: The original jRMTool was a research prototype and, in scaling up to large scale systems, requires additional visualization facilities such as the ability to:
  - Define / Present sub-architectures, thus enabling a hierarchical modelling as described in [30];
  - Grey out specific edge types (in particular expected edges and self loops);
  - Grey out any nodes not immediately connected to a specified node of interest.
- Greater integration with the Eclipse IDE. In particular, the participants would have liked the ability to link directly back into the 'calling' and 'called' code from the 'edge-information' view.
- Create and update the source code map through a visual interface.

However, overall the process was deemed to have provided high-quality components, and the participants were sufficiently enthused that they championed it to their colleagues in two teams. Indeed, one of these recommendations has resulted in a separate study into architecture conformance using Reflexion [31].

4.5.4 How did the Process Support Component Encapsulation

Having established that the process seemed to provide support for component encapsulation in these two instances, we next turned our attention to assessing the form of this support. Specifically, we noted two related cognitive processes that were prevalent in the talk-aloud data generated by both participants. These were episodes when the participants Hypothesized, Evaluated and Refined (HER) their assumptions about the component to be encapsulated, and episodes where they Hypothesized Evaluated and Accepted (HEA) their assumptions. These can be seen from the following quotes:

> "I'd be interested to know if the action forms are using helpers, because I'd totally expect the action handler to use them ...[calculates Reflexion Model] so there are some calls ...most of the calls from struts are going through the action handlers ...expected. (HEA)"

> "I would expect the service tier to use it, ...so I'll add another node ...and I would be very, very surprised if there was an arc from the services to the helpers ...it shouldn't happen, these guys shouldn't have any actual processing logic ...[calculates Reflexion Model] ...so there are three calls from the helpers to the services, which suggests there might be a problem there, sounds like something is being violated there." (HER)

> "There are no unwanted dependencies on Servlets, which is a relief. I can essentially forget about this package now" (HEA)……37 unwanted links to Delivery are a cause of concern" (HER)

> "Looks like there's a few links all right, from rest of system back to view, ...I was hoping there would be none, so that's a shame." (HER)

> "There is a call from helpers to struts ...didn't expect that ...It's always been unclear to me whether the validation occurs in the action handle or if it validates itself ..." (HER)

> "OK, jsputil ...oh right, obviously that has no place in rest of system, OK so that's gonna have to be the first thing I'll have to tackle. Straight away I can see that the jsputil should be in the web component." (HER)

The HER episodes presented above map closely to cognitive conflict theory, as described in section 3.1. Specifically, the software engineer makes explicit hypotheses regarding the relationships between high-level components in the system using his/her HLM. The Reflexion Model is used to evaluate these hypotheses and some of them are found wanting, thus causing cognitive conflict. It is interesting to note, from our data, that HER episodes typically led to detailed exploration of the parts of the source code that repudiated these hypotheses, in order to analyze the surprising inconsistencies. This exploration often lead to a change in the source code mapping, a change in the engineers' High-Level Models or a (refactoring) change in the source code. Thus, in line with cognitive conflict theory, learning is motivated by repudiated hypotheses and equilibrium is subsequently restored.

Indeed, the stronger the user's initial mental model, the greater the cognitive conflict when a hypothesis is repudiated, and the greater the resultant impetus for learning. Thus, engineers with strong architectural assumptions about the system under study should gain most from Reflexion Modeling. In contrast, relative novices, with less strongly embedded assumptions about the system, would be less likely to benefit from Reflexion Modeling, as the conflicts they experience would not be so surprising to them.

This would seem consistent with the empirical literature. Specifically, it is notable that, in all successful evaluations [29], [32], [33], [38], [39] of Reflexion Modelling, the user is someone who could legitimately be expected to have strong assumptions about the subject system's domain or macro-structure. In another study we performed at IBM, software immigrants [57] new to the LMS system (and thus possessing only a weak mental model) had a poor impression of Reflexion Modelling. This suggests a refinement of previous Reflexion Modelling literature [39], which ignored the significance of a strong mental model and which suggested that the modelling technique would be particularly beneficial for novices.

Corollary 1: Users of Reflexion Modelling benefit from a strong, pre-formed mental model, even if that model is flawed.

Corollary 2: Reflexion Modelling users to enter a cycle of hypotheses generation and hypothesis evaluation. This cycle provides a strong impetus for detailed exploration and learning when hypothesis are shown to be inconsistent with the system.

Interestingly, cognitive conflicts would be much less pronounced if the software engineer was simply shown a diagram of the existing system without first being required to explicitly state their own model (and then being presented with the conflicts). This lack of conceptual change is apparent in a number of studies of software visualization tools that simply present their findings [18] without forcing the user into a situation where they must prematurely commit to a model.

In contrast, HEA episodes, where there was no hypothesis repudiation, typically led to task-switching, where the engineer changed their focus of attention on to another hypothesis and on to other parts of the Reflexion Model. That is, expected edges were studied far less often than unexpected edges, a finding in contrast to [39]. The participants' focus on unexpected edges in this study could also mean that unexpected relationships in the source code might go un-explored. For example, an expected edge in the Reflexion Model might be made up of 10 expected source-code relationships and one unexpected source-code relationship. Because these expected edges were not explored in detail, the unexpected relationship may remain uncovered. [38] implicitly acknowledge that this type of problem may occur when they characterize the technique as 'approximate' and, towards this end, the jRMTool does quantify the number of source-code relationships underpinning each edge. Yet the participants in our study seldom used this edge attribute in their efforts and never used it to identify unexpected relationships.

This finding is in line with 'fit-to-theory' literature discussed in section 3.1 where the software engineer accepts that the expected edges presented map exclusively onto the relationships they expect to exist within the source code. While the engineers know the approximation inherent in the Reflexion Modelling approach, they tended to ignore the possible inconsistencies between edges and source code relationships, preferring instead to let the edge data 'fit' their theory.

Corollary 3: Users of Reflexion Modelling should not ignore expected edges.

When discussing Reflexion Modelling for this task, Participant 2 noted as a supportive attribute of Reflexion Modelling: working at a level above source code. He stated that, with Reflexion Modelling, he could avoid many hundreds of the compile dependency errors he would encounter when making changes at the source code level:

> "The alternative to this, maybe, is to make the changes to the code. But if I could use the model ... again, not touching the compiler, until I really have to. I could gradually put stuff into this oddball node. Then at the end when I wanna start actually changing code you can, actually start the refactoring proper and keep rebuilding my Reflexion Model, and eventually all the stuff in my oddball node will go down to zero and I'll have no stray links, so I'm gonna try that." [TA]

Thus, he identified the primary role of Reflexion Modelling in the process:

> "[the] compiler is like a final check, Reflexion should be your first check." [TA]

Of course, an even better solution in this instance would be to create automatic refactorings, triggered by the Reflexion Model. Work is underway at the University of Bremen is to investigate the possibility of this goal.

4.5.5 Adherence to the Process

The original process specified that participants would have a phased approach. Firstly, the participants would create a High-Level Model with two nodes: The potential component and the 'Remainder of the System'. They would then assess the Reflexion Model to refine the contents of the component. Afterwards, the nodes could be broken down to further refine the contents of the component and to elicit its individual interfaces.

In fact, the behaviour of the participants was less structured. Participant 2 for example, immediately divided the UI component into four different nodes, based on its package structure. Participant 1, while dividing the system up into the two nodes originally, moved quickly to refine the ‘Remainder of System’ node into further sub-types and only then spent substantial time exploring his Reflexion Model.

In addition, the interface-elicitation rationale for breaking down the components was never in evidence. While this was understandable for participant 2, whose goal was to create a component with no interface to the system, it was surprising for participant 1. As previously stated in section 4.5.1, he refined the nodes in his High-Level Model based on the rationale of identifying the component’s constituent parts. Indeed, he only became aware of the possibility of using edge information for interface identification in the interview after his session, even though the proposed process was described in advance of his case study. However, in this post-session interview, he was positive about the ability of the edges to inform on the interfaces of the component.

5. VALIDITY

Important to any experiment or case study is an assessment of its validity. Validity refers to the meaning of experimental results [40] [43] and can be categorised under three headings [45]:

External validity

External validity is the degree to which the conclusions of a study are applicable to the wider population. A controlled experiment could afford a larger population size, allowing the possibility of statistical analysis, and thus, generality. This is not possible in in-vivo case-studies such as ours. We have partially countered this weakness by having two case study participants rather than one and demonstrating a degree of external validity by showing repeated success.
However, it should be noted that the constrained nature of a controlled experiment destroys the ecology of the setting and prevents researchers from gaining accurate insights into the behaviour of experienced industrial software engineers in realistic scenarios [43]. This concern is called ecological validity - a subset of external validity. Our experiment rates very high on the scale of ecological validity due to the use of real, experienced software engineers, in their workplace, performing actual work tasks, on a large commercial system.

Internal validity

Internal validity is the certainty with which we can say that the known variables in the study are the only causes of what was observed. The only factors that were altered from the participants’ normal in-vivo practice were the use of the process, the use of the jRMTool and talk-aloud. The positive response in terms of comments, championing the technique and, in the case of participant 2, in terms of improved performance in completing his task suggests that one or more of these three factors was at play. The talk aloud protocol may have impacted on the participants’ performance but, given the extra processing load this protocol implies, it is likely that any impact would have been negative rather than positive. Hence the effect is likely to be related to the process and / or the jRMTool.

Construct validity

Construct validity refers to the degree to which the structure of the experiment affords the measurement of what the experimenter set out to measure. The empirical studies in this publication demonstrate high construct validity by using multiple streams of data to assess their research questions. In terms of the process we were attempting to determine if it was helpful and how it was helpful. We have tried to ensure high construct validity by evaluating both the product and the process using talk-aloud, metrics, interviews, diaries, architects and multiple participants. Combined with available contextual evidence, we are able to state with high confidence that the process was a large improvement on current practices.

6. SUMMARY

We have demonstrated and evaluated Reflexion-based component encapsulation, using two participants who actuated the technique on a large, commercial learning-management system. This evaluation was undertaken on both the process and product and triangulated from several sources.

In terms of the process, the participants were enthusiastic in their reporting on the technique and in their championing of it to colleagues. Likewise, contextual knowledge suggested that, in the case of participant 2 in particular, the component could not have been encapsulated using any of the alternative techniques available to him. In addition, the components produced by the technique were deemed of high quality by the architect in charge of the system. All these are positive indicators for the technique's use in targeted component encapsulation. However, the reliance of the technique on mnemonics was noted and the technique must still be considered largely untested with respect to interface-elicitation, even though early indications from this study were positive.

In terms of 'how' the Reflexion Modelling process supported targeted component encapsulation, the findings were also revealing. Software engineers liked to work at the modelling level and tended to use a cycle of Hypothesis, Evaluation and Refinement to explore the component structure of the system. This seems to be quite a powerful technique but also suggests that the technique should be accompanied by several guidelines. Most notably, before software engineers use the technique, they should have strong mental expectations as to the structure of the system. Likewise, they should be asked to explore expected edges for unexpected source-code relationships and they should be asked to consider deriving the component's interfaces as they refine their Reflexion Models.

7. FUTURE WORK

Research scope exists far beyond the work reported on in this publication. For example, as mentioned in the *participant evaluation section*, there is research scope in removing the reliance of the technique upon mnemonic naming conventions. One approach we suggest is to augment the component encapsulation process with a variation on a feature mapping technique called software Reconnaissance [68]. Work towards this agenda can be found in [33], [34], [35]. Future work will see us evaluating this approach.

The ability to quickly and clearly identify useful refactorings is also touched upon in the paper. It is our intention to expand our existing case studies to the evaluation of our technique as a means of identifying and thus aiding the implementation of refactorings for large systems.

Both participants in the case study highlighted a list of improvements to the tool support provided for our process. We are currently incorporating these suggested changes into the existing jRMTool. Interestingly, many of the suggested improvements to the tool support are currently available in the Bauhaus reengineering tool set [3]. It would seem that a natural next step is to implement a larger study and evaluations using the Bauhaus toolset.

Another research avenue we wish to explore, arising from the architect's evaluation, is the potential to use structural summaries to help architects enforce design constraints, as prescribed by them during system design. Some work has already been done on this [55], [6] but not with a toolset as lightweight as Reflexion Modelling.

With this approach the architect would create the model of the system before it has been developed. As parts of the system are created the architect can map these pieces to the high-level model. If constraints of the architecture are to be broken, they can be caught at implementation time, rather than at the end of the project in retrospect. We have already carried out exploratory studies in this area [31] and intend to progress this work further after promising initial results.

8. CONCLUSIONS

In this publication, we explicitly described a small variation on the software Reflexion Modelling technique, for the encapsulation of user-targeted components. We evaluated this process using a large industrial case study with two participants, carrying out real-world tasks on a large commercial system.

This evaluation took the form of observation notes, talk-aloud, contextual information and diary data gathered from the two participants. Combined with interviews and evaluations from a system architect, they show strong approval for the process and the resulting product. When we evaluated 'how' the process supported targeted component encapsulation, we identified the following best-use guidelines:

- The technique is better suited to more experienced developers with strong pre-formed mental models of the system under study;
- Software engineers using the technique should be warned to explore all Reflexion Model edges, not just the unexpected ones;
- Software Engineers may need additional support when non-standard naming conventions have been adopted in the organization;
- Software Engineers may need additional prompting to derive interfaces for the encapsulated components. Indeed, the efficacy of the process for interface elicitation remains largely uncertain, even if early indicators from this study are positive. This is an area of future work for the group.

More generally, our findings suggest that cognitive conflict, where a software engineer pre-commits to a model of the system and then is allowed compare that model to the actuality of the system, is a powerful mechanism, which can direct learning and motivate the learner. These beneficial effects should be further explored and evaluated as, if they prove generally applicable, cognitive conflict could be leveraged in a range of software visualization tools.

In summary, all avenues of evaluation that we undertook have indicated that this process is a useful and significant improvement upon normal approaches for targeted component encapsulation undertaken in industry.

ACKNOWLEDGEMENTS

We would like to thank the Code Driven Business Process Recovery (CDBPR) Research Group, QAD Ireland, Lero (supported by Science Foundation Ireland grant number 03/CE2/I303_1), the IBM Dublin Software Laboratory, QAD Ltd, Gail Murphy for her help with the jRMTool and Enterprise Ireland for their contribution in enabling this work.

REFERENCES

1. Aldrich J. Chambers C. Notkin D. Archjava: connecting software architecture to implementation *International Conference on Software Engineering* 2002; 187-197
2. Basili, VR. The Role of Experimentation in Software Engineering: Past, Current and Future. *Keynote address in 18th International Conference on Software Engineering*, 1996
3 Bauhaus Reengineering Toolset. http://www.bauhaus-stuttgart.de/bauhaus/ [Last accessed 15/03/2006].
4 Biggerstaff TJ. Design recovery for Maintenance and Reuse. *IEEE Computer* 1989; **7**(22):36-49.
5. Blackwell A. and Green T.. Notational Systems – the Cognitive Dimensions of Notations Framework. In HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science. John M. Carroll (Ed.)
6. Bundy A., Blewitt A., and Stark I.. "Automatic Verification of Java Design Patterns". Proceedings of 16th International Conference on Automated Software Engineering
7. Cheesman J, Daniels J. *UML Components : A simple Process for Specifying Component-Based Software*; Addison Wesley, 2001
8. Chikofsky EJ, Cross II JH. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software* 1990; pages 13-17.
9. Chiricota Y, Jourdan F, Melancon G. Software components capture using graph clustering *International Workshop on Program Comprehension* 2003; pages 217-227
10. Choi SC,Scacchi W. Extracting and restructuring the design of large systems. *IEEE Computer* 1990; **1**(7):66-71.
11. Christl A, Koschke R, Storey M-AD. Equipping the Reflexion Method with Automated Clustering. *Working Conference on Reverse Engineering* 2005; pages 89-98
12. Chung W, Harrison W, Kruskal V, Ossher H, Stanley J, Sutton M, Tarr P. Working with implicit concerns in the concern manipulation environment *Linking Aspect Technology and Evolution Co hosted with Aspect Orientated Software Development* 2005;
13. Cimitile A, Visaggio G. Software salvaging and the call dominance tree. *Journal of Systems Software* 1995; 28(2):117-127.
14. Data Transfer Object: *http://en.wikipedia.org/wiki/Data_Transfer_Object* [Last accessed 12/09/07]
15. Doval D, Mancordis S, Mitchell BS. Automatic clustering of software systems using a genetic algorithm. *Proceedings of the International Conference on Software Tools and Engineering Practice* 1999;
16. Eclipse IDE Homepage. http://www.eclipse.com [Last accessed 12/09/06].
17. Ericsson K, Simon H. Protocol Analysis - Revised Edition, Verbal Reports as Data. *MIT Press*, 1993
18. Exton C. and Kolling. M.. Concurrency, objects and visualisation, in Proceedings of ACM SIGCSE Fourth Australian Computing Education Conference (ACE2000), Melbourne, December, pp 109-115.
19. Fenton NE, Pfleeger SL. *Software metrics: A rigorous and practical approach*; Thompson Computer Press, 1997.
20. Gall, Harald, Klusch Finding objects in procedural programs: an alternate approach. *Proceeding of Second Working Conference on Reverse Engineering* 1995; pages 208-216.
21. Girard J-F, Koschke R. Finding components in a hierarchy of modules: a step towards architectural understanding. *International Conference on Software Maintenance* 1997; pages 58-65
22. Hassan A. and Holt R. Using Development History Sticky Notes to Understand Software Architecture. Proceedings of the 12th International Workshop on Program Comprehension. pp 183-193.
23. Hutchens DH, Basili VR. System structure analysis: clustering with data bindings. *IEEE Transactions on Software Engineering* 1985; **8**(SE-11):749-757.

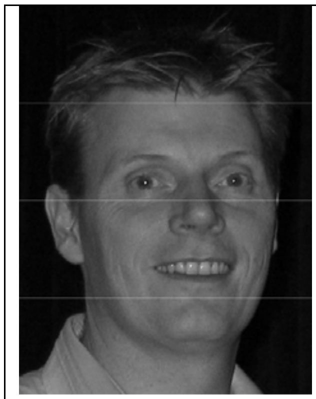
24. Johnson PD. Mining legacy systems for business components: an architecture for an integrated toolkit. *Proceedings of the 26th International Computer Software and Applications Conference* 2002;
25. jRMTool Reflexion Modelling eclipse plug-in. http://www.cs.ubc.ca/murphy/jRMTool/doc/ [21/12/2003].
26. Kazman, Rick, Carrиre Playing detective: reconstructing software architecture from available evidence. *The Software Engineering Institute at Carnegie Mellon University* 1997; CMU/SEI-97-TR-010, ESC-TR-97-010.
27. Kniesel, G. *First Workshop on Unanticipated Software Evolution. ECOOP* 2002.
28. Ko, A. J., DeLine, R., Venolia, G.. Information Needs in Collocated Software Development Teams. *International Conference on Software Engineering (ICSE)*, 2007: 344-353.
29. Koschke, R. Atomic architectural component recovery for program understanding and evolution. *Institute for Computer Science, University of Stuttgart* 2000; PhD. Thesis.
30. Koschke R, Simon D. Hierarchical Reflexion Models. *Working Conference on Reverse Engineering* 2003;
31. Le Gear A., Buckley J., and Mcilwaine C.. Exercising Control Over the Design of Evolving Software Systems Using an Inverse Application of Reflexion Modelling. Proceedings of CASCON Dublin Symposium.
32. Le Gear A.. "Component Reconnexion" PhD Thesis, University of Limerick.
33. Le Gear A, Buckley J, Collins JJ, O'Dea K. Software reconn-exion: understanding software using a variation on software reconnaissance and Reflexion Modelling *International Symposium on Empirical Software Engineering* 2005;
34. Le Gear A, Buckley J. Reengineering towards components using "Reconn-exion" *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* 2005; pages 370-373
35. Le Gear A, Buckley J. Reengineering Towards Components with "Reconn-exion." *ESEC/FSE Doctoral Symposium 2005* 2005;
36. Lindig C, Snelting G. Assessing modular structure of legacy code based on mathematical concept analysis. *International Conference on Software Engineering* 1997; pages 349-359.
37. Mitchell BS, Mancoridis S, Traverso M. Search Based Reverse Engineering *Proceedings of the 14th international conference on software engineering and knowledge engineering* 2002; pages 431-438.
38. Murphy GC, Notkin D, Sullivan K. Software Reflexion Models: bridging the gap between source and high-level models. *Symposium on the Foundations of Software Engineering* 1995; pages 18-28
39. Murphy GC,Notkin D. Reengineering with Reflexion Models: a case study *IEEE Computer* 1997; **2**(17):29-36.
40. Oates B.J.. Researching Information Systems and Computing. *Sage Publications.* 2006.
41. Ogando RM, Yau SS, Liu SS. An object finder for program structure understanding in software maintenance. *Journal of Software Maintenance: Research and Practice* 1994; **5**(6):262-283.
42. O'Brien M.P.. Evolving a model of the information-seeking behaviour of industrial programmers. *Ph.D Thesis, University of Limerick,* 2007.
43. O'Brien M.P., Buckley J, Exton C. Empirically studying software practitioners - bridging the gap between theory and practice. *International Conference on Software Maintenance* 2005; pages 433-442.
44. Patel S, Chu W, Baxter R. A measure for composite module cohesion. *Proceedings of the 14th international conference on software engineering.* 1992; pages 38-48.
45. Perry D, Porter A, Votta L. A Primer on Empirical Studies. *Tutorial Presented at the International Conference on Software Maintenance* 1997;
46. Piaget, J.. The Equilibration of Cognitive Structures: The Central Problem of Intellectual Development. Chicago: University of Chicago Press. 1985.
47. Pohl K., Bockle G., and van der Linden F.. Software Product Line Engineering: Foundations, Principles and Techniques. *Springer. ISBN: 978-3540243724*
48. Robillard MP, Murphy GC. Concern graphs: finding and describing concerns using structural program dependencies. *International Conference on Software Engineering* 2002;
49. Russo J, Johnson E, Stephens D. The Validity of Verbal Protocols. *Memory & Cognition* 1989; Vol. 17.
50. Schwanke RW. An intelligent tool for reengineering software modularity *International Conference on Software Engineering* 1991; pages 83-92
51. Scientific Method. http://servercc.oakton.edu/~billtong/eas100/scientificmethod.htm [Last accessed 16/10/07].
52. Seacord RC, Plakosh D, Lewis GA. *Modernizing Legacy Systems*; Addison Wesley, 2003
53 Seaman C.B. and Basili V.R.. Communication and Organization: An Empirical Study of Discussion in Inspection Meetings. *IEEE Transactions on Software Engineering* 1998. 24(7): 559-572
54. Seaman C. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering* 1999; **1**(25):557-572.
55. Sefika M., Sane A., and Campbell R.H.Monitoring Compliance of a Software System with its high-level Design Models. Proceedings of the 18th International Conference on Software Engineering. pp 387-396

56. Shaft TM. The role of application domain knowledge in computer program comprehension and enhancement. *Doctoral Dissertation, Pennsylvania State University, University Park PA, 1992; 213–269.*
57. Sim S.E. and Holt. R. The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize. *Proceedings of the 20th International Conference on Software Engineering, pp. 361-370.*
58. Stephens W, Myers G, Constantine L. Structured design. *IBM Systems Journal* 1974; **13**(2).
59. Strike, K., and Posner, G.. (1992) A Conceptual Change View of Learning and Understanding. In L. West & R. Hamilton (Eds.), Cognitive structure and conceptual change (pp. 211-232). London: Academic Press.
60. Szyperski C.. Component Software: beyond Object Oriented Programming. N*ew York ACM Press / Addison Wesley Publishing Co.* 1998
61. Szyperski C. Component technology - what, where and how? *International Conference on Software Engineering* 2003; pages 684-693.
62. Transfer Object: *http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html* [Last accessed 12/09/07]
63. Transfer Object Ass.: *http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObjectAssembler.html* [Last accessed 12/09/07]
64 Valasareddi RR, Carver DL. A graph-based object identification process for procedural programs. Proceedings of the Working Conference on Reverse Engineering 1998; page 50.
65. View Helper: http://java.sun.com/blueprints/corej2eepatterns/Patterns/ViewHelper.html [Last accessed12/09/07]
66. Von Mayrhauser A., and Vans A.M.. Identification of Dynamic Comprehension Processes During Large Scale Maintenance. *IEEE Transactions on Software Engineering* 1996, 22(6): 424–437, 1996.
67. Wendehals L. Improving Design Pattern Instance Recognition by Dynamic Analysis. *In Proceedings of WODA: The Workshop On Dynamic Analysis 2003.* http://www.cs.nmsu.edu/~jcook/woda2003 [last accessed 16/10/07]
68. Wilde N, Scully MC. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice* 1995; **1**(7):49 - 62.
69. Yeh AS, Harris DR, Reubenstein HB. Recovering abstract data types and object instances from a conventional procedure language. *Working Conference on Reverse Engineering* 1995; pages 227-236.
70. Yourdon E, Constantine LL. *Structured design*; Prentice Hall, Englewood Cliffs, NJ, 1979.
71. Zweben SH, Edwards SH, Weide BW, Hollingsworth JE. The effects of layering and encapsulation on software development cost and quality *IEEE Transactions on Software Engineering* 1995; **3**(21):200-208.

AUTHORS' BIOGRAPHIES

	Jim Buckley obtained an honours BSc degree in Biochemistry from the University of Galway in 1989. In 1994 he was awarded an MSc degree in Computer Science from the University of Limerick and he followed this with a PhD in Computer Science from the same University in 2002. He currently works as a lecturer in the Computer Science and Information Systems Department at the University of Limerick, Ireland. His main research interests are in theories of information seeking, software reengineering and software maintenance. In this context, he has published actively at many peer reviewed conferences / workshops and is a Faculty Fellow with the IBM Centre for Advanced Studies in Dublin. He has also co-organized a number of international conferences and workshops in this domain. He currently coordinates two research projects at the University: one in the area of software visualization and the other in architecture-centric re-engineering and evolution.
	Andrew LeGear received a BSc. in computer systems from the University of Limerick in 2003. He is a recently graduated PhD. candidate at the same University. Andrew has worked as a researcher in the Software Architecture Evolution group since 2003 and collaborates on research projects with QAD Ltd. and IBM Inc. He is the author of numerous peer-reviewed workshop and conference papers and project proposals. His interests include software maintenance, software engineering and software evolution.

	Dr. Chris Exton is currently a lecturer in the department of Computer Science and Information Systems at University of Limerick. He has worked extensively in the commercial software development field in a variety of different institutions including software houses, manufacturers and food retailers. His last industrial appointment before embarking on his academic career was as a senior technical consultant with ANZ Bank, Australia for a period of three years. His work in industry has included Software Engineering positions in Australia, Ireland and the UK, where he has worked in companies as diverse as Ashling Microsystems Limerick, and Coca-Cola, London. Aside from his industrial software development experience he has completed a PhD in the area of Distributed Systems at Monash University, Melbourne, Australia. Since then he has worked at a number of Schools and Departments around the world. These include the School of Network Computing and the School of Computer Science and Software Engineering at Monash University, Australia, the Department of Computer Science, University College Dublin, Ireland and the Department of Computer Systems, Uppsala University, Sweden. His main research interests are in programming psychology/behaviour and software tools. In this context, he has published actively at many peer reviewed conferences / workshops and is a Faculty Fellow with the IBM Centre for Advanced Studies in Dublin.
	Ross Cadogan received a BSc in Computer Science from University College Cork in 2003. He joined IBM on graduating from college, and works on IBM Workplace Collaborative Learning in IBM's Dublin Software Lab. His roles have included integrating IBM's learning system with team spaces and Sametime for collaborative and realtime capabilities.
	Trevor Johnston joined IBM in 2003 as an intern on the "Extreme Blue" programme before returning to complete his degree in computer applications from Dublin City University. Once obtained, he returned to IBM full-time where he now works on IBM Workplace Collaborative Learning. In this role, he specialises inWorkplace Client and search capabilities. He is interested in open source software and in technical writing and has (co-)published a number of articles and books for IBM.

	**Bill Looby** is a graduate from the University of Limerick. He spent some time in Cork developing network backup software before moving to Dublin to join Lotus, where he spent several years developing localisation tools and architecting web content globalisation systems, Collaborating with the Localisation Research Centre in the University of Limerick on localisation systems and emerging standards, Bill initiated the creation of the Translation Vendor Web Service standard. Bill moved into the architecture of end-user web site development systems with the advent of the Workplace product, and most recently has taken the role as DSL Database Architect with specific application to Workplace Collaborative Learning.
	Rainer Koschke is a professor for software engineering at the University of Bremen in Germany. His research interests are primarily in the fields of software engineering and program analyses and his current research includes architecture recovery, feature location, program analyses, clone detection, and reverse engineering. He is one of the founders of the Bauhaus research project, founded in 1997 to develop methods and tools to support software maintainers in their daily job through reconstructed architectural and source code views. He teaches reengineering and software engineering at the University of Bremen and holds a doctoral degree in computer science from the University of Stuttgart, Germany. He is the current Chair of the IEEE TCSE committee on reverse engineering and initiator and maintainer of the IEEE TCSE online bibliography on reengineering.

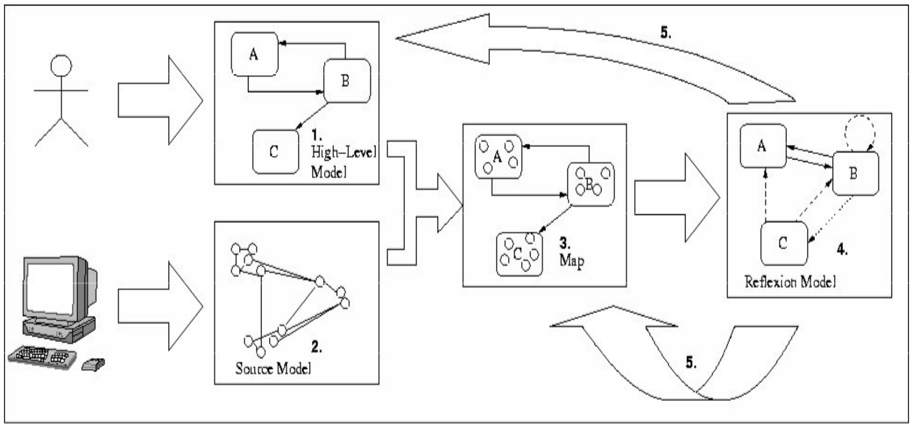


Figure 1: The Software Reflexion Modelling Process

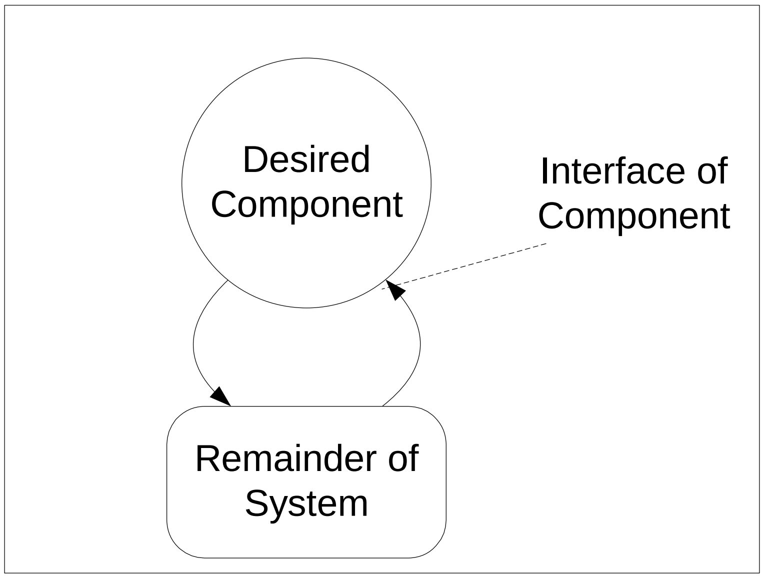


Figure 2: Encapsulating a component and making its interface explicit.

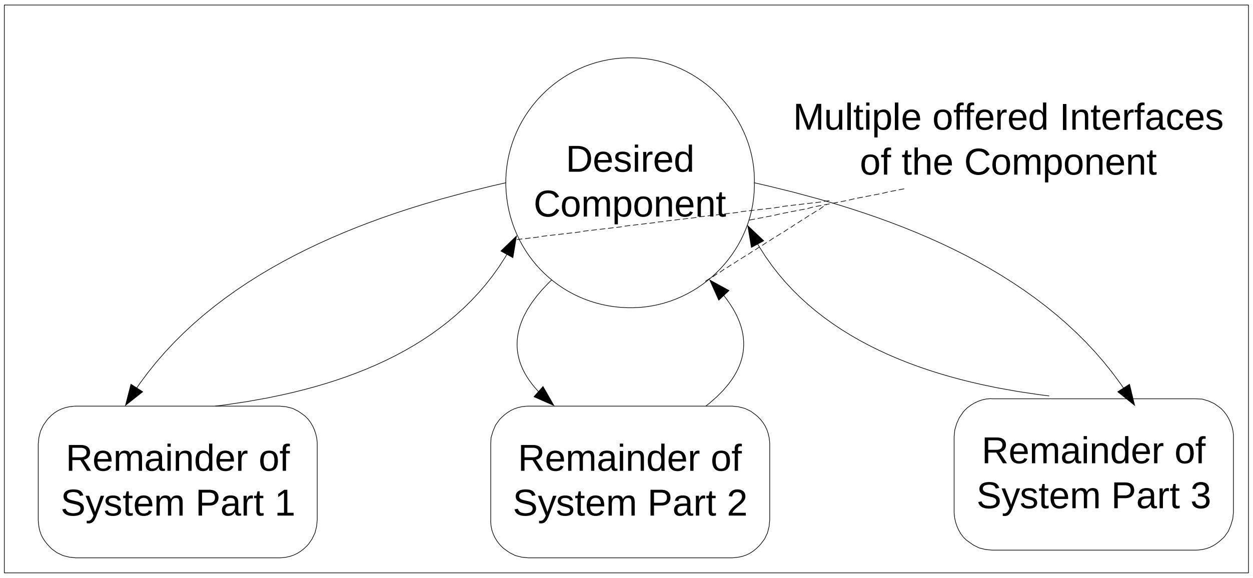
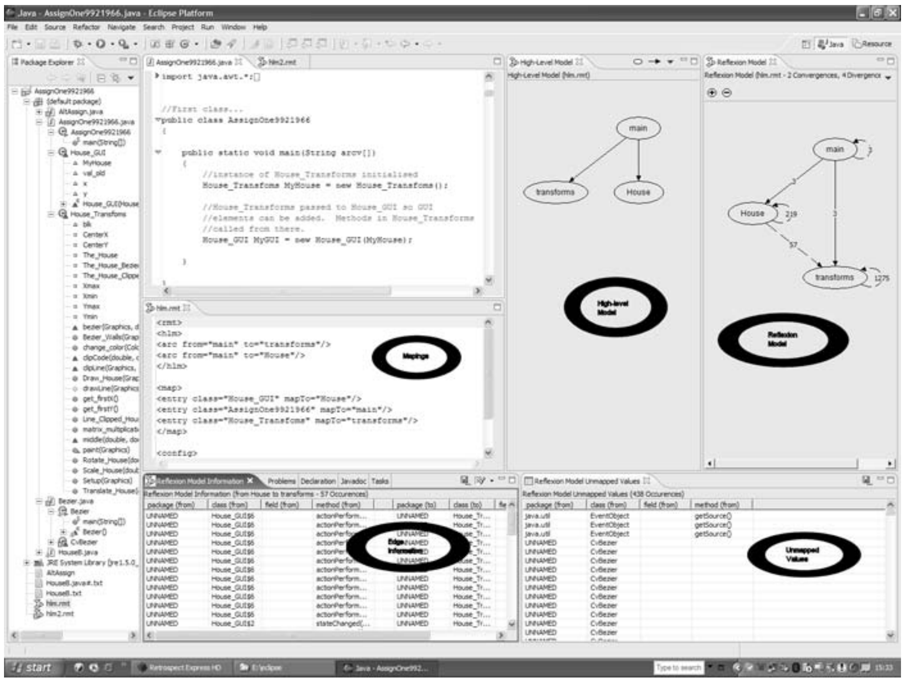

Desired Component
Multiple offered Interfaces of the Component
Remainder of System Part 1
Remainder of System Part 2
Remainder of System Part 3

Figure 3: Identifying multiple interfaces on a component using Reflexion Modelling.



```
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>
```

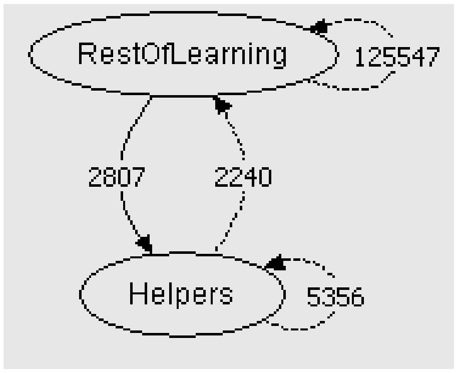
Reflexion Model Information (from House to transforms - 57 Occurences)

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]		UNNAMED	[illegible]	
UNNAMED	House_GUI$6		[illegible]		[illegible]	[illegible]	
UNNAMED	House_GUI$6		[illegible]		[illegible]	[illegible]	
UNNAMED	House_GUI$6		[illegible]		UNNAMED	[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 (438 Occurences)

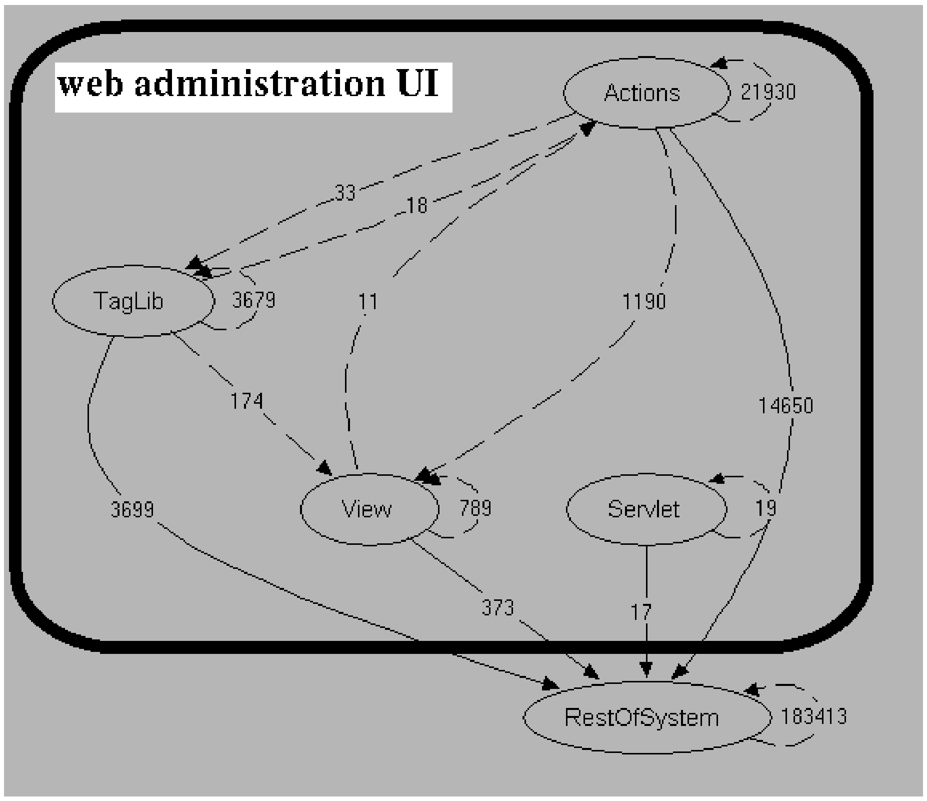
package (from)	class (from)	field (from)	method (from)
java.util	EventObject		getSource()
java.util	EventObject		getSource()
java.util	EventObject		getSource()
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED	CvBezier		
UNNAMED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

Figure 4: A screenshot of the jRMTool eclipse plug-in.



```
<map>
<entry class="^.*Helper$" mapTo="Helpers"/>
<entry class="^.*$" mapTo="RestOfLearning"/>
</map>
```

Figure 5: First iteration by the first participant.



```
<map>
<entry package="com.ibm.workplace.elearn.action" mapTo="Actions"/>
<!--<entry package="com.ibm.workplace.elearn.delivery" mapTo="Delivery"/>-->
<entry package="com.ibm.workplace.elearn.servlet" mapTo="Servlet"/>
<entry package="com.ibm.workplace.elearn.taglib" mapTo="TagLib"/>
<entry package="com.ibm.workplace.elearn.view" mapTo="View"/>

<entry class=".*" mapTo="RestOfSystem"/>
</map>
```

Figure 6: Web administration user interface component. (Note the bolded black box is not part of the tool's visual output.)

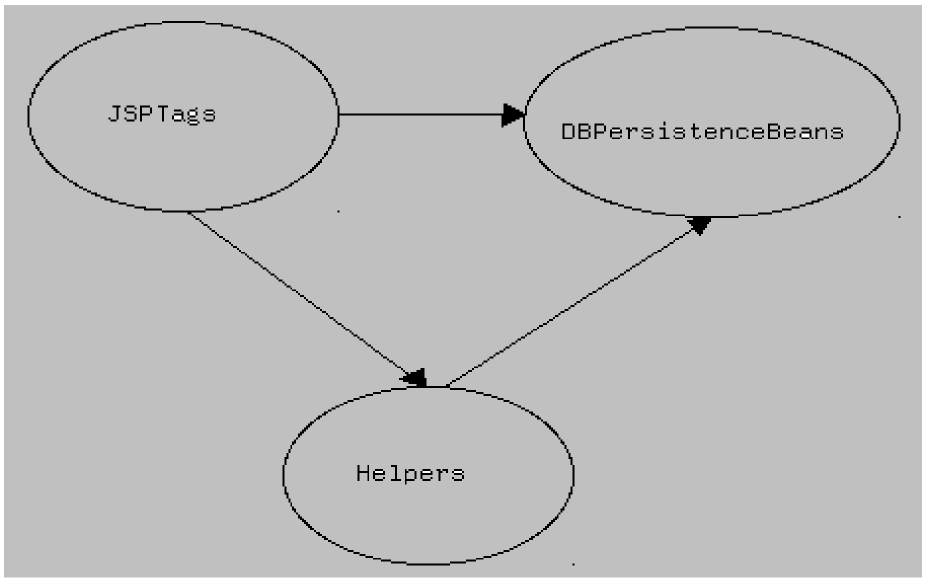


Figure 7: The Helper DTO design pattern (lower 2 edges) and the View Helper Pattern (upper edge) in the LMS.

Table 1: Coupling and Cohesion Classification by Participants

Partic-ipant	Intra-Component Coupling	Inter-Component Coupling	Intra-Component Cohesion
1	GlobalVariable (4)	Similar Data-type Parameters (2)	Communicational (3)
2	GlobalVariable (4)	Parameters control behaviour (3)	Functional (1)