The Dimension of Separating Requirements Concerns for the Duration of the Development Lifecycle

Siobhán Clarke†*, William Harrison, Harold Ossher, Peri Tarr

† School of Computer Applications,
Dublin City University,
Dublin 9,
Republic of Ireland.
+353-1-8388 702
sclarke@compapp.dcu.ie

IBM T.J. Watson Research Center,
P.O.Box 704,
Yorktown Heights,
NY 10598.
+1-914-784-7278
{harrisn, ossher, tarr}@watson.ibm.com

1. Introduction

"Separation of concerns" is a fundamental principle within software engineering, with its benefits well-documented. Looking at "separation of concerns" from the perspective of its application to each phase of the software development lifecycle, considerable research exists applying the principle within each individual phase. Some examples from the many approaches within the requirements engineering domain are multiple views [7], or separation of requirements by feature [5]. Much work also exists within the analysis and design domain, where role modelling [10], Catalysis [2], and contracts [4] are just some examples. Within the implementation domain, work on subject-oriented programming [3,8] and aspect-oriented programming [6] has identified difficulties associated with code tangling in software development. Each has provided solutions for separating code that affects many units of functionality in the system (i.e. *cross-cutting* code), with corresponding composition techniques to integrate cross-cutting and component code. More recent work on multi-dimensional decomposition [11] extends and opens the possibilities for separation of concerns with *hyperslices*, which support simultaneous decomposition according to multiple dimensions of concern, *across the full development lifecycle*.

One dimension of concern within this multi-dimensional context is the separation of individual requirements, with the maintenance of that separation for the duration of the software development process – i.e. through each of the software lifecycle phases of requirements specification, software analysis/design and software implementation. Separation of requirements concerns across the lifecycle, and co-ordinating those separated concerns across all lifecycle artefacts, requires decomposition mechanisms that allow for software design to match both requirements specifications and code. In the context of the object-oriented development paradigm, however, current object-oriented design methods suffer from the "tyranny of the dominant decomposition" [11] problem, where the dominant decomposition dimension is by object. As a result, designs are caught in the middle of a significant structural *misalignment* between requirements and code. The units of abstraction and decomposition of object-oriented designs align well with object-oriented code, as both are written in the object-oriented paradigm, and focus on interfaces, classes and methods. However, requirements specifications tend to relate to major concepts in the end user domain, or capabilities like synchronisation, persistence, and failure handling, etc., all of which are unsuited to the object-oriented paradigm.

This misalignment across the development lifecycle leads to *scattering* of the design and code of individual requirements across multiple classes and *tangling*, where individual classes in the design and code may address multiple requirements. Scattering and tangling are

* Siobhán Clarke is partially funded by IBM Ireland Ltd.

devastating from the point of view of traceability – the ability to determine how one software artefact (e.g. requirements, design, code) affects others. This leads to a host of problems, including: impaired comprehension, inability to determine how a change in one artefact affects others, increased complexity of addition, removal or modification of requirements, and potentially high-impact of change – even a small, well-contained change to requirements can affect a large part of the design and code.

No single development paradigm is appropriate for all software artefacts, and so we need to address the misalignment problem by providing additional means of further decomposing artefacts written in one paradigm so that they align with those written in another. This approach suggests that it must be possible to reify *features* within the object-oriented paradigm to permit encapsulation of feature concerns, as specified in the requirements, within designs and code. Features may be domain-specific, or cross-cutting aspects like persistence, error detection/handling, logging, tracing, caching, synchronisation etc. Our work on *subject-oriented design* is centred on this approach, and is an outgrowth of the work on subject-oriented programming, which addressed misalignment and related problems at the code level [3,8]. Like subject-oriented programming, subject-oriented design supports decomposition of object-oriented software into modules, called *subjects*, that cut across classes, and integration of subjects to form complete designs. In this position paper, we focus on the design phase, since design models can be viewed as a specification bridge between requirements and code. We are working with subject-oriented design in the context of UML [1], though it can be applied to other design languages.

2. Subject-Oriented Design

A *subject-oriented design* is an object-oriented design model that is divided into *design subjects*. A design subject encapsulates some concern in an object-oriented design. Design subjects are themselves object-oriented design models or design model fragments. They model all, and *only*, those parts of a software system that pertain to the concern they encapsulate, and they model those pieces from that concern's perspective. For example, we look here at the construction and evolution of a simple software engineering environment (SEE) for programs consisting of expressions (example more fully introduced in [11]). The SEE should include features to evaluate, check and display expressions, and permit optional logging of operations. By reifying each feature as a separate subject, we come up with different subjects that may be modelled separately: the expressions, the evaluation tool, the checking tool, the display tool, and a logging utility. Expressions are modelled as abstract syntax trees. Figure 1 presents an example of the structural design of one of these subjects – the evaluation tool.

The evaluation tool has one view of expressions. The check tool will have a different view of expressions in its design, as will the display subject and the subject handling the inherent properties of abstract syntax trees. Any full system's design will be a collection of some set of potentially overlapping design subjects. In the subject-oriented design paradigm, we introduce the notion of a *composition specification* that describes how overlapping design elements in different subjects correspond, and how they can be understood as a whole. A composition specification includes a *composition relationship* that specifies what design elements in different subjects correspond. Composition relationships may also have *reconciliation specifications*, which indicate how differences in the specifications of corresponding elements can be overcome; and *integration specifications*, which describe how to synthesise multiple corresponding design elements into a single design unit that subsumes the originals. Figure 2 illustrates the composition relationships for the SEE example.

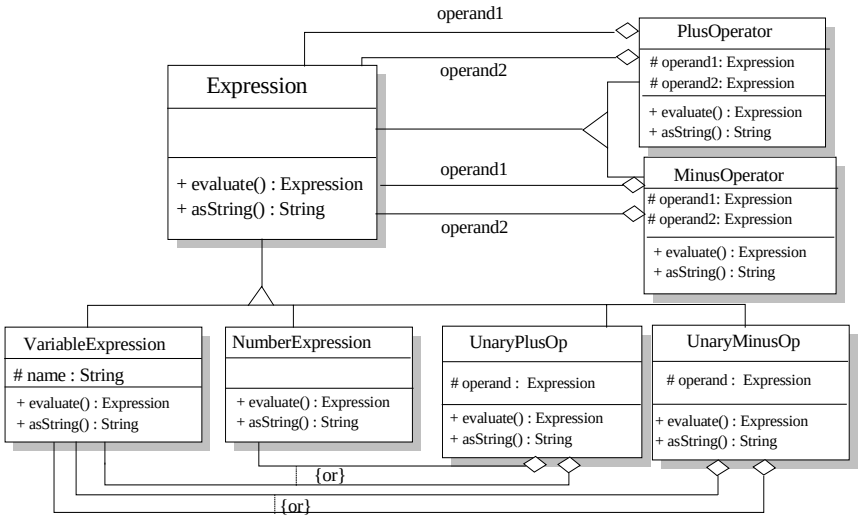

Figure 1: Evaluation Subject Structural Design

At the simplest level, correspondence between two specific design elements that are instances of the same design language construct can be specified. For example, a class *Expression* from subject *Evaluation*, and class *Expression* from subject *Check* correspond, and in a composition of *Evaluation* and *Check* a class *Expression* should also appear that is the combination of the *Expression* classes from both subjects. A more general kind of matching may also be specified for common cases. Correspondence relationships can be annotated with a matching specification that says how correspondence is to be determined between subsidiary elements (for example, a matching specification on a correspondence relationship between classes induces derived correspondence relationships among the members of those classes). Matching specifications are based on the values of the properties of the elements. For example, matching may be based on the names of the elements, or on a combination of the names and types of the elements. Matching by name is appropriate for the SEE, and is illustrated in Figure 2.

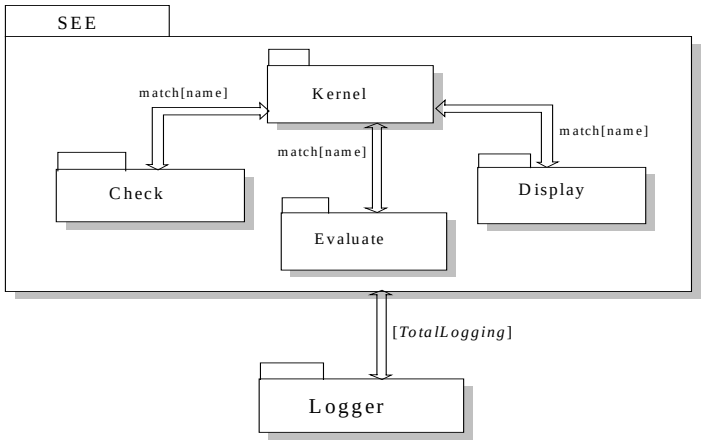

Figure 2: Composition Relationships for SEE

To specify how a design separated into subjects is to be understood as a whole, correspondence relationships are annotated with integration specifications. Integration specifications specify how to synthesise a single, composed design element from a collection of corresponding elements. Synthesis of the composed design is optional; it might be desirable, for example, to permit completeness checking or various forms of analysis. In this case, the designs can be composed as guided by the composition relationships, in much the same way as code subjects are composed in subject-oriented programming [8]. Better

alignment of the design with the code is achieved by coding each individual design subject as a code subject, and then composing the code subjects with a *composition rule* [8] derived from the composition specifications in the design.

Many different kinds of integration specifications are possible. For example, the views contained in different elements might be *merged*, such that the composed element contains all views specified for the element. The kind of integration where one design subject's view is intended to replace that of another is an *override*. This often occurs when, for example, design subjects support change requests from test teams. *Select* integration can also be specified by including some criterion in the design that can be used to select from a number of different design elements as appropriate at run-time. In Figure 2, each of the composition relationships specifies *merge* integration, as denoted by the two-way composition relationship arrow.

Various constraints may be specified for the integration of design elements, for example, ordering, dependency or mutual exclusion. As described in [9], many kinds of cross-cutting concerns affect the definition of collections of operations that span multiple units of functionality. For subject-oriented design, we maintain the focus on *operation-level joining* [9] for our integration specifications. In UML [1], interaction diagrams are used to model the dynamic aspects of a system, modelling (in many situations) the flow of control of a particular scenario. We can extend the semantics of interaction diagrams to allow them to be used by composition relationships, where integration specifications specify the flow of control between a particular subject and any subject with which it will be integrated. From our example, the intent of the logging feature for the SEE is that operations of the SEE are logged before and after execution. We model this with `beforeInvoke()` and `afterInvoke()` operations that are called before and after the execution of operations to be logged, respectively. The flow of control associated with the logging feature may be modelled with interaction diagrams that are used by the integration specification in the composition relationship between the logging subject and subjects containing operations to be logged.

3. General Purpose Design Subjects

There are many situations where features are not specific to the end-user domain of an application (like support for persistence or synchronisation), and therefore might be useful in many different situations. In the SEE example, the logging of operations is applicable to any subject that has operation elements (i.e. all subjects). The specification of the composition relationship, with reconciliation specifications and integration specifications, between such subjects and other subjects will generally have the same pattern. It is convenient to identify, name and define such *composition patterns*. Portions that might vary in different contexts can be separated out as parameters. Composition patterns can be instantiated whenever needed by naming them and supplying the parameters. An example of the use of a composition pattern is shown in Figure 2, where the composition relationship involving the Logger subject uses the pattern *TotalLogging*, specifying concisely that all operations are to be logged.

4. Conclusion

We believe that it is extremely important to separate concerns throughout the development lifecycle, and co-ordinate separated concerns across all lifecycle artefacts. Misalignment of requirements, design and code in current approaches result in a host of well-known problems, including weak traceability, poor comprehensibility, scattering, tangling, coupling, poor evolvability, low reuse, high impact of change and reduced concurrency in development. The subject-oriented design approach supports the decomposition of software designs that align with both requirements specifications and with object-oriented code. This allows all features, including those that have a cross-cutting impact on the software system, to be decomposed into subjects and designed and developed separately from the rest of the system. Code

subjects can be written from design subjects, and then composed using composition rules derived from the composition relationships at the design level.

References

[1] Booch, G., Rumbaugh, J., Jacobson, I. *"The Unified Modelling Language User Guide"* Addison-Wesley, 1998

[2] D'Souza, D., Wills, A. *"Objects, Components and Frameworks with UML. The Catalysis Approach"* Addison-Wesley, 1998

[3] Harrison, W., Ossher, H., *"Subject-Oriented Programming (a critique of pure objects)"* In Proc. OOPSLA'93

[4] Holland, I. *"Specifying Reusable Components Using Contracts"* In Proc. ECOOP'92

[5] Jackson, M., Zave, P. *"Distributed Feature Composition: A Virtual Architecture for Telecommunications Services"* IEEE TSE Special Issue on Feature Interaction

[6] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J, Irwin, J., *"Aspect-Oriented Programming"* In Proc. ECOOP'97 (Finland, June 1997) Springer-Verlag

[7] Nuseibeh, B, Kramer, J., Finkelstein, A. *"A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification"* IEEE Transactions on Software Engineering, October 1994.

[8] Ossher, H., Kaplan, M., Katz, A., Harrison, W., Kruskal, V. *"Specifying Subject-Oriented Composition"* Theory and Practice of Object Systems, Volume 2(3), 179-202, 1996

[9] Ossher, H., Tarr, P. *"Operation-Level Composition: A Case in (Join) Point"* in Proc. ECOOP'98, Workshop on Aspect-Oriented Programming

[10] Reenskaug, T., Wold, P., Lehne, O.A. *"Working with objects: The OORam Software Engineering Method"* Prentice-Hall, 1995

[11] Tarr, P., Ossher, H., Harrison, W., Sutton, S.M. *"N Degrees of Separation: Multi-Dimensional Separation of Concerns"* in Proc. ICSE'99