Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code

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

ABSTRACT

In practice, object-oriented design models have been less useful throughout the lifetime of software systems than they should be. Design models are often large and monolithic, and the structure of the designs is generally quite different from that of requirements. As a result, developers tend to discard the design, especially as the system evolves, since it is too difficult to keep its relationship to requirements and code accurate, especially when both are changing. This paper presents a different approach to designing systems, based on flexible decomposition and composition, that closely aligns designs with both requirements specifications and with code. We illustrate how this approach permits the benefits of designs to be maintained throughout a system's lifetime.

Keywords

Analysis and design methods, software engineering practices.

1. INTRODUCTION

Software design is an important activity within the software lifecycle and its benefits are well documented (e.g., [Bch94, CAB93, CD94, Jac94, MS91, RL+90, SM89]). They include early assessment of the technical feasibility, correctness and completeness of requirements; management of complexity and enhanced comprehension; greater opportunities for reuse; and improved evolvability. These benefits are seldom realised in practice in large-scale software systems, however. In our experience, many developers either do not create designs at all, create very minimal, informal design "sketches" that are discarded once system development is underway, or fail to keep their designs up-to-date as requirements and code evolve. At best, this means that developers cannot obtain the benefits of design through the maintenance and evolution phases, which constitute the majority of a software system's lifetime. The popularity of UML [BR98] might lead to more, and more widely understood, designs being created during the design phase, but creating designs during the initial design phase does not address the issue of keeping designs up-to-date later in the software lifecycle.

OOPSLA '99 11/99 Denver, CO, USA

We believe that three primary problems underlie the inability or disinclination of developers to use object-oriented designs throughout the software lifecycle. First, design models are often large and monolithic. This reduces comprehension, maintainability, and reusability. Further, monolithic designs can inhibit many useful forms of concurrency during design processes. The abstraction units in object-oriented designs—interfaces, classes, and packages—are centralised notions; only one designer at a time can work on a given unit. Centralisation means that designers are forced to commit early to the structure and contents of shared design units and concepts, which may overly constrain the set of possible designs too early and may consequently lead to significant impact of change.

Second, we believe that designs are too difficult to reuse. Designs, like code, tend to bundle too many pieces together. Complete classes designed for a particular system are typically too specialised to be of general use. If they really are more generally useful, they often include much more functionality than any given client would use, which decreases comprehensibility and, potentially, usability. Further, effective reuse requires powerful mechanisms for customisation and adaptation. The standard object-oriented mechanisms—subclassing, polymorphism, delegation and design patterns—are useful in this context, but not sufficient, particularly because they require a considerable amount of preplanning. Developers may therefore be forced to make invasive, rather than additive, changes to adapt design units, which compromises reuse and future evolution.

Finally, and perhaps most importantly, there is significant structural *misalignment* between requirements and code, with design caught in the middle. The units of abstraction and decomposition in requirements tend to relate to features and capabilities and other major concepts in the end user domain. New or changed requirements, which cause system evolution, also tend to be structured this way. Object-oriented code, however, focuses on interfaces, classes, and methods. These dramatically different structures mean that *traceability* between requirements and code is poor. Moreover, *scattering* and *tangling* may occur: a single requirement is implemented by code in many classes (scattering), and a single class contributes towards implementing many requirements (tangling). 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 relatively small, well-contained change to requirements can affect a large part of the design.

The use of modern object-oriented design languages, including UML, produces designs that align well with object-oriented code, and for good reason. As a result, however, these designs align poorly with requirements, introducing traceability and tangling problems. When a requirement is added or changed, it typically leads to widespread changes across both design models and code. Developers can be forgiven for not incurring the cost of dealing with such changes twice—once in design and once in code—and changing only the code. As long as requirements, design, and code are misaligned, we believe these problems are fundamental.

In this paper, we discuss an approach to design that addresses this misalignment problem and, in so doing addresses the other problems noted earlier that have impeded the successful use of designs. The approach is based on the flexible decomposition and composition provided by *subject-oriented programming* [HO93, OK+96]. This approach permits standard design models to be decomposed into smaller, potentially overlapping, units, called *design subjects*. Each design subject encapsulates a single, coherent piece of functionality (e.g., one or more features or components, often cutting across multiple classes), designed from its own perspective and modelled using standard, object-oriented design constructs. Unlike the units of modularity present in object-oriented design languages, design subjects may be chosen so as to align with the structure of the requirements. Design subjects may then be composed together in different ways to produce complete designs or larger-scale design fragments. Concept overlaps and mismatches between design subjects are resolved during the composition process. The composition mechanism facilitates a range of additional capabilities, including the ability to mix-and-match features, and to specify and enforce expected interactions among classes.

Subject-oriented programming allows object-oriented code to be decomposed into subjects in similar fashion. The code subjects are then composed to produce the entire system. Each design subject can therefore be refined separately to a code subject, and the details of the code composition can be derived from those of the design composition. There is excellent traceability at this level, because code and design subjects correspond directly, and within a single subject, standard object-oriented design or code are used. This traceability facilitates both evolution and "round-tripping": projecting changes in the design into the code and requirements or, for that matter, reflecting the changes made in the code back into the design and requirements.

When a requirement is added or changed, a new design subject can be created to address it. The new design subject can then be composed with the existing design, thus enhancing or replacing parts of the existing design. Then the design subject can be refined to a code subject, which is similarly composed with the existing code. The changes are localised, so there is no tangling, and traceability is preserved. In addition to thus dealing with the alignment problem, the decomposition into subjects reduces the monolithic nature of the design and allows for concurrent development, while subject-oriented composition provides a powerful mechanism for integration, evolution, customisation adaptation and improved reuse.

We describe our approach in the context of UML, though it can also be applied to other object-oriented design languages. We introduce, informally, minimal extensions to UML to allow for decomposition into subjects and specification of the relationships between them. Any existing UML design can be used unchanged as a single subject, or can be decomposed into multiple subjects if desired. Subject-oriented design, like subject-oriented programming, can therefore be adopted gradually.

The rest of this paper is organised as follows. We begin, in Section 2, by introducing an example that motivates the need for subject-like composition and decomposition. In Section 3, we describe our model of subject-oriented design. Then, in Section 4, we apply the model to the example from Section 2 and demonstrate how the division of the original design into subjects, based on the requirements, addresses many of the issues that are raised in Section 2. Section 5 describes related work. Finally, Section 6 presents some conclusions and future work.

2. MOTIVATION

To illustrate some of the pervasive and serious problems that help motivate our work, we present a running example (partially introduced in [TO+99]) involving the construction and evolution of a simple software engineering environment (SEE) for programs consisting of expressions. We assume a simplified software development process, consisting of informal requirements specification in natural language, design in UML, and implementation in Java.

2.1 The Initial Software System

We begin initially with the following requirements specification for the SEE:

> The desired SEE supports the specification of expression programs. It contains a set of tools that share a common representation of expressions. The initial tool set should include an *evaluation* capability, which determines the result of evaluating expressions; a *display* capability, which depicts expressions textually; and a *check* capability, which optionally determines whether expressions are syntactically and semantically correct. The SEE should permit optional logging of operations.

This requirements specification identifies several concerns that must be realised in the design: the SEE, expressions, the evaluation tool, display tool, check tool, and a logging utility that can be included or excluded from the environment.

Based on these requirements, we produce a UML design for the system, shown in Figure 1. The design represents expressions as abstract syntax trees (ASTs) and defines a class for each type of AST node, where each class contains accessor and modifier methods, plus methods `evaluate()`, `display()` and `check()`, which realise the required tools in a standard, object-oriented manner[1]. Logging is modelled as a separate, singleton

[1] Clearly, countless alternative designs are also possible. We chose to use a simple one here, and we will describe, later in this section, some general kinds of problems that other approaches – notably, those that use design patterns – produce.

class (`Logger`); the intent is for each AST operation to invoke `Logger.beforeInvoke()` prior to performed its action, then to invoke `Logger.afterInvoke()` just before it terminates. The Logger permits applications to turn logging on and off with its `turnLoggingOn()` and `turnLoggingOff()` methods. When logging is off, `Logger`'s `beforeInvoke()` and `afterInvoke()` methods are essentially no-ops. This permits logging to be optional, as required.

The design demonstrates some important features. The mapping from design to code is straightforward and quite direct—every concern (i.e., class) in the UML class diagram has a direct correspondent in the code. This is not unexpected, since both are object-oriented, and much of the reason for the trend toward object-oriented design is that it permits a direct mapping between design and object-oriented code.

The mapping between requirements and design, on the other hand, is extremely complex. Note, for example, the following problems:

- The SEE tools (evaluation, checking, and display), which are described as *encapsulated* concerns in the requirements, are not encapsulated in the design. In fact, these capabilities are *scattered* across the AST classes—each class contains a method that implements these capabilities for its own instances. Scattering is negative from an evolutionary perspective: the impact of a change to a single requirement, well localised at the requirements level, can nonetheless be extremely high, because that change necessitates multiple changes across a class hierarchy.
- The logging capability is realised as a first-class concern in both the requirements and the design. Nonetheless, the protocol for logging requires co-operation from each method in each AST class, to appropriately invoke `Logger.beforeInvoke()` and `Logger.afterInvoke()`. This is *tangling*—satisfying a given requirement necessitates interleaving design details that address the requirement with details that address other requirements. Tangling is a serious detriment to software comprehension, reuse, and evolution, because it is impossible to deal with the design details pertaining to one requirement without constantly encountering and having to worry about intertwined details pertaining to other requirements. For example, it is difficult both to determine how a change to the logging requirement will impact the design, and to effect such a change *additively*, rather than *invasively*

Scattering and tangling are also devastating from the point of view of *traceability*: the ability to determine readily how a piece of one software artefact (e.g., requirement, design, code) affects others. Traceability makes it possible to look at a change to a requirement, and to find those parts of the design and code details that are affected by the change. Traceability is essential to keeping requirement and design documents up-to-date with respect to evolving code. Without it, these documents are likely to become obsolete and useless, since, when it is difficult to determine how a proposed change to one will impact the other, changes may not be propagated across them consistently, or at all.

These problems, and others present in this design, occur because the concerns identified in the requirements, which are based on *features* of the SEE, are different from those used to modularise the design, which are the *objects* and *classes* that implement the SEE. Thus, the requirements concerns generally are not, and cannot readily be, encapsulated in the design. This is different from the relationship between design and code, where the respective sets of concerns are very similar. In the process of creating designs from requirements, UML and other object-oriented design formalisms and languages necessitate a transition from feature (or other) concerns to object concerns.

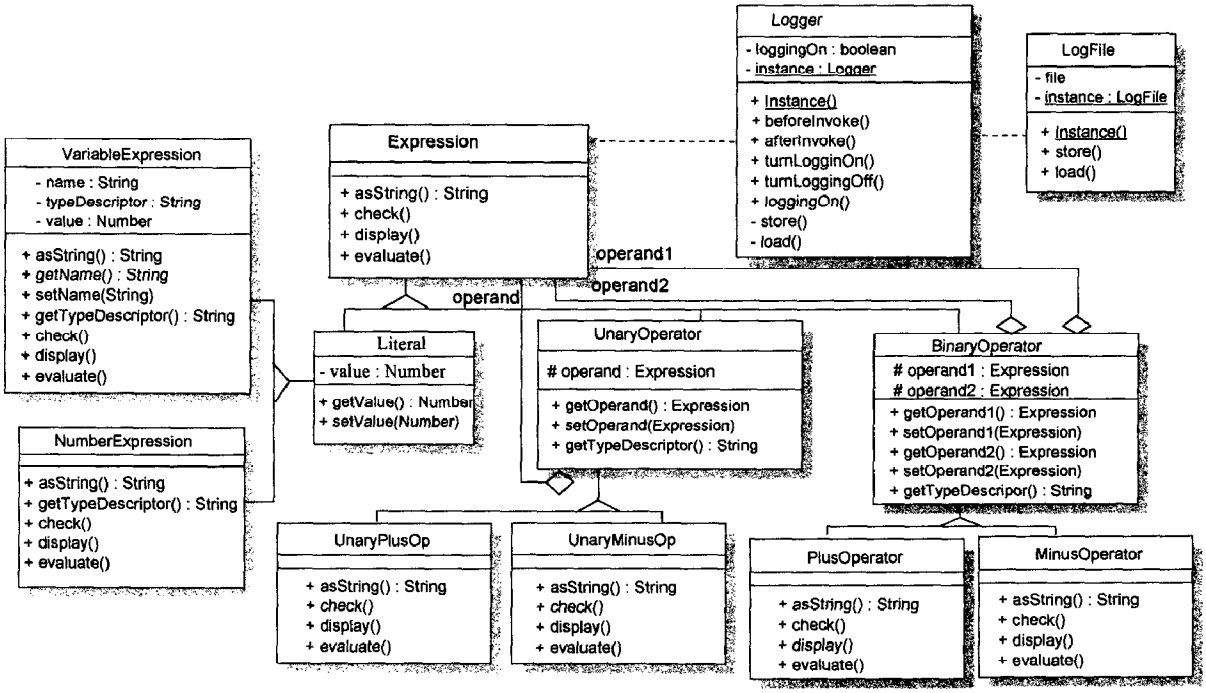

Figure 1: UML Design for SEE

This transition essentially results in the discarding of the encapsulation of those concerns identified during requirements specification in favour of concerns mandated by the design and coding paradigms. In achieving a close tie to code, object-oriented design loses one of its two "faces": the one that connects it with requirements. Scattering and tangling are, in fact, symptomatic of this mismatch. Some earlier design paradigms, notably those that permitted functional decomposition, exhibited the opposite problem: they facilitated the production of designs that aligned well with requirements, but that did not align well with object-oriented code.

This point is particularly important. ***In general, most design paradigms are not sufficiently powerful to permit designs to wear both faces*** —they allow the design to align with either the requirements or the code, but not both. Thus, designs fail to achieve one of their primary purposes: to promote *traceability* by bridging the gap between requirements and code. Traceability is an important prerequisite to evolution, as is encapsulation, which aids in limiting the impact of any given change. Note, for example, that it is difficult both to determine how a change to the logging requirement will impact the design, and to effect such a change additively, rather than invasively. Limited traceability and encapsulation, as is present in the SEE design, result in reduced evolvability. Consequently, they also result in the eventual obsolescence of requirements, design or both, since changes may not be propagated across them consistently if it is difficult to determine how a proposed change to one will impact the other.

The misalignment of requirements and design also has ramifications for the design process itself. For example, designers are limited in their ability to work concurrently on the design (and, in fact, on the code), to a much greater degree than when producing a requirements specification. Specifically, it would be desirable to have a compiler expert work on the AST representation itself, a user interface expert work on the design of the display feature, etc. The scattering and tangling of these features results, however, in interdependencies across these features and across the classes that hampers concurrent design and implementation. Classes are inherently centralised notions, so it is also often fairly difficult to permit concurrent development of the same classes. Further, while the logging capability can be designed independently of the AST classes, all of the other developers must be aware of its presence and must design with it in mind. For the same reasons, all of the SEE tool designers must wait for the "core" AST to be defined before they can work effectively even if designers could work in parallel on features. This opens the door to a variety of errors, and it can result in delays while designers wait for one another.

2.2 An Evolving Headache

After using the SEE for some time, the clients request the inclusion of different forms of optional checking; initially, they ask for a def/use checker and a style checker that verifies conformance to local naming conventions. The check feature thus becomes a "mix-and-match" capability—clients can choose any combination of syntax, def/use, and/or style checking to be run on their expression programs when they invoke the check tool.

This change in requirements is additive—it need not affect any other requirement. At the design level, however, the change is not as straightforward, since the check feature is not encapsulated as a concern in the design. In fact, this change necessarily affects all AST classes in the design. One possible approach to designing the new forms of checking would be to create new subclasses of the AST classes, where a given subclass overrides the original (syntax) `check()` method with one intended to provide def/use or style checking for a particular kind of AST class. Clearly, while this approach is non-invasive, it is completely impractical, as it results in combinatorial explosion of classes with each new feature. A better approach is to use the Visitor design pattern [GH+94] to represent checking, and to provide different visitors that correspond to the different kinds of checking.

The visitor approach, which is depicted in Figure 2 facilitates "mix-and-match" without combinatorial explosion. It requires, however, an invasive change to all of the AST classes, to replace the `check()` methods with `accept(Visitor)` methods. The use of visitors also introduces a second complication. The logging feature requires the visitors to invoke `Logger.beforeInvoke()` and `Logger.afterInvoke()` appropriately, further increasing the scattering and tangling problems associated with this feature.

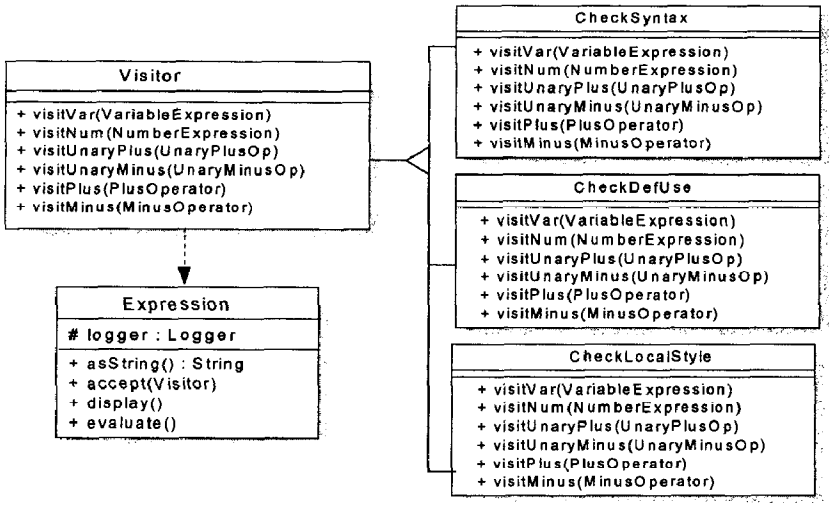

Figure 2: Using Visitor to Separate Check Functions

This evolutionary change, which appeared to be straightforward and additive from the clients' perspective and from its impact on the requirements, demonstrates, in a microcosm, the spectrum of problems resulting from the misalignment problem. Scattering and tangling lead to weak traceability and poor encapsulation of requirements-level concerns within the design, and subsequently, the code. They also make the propagation of requirements changes to design and code very difficult and invasive. It is even difficult to determine which design elements are affected by a given requirements change. The level of effort needed to propagate changes from requirements to design is much greater than the effort to propagate the changes from design to code, precisely because of the misalignment.

2.3 When the Solution is the Problem...

Countless other design approaches are possible for the SEE, and some of them address some of the issues that have been raised. For example, the judicious application of design patterns might help solve some of these problems. While it is impossible to elaborate the possible design approaches (with or without design patterns) exhaustively, we explore briefly some of the design pattern alternatives to illustrate why neither they, nor other approaches, address the whole problem.

Visitor: The initial use of the visitor pattern to model checking would have facilitated greatly the addition of the new checkers. Note that this is the case precisely because visitors provide encapsulation of features, which results in better alignment of design with requirements. While visitors promote some forms of evolution, they hinder other forms. For example, adding a new type of expression, like assignment, is simple in the original design shown in Figure 1, but it would necessitate invasive changes to *all* visitors [GH+94].

Observer with Factory: To reduce the coupling between the logger and the AST classes, we might have chosen to accomplish logging via observers. Since observer operates at the *instance* level, rather than the *class* level, it would be best to use a factory to create objects, since the factory can decide transparently whether or not to register the logger with any newly created objects. This approach would achieve the looser coupling. Observer is, however, an extremely heavyweight solution that incurs high overhead, in both complexity and performance. Further, it does not improve the scattering problem, as AST methods must notify any observers, thereby scattering the implementation of logging across all the AST classes. Used in conjunction with visitors for the AST tools, the design for the SEE becomes significantly larger and more complex, with many more interrelationships among the classes to be represented and enforced.

Decorator with Factory: As an alternative to observer, we could choose to represent logging using the decorator pattern, where decorators perform logging (if desired). Decorator, like observer, helps to reduce coupling, and, unlike observer, it reduces tangling by segregating logger notification code into separate, decorator objects. Again, since decorators operate on a per-instance basis, the use of a factory would be prudent. Unfortunately, the decorator solution is significantly more problematic than the observer solution, because of the *object schizophrenia* problem. That is, to ensure that logging occurs consistently, it is necessary to ensure that *all* messages to all objects go through the decorator, *not* directly to the object itself. Once a method on an object is invoked, however, that method may invoke others, which, in turn, must go through the decorator. This means that the object must know about its decorator(s), which introduces a new form of coupling and tangling (i.e., each class must include code to implement the interaction with the decorator).

Summary: Design patterns and other design approaches can help to alleviate some, but not all, of the problems we have identified. Unfortunately, in ameliorating some problems, they introduce other problems or restrictions [GH+94, Vli98]. The need to preplan for change—which we see in the use of all design patterns, since designs and code must be pre-enabled with the pattern to avoid subsequent invasive changes to incorporate them—is especially problematic. It is impossible to anticipate every kind of change that might be required; even if it were possible, flexibility always comes at a cost in terms of conceptual complexity and/or performance overhead, as the visitor, observer, and decorator patterns demonstrate. Enabling for some forms of change inhibits other kinds of change—for example, introducing visitors will promote the future addition of new types of checkers, but it greatly complicates the addition of new types of expressions.

Thus, while design patterns and other design approaches are very useful, they cannot address the issues we have raised—their use results in the exchange of one set of problems for another. In some cases, the new set of problems is acceptable, but in others, it is not. As long as the misalignment problem exists, its consequences—weak traceability, low comprehensibility, scattering, tangling, coupling, poor evolvability (including high impact of change and invasive change), reduced concurrency in development, etc.—will be present.

3. MODEL

Two general approaches exist to addressing the misalignment of requirements, design, and code. One is to impose the same development paradigm on *all* software artefacts. This is, in fact, precisely the approach that has been used to provide closer alignment between designs and code—both are written in the object-oriented paradigm. This approach is not appropriate when applied to requirements specifications, however, as requirements deal with concepts in the *user's* domain, while designs and code deal with concepts in the *programming* domain.

The other approach to addressing the misalignment problem is to provide additional means of further decomposing artefacts written in one paradigm so that they can align with those written in another. This approach suggests, for example, that it must be possible to reify *features* and other kinds of concerns [TO+99] within the object-oriented paradigm to permit encapsulation of feature concerns, as specified in the requirements, within designs and code. We have chosen to adopt this approach, in recognition of the fact that different paradigms are appropriate under different circumstances, so that homogeneity, while appealing, is likely to be inadequate. Our approach, which we call *subject-oriented design*, is an outgrowth of the work on subject-oriented programming, which addressed misalignment and related problems at the code level [HO93, OK+96]. 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.

A *subject-oriented design* is an object-oriented design model that is divided into *design subjects*. Each design subject separately describes that part of a system or component that pertains to a particular *concern* [Par72]. *Composition relationships* describe how these design subjects relate to one another, and hence, how they can be understood together as a complete design.

3.1 Design Subjects

Each design subject is an object-oriented design model that encapsulates the design of just that part of a system that pertains to a particular concern. In the context of alignment of requirements, design and code, a design subject might encapsulate those design elements whose purpose is to satisfy a specific requirement, or perhaps a coherent set of related requirements.

For example, the requirements specification for the expression SEE described in Section 2 identifies a requirement for a "display" feature. In the UML class diagram shown in Figure 1, operations to support the display feature appear in all of the expression AST nodes; thus, the display feature cuts across many UML classes, reflecting the misalignment problem. To permit better alignment of design with requirements, it is desirable to encapsulate the display operations into a single design unit.

Conceptually, a design subject can be written in any design language, but our focus in this paper is on UML. A UML design subject can contain any valid UML diagrams, but we deal only with class and interaction diagrams in this paper. Application of this approach to other design languages and to the other UML diagrams remain interesting issues for future research. The kinds of requirements whose designs can be described in design subjects are many and varied. They include units of functionality, features [TF+98, GF+98], so-called *cross-cutting* requirements, like persistence or distribution, that affect multiple units of functionality, and variants (requirements that identify particular selections in a space of choices, such as of target system or level of capability). Design subjects can also encapsulate concerns of other kinds, such as units of change, or subdomains [TO+99].

Design subjects thus provide an additional means of decomposing systems, complementing those provided by the other UML diagrams. They permit the encapsulation of all, and only, those design elements that pertain to a given concern. Whereas the design elements in a conventional UML design model must be defined completely with respect to the entire system, the design elements in a design subject need only contain those details that are relevant to the concern it encapsulates.

It is possible—indeed, expected—that some of the same concepts may be relevant to multiple design subjects. For example, both the "display" and "check" features require knowledge about how to traverse expression ASTs; thus, if they were each modelled as a separate design subject, they would both include their own views of the child attributes of AST classes. These views may, but need not, be identical; one might, for example, model the links to each child as a separate structural relationship, while another might model the links to all children as a single one-to-many relationship. Design subjects may therefore *overlap*, and they may include some differences in their views of overlapping parts. This is a strength of design subjects—they permit each of the different parts of a system under design to model the same concepts in whatever way is most appropriate to that subject's view and purpose. Differences in views can be identified and resolved, using *composition relationships* (discussed in the next section), as part of the design process. With UML, design elements that support the same concept, but have different views that necessitate different specifications, must be specified separately. However, since there is no means of synthesising a complete design from incomplete pieces in UML, such elements will remain separate throughout the design cycle.

The criteria for choosing a set of design subjects into which to decompose a system are much the same as for any design decomposition activity. The decomposition of a design into subjects is generally based on attempts to satisfy different system and software engineering goals and requirements, including reusability, evolvability, traceability, comprehensibility, etc. We do not prescribe any particular selection criteria or design process—many are appropriate. In this paper, we emphasise design subjects that match requirements, thus addressing the misalignment problem. We also note that design subjects are particularly well suited to languages that produce views of objects that overlap and "cut across" one another. Two exemplars are role modelling [RW+95] and use case analysis [BR98].

Design subjects are represented as UML packages with design elements contained within them, either directly, or by reference to other parts of the overall design.

3.2 Composition Specification

As noted above, design subjects support the decomposition of systems into *potentially overlapping* design models. Overlap occurs whenever two design subjects describe their own views of the same concepts. For example, the subjects encapsulating the "display" and "check" features both model views of Expressions; one describes Expressions to have a `display()` operation, the other a `check()` operation, but they share the concept "Expression." The ability to describe overlapping design models provides considerable decomposition and encapsulation power. It also means, however, that understanding the system as a whole requires the identification of corresponding elements from different design subjects, and understanding of how these elements fit together to describe the shared concept fully.

We therefore enhance UML to support composition specification. We introduce a new kind of relationship called a *composition relationship*, which identifies corresponding design elements in different subjects, and may be annotated with optional *reconciliation* and *integration specifications* which describe how the corresponding elements are to be understood as a whole. Many of the details derive from the composition rules used for specifying composition of code subjects [OK+96].

3.2.1 Composition Relationships

Composition relationships between two or more design elements in different subjects denote the fact that those design elements *correspond*, in the sense that they represent views of a single concept, and may be composed into a single entity. Composition relationships can be described between design elements of any kind (e.g., classes, operations, design models, etc.), but all elements in a given relationship must be of the same kind. This new kind of relationship that we introduce into UML coexists with all other UML relationships.

For example, if we realise the display and checking requirements as design subjects, both subjects would contain classes that model the concept "Expression" in different ways. These classes

therefore correspond, and this correspondence is specified by means of a composition relationship.

Where there is considerable overlap between design subjects, specifying all appropriate correspondences between design elements individually would be a great deal of work, and would lead to highly cluttered designs. Instead, we exploit uniformity, noting that within a particular context, one can often characterise a multitude of correspondences succinctly by means of a *matching specification*, which is a rule for computing correspondences. For example, a common strategy for computing correspondences is based on matching the *names* of design elements, as in the case of the display and checking subjects. Matching specifications, such as *match[name]*, are attached to higher-level, explicit correspondence specifications, and apply to all design elements within the corresponding elements.

For example, all classes, operations and instance variables in the SEE design subjects are to correspond by name. This can be specified by means of a *match[name]* specification attached to a composition relationship at the highest level: between the design subjects themselves. These general matching specifications can be overridden as needed to describe exceptions to the general rule [OK+96].

Composition relationships are shown in UML diagrams as fat *composition arrows*; the positioning of the arrowheads depends on the kind of integration involved (described below). Matching specifications are shown as annotations on composition arrows. Correspondences implied by matching specifications are not shown as separate arrows.[2]

3.2.2 Reconciliation Specifications

Corresponding design elements in different subjects may represent either the same or different views of a concept. When the views are the same, composition relationships identifying the corresponding elements are sufficient. When the views are different, however, it is also necessary to describe how the differences among the corresponding elements are to be *reconciled*—that is, how the different views relate to one another. This is done by means of *reconciliation specifications* attached to composition relationships. In UML diagrams, they are shown as annotations attached to composition arrows.

The SEE example, as presented, does not require reconciliation specifications. As we shall see, this is common in cases where design subjects are produced together as part of an integrated design effort. Suppose, however, that users had imposed an additional requirement—to support textual display in addition to graphical display. In the overall design, that feature would be specified as a subject separate from the graphical display subject. In the textual display subject, the expression class would be likely to include an operation `display(Stream s)`. In defining the correspondence, this textual `display` operation would correspond to the graphical display subject's `display()` operation. This would result in a signature mismatch between the two `displays` that would have to be reconciled. The reconciliation could be accomplished, for example, by specifying a default stream for the textual display.

[2] It might, of course, be desirable to be able to view the full set of correspondences. Appropriate tool support could provide this capability readily.

3.2.3 Integration Specifications

Composition relationships also specify how the collection of corresponding design elements is to be formed from its elements. In the example above, for instance, where the two "display" operations correspond, the designer may need to specify whether one of the specifications completely defines the desired result or if both are to be satisfied. If both are to be satisfied, the designer must indicate in what order the two specifications should be satisfied and, in the case of a value-returning function, how a single return value should be computed. *Integration specifications* may also be part of a composition relationship to specify these details.

Although *integration specifications* specify how to synthesise a single, composed design element from a collection of corresponding elements, the synthesis need not be carried out as part of the design process. 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 is composed in subject-oriented programming [OK+96]. By focusing on the issue of the formation of the result from the separate subjects' specifications, the integration specifications facilitate an understanding of a complete design. The fact that it allows the description of the composed design to rise above the details of each design also simplifies the forward projection of requirements' changes into changes in the design.

Three common types of integration specifications are called: *merge*, *override*, and *select*.

"Merge" integration: *Merge* integration is, perhaps, the most commonly useful form. When subjects represent different *optional* features of a system, or when different teams have concurrently designed various subjects, the merge integration specification is generally appropriate. For the SEE example, merging the "display" and "check" subjects, each of which contain definitions for the Expression class, yields a single specification for Expression that contains both the `display()` and `check()` operations. Where there is no overlap of concept definitions between subjects, the merge integration denotes a simple union of all the design elements in the subjects being merged. Where overlap exists, and composition relationships define correspondences (either explicitly, or implicitly via matching), the meaning of a merge integration for different kinds of corresponding design elements varies in ways appropriate to the kind of element.

Looking at some design language constructs from UML, for *classifiers* and *attributes*, ***merge*** indicates that the composed design contains a single element whose elements are obtained, by aggregation or identification, from the subjects connected by the composition relationship. Merging corresponding *operations* indicates that the specification of the (unified) operation results from the aggregation of the specifications of those operations in all of the corresponding subjects. This aggregation has consequent implications on other diagrams in the design, most of which are straightforward. For example, in interaction diagrams, a call to an aggregated operation implies a call to each of the operations in the aggregate. Another example is illustrated by the treatment of *dependency* and *association* relationships. Elements in the composed subject are dependent (dependency relationship), or are

associated (association relationship) if they are dependent/associated within any of the subjects being composed. The construction and display of a synthesised composite design is especially useful for illustrating such deeper consequences of a composition relationship.

At the opposite extreme, automatic synthesis of a single set of *generalisation (inheritance)* relationships for a composite design is both *difficult* and *dubious*, even though the manual definition of the merged inheritance relationships can be both meaningful and useful. A simple merge of hierarchies that have classes in common has the potential of resulting in inheritance anomalies (e.g., cycles), making it *difficult*. Within any one subject, generalisations can be defined to indicate that elements inherit properties from the other elements. But, for purposes of composition, each element can be taken as having a complete meaning to itself, regardless of whence that meaning came[3]. This means that an element resulting from the composition is completely specified without needing generalisation relationships, making them *dubious*. The difficulty with anomalies is, in fact, a symptom of the dubiousness of an automatic synthesis. For example, an element and one of its specialisations within one subject might correspond to two unrelated elements in another subject. In a case like this, it cannot be determined automatically what generalisation relationship, if any, is appropriate in the composed subject. This problem with inheritance relationships also occurs in other generalisation relationships.

In most cases, however, it is possible to re-synthesise generalisation relationships in the composed subject from those in the subjects being composed. This can be accomplished with appropriate tooling whose details, in the case of inheritance relationships, may be target-language specific. Such re-synthesised relationships can be used as an aid to the composition designer in understanding the composed subject.

Merge integration is denoted by a multi-headed composition arrow, with the design elements to be merged at the ends of the arrow.

"Override" integration: A subject can be created which, either by serendipity or with knowledge of some other subject, is intended to extend, customise or in some way change the behaviour specified by other subjects. In such cases, the system designer specifies that the new subject's design elements *override* those to which they correspond in the other subject in an integration process. The composition relationships indicate which corresponding design elements are to be replaced. In the case of override integration, reconciliation specifications are not required or used. For *classifiers*, *attributes* and *relationships*, the specifications of design elements in the subject to be updated are nullified in favour of the corresponding specifications of the elements in the overriding subject. For *operations*, the specification of an operation comes solely from the specification of the overriding operation. As with *merge* integration, *override* integration may have implied consequences on other UML diagrams. For example, any interaction diagrams containing operations that are replaced are effectively changed by using the full interaction definition of the operation in the replacing subject instead of the full interaction definition of the operation to be replaced. Complete interaction diagrams may themselves be replaced.

The discussion of the usefulness of displaying synthesised interaction diagrams and of the issues surrounding the generalisation relationship is the same for *override* as they are for *merge*.

Override integration is denoted by a one-way composition arrow, with the arrowhead at the end of the element to be overridden.

"Select" integration: Sometimes corresponding operations describe different ways of performing some functionality. For example, the SEE has two different style checkers, each of whose "check" operations performs the checking in a different way. Merge specifies that, whenever a user requests checking, **both** kinds of checks are to be performed. Override integration chooses **one**, at design time. A third alternative is to specify in the design some criterion that can be used to select the appropriate one to execute at run time, whenever an actual "check" call is made. The criterion, in this case, might be the value of an environment or preference variable set by the user to indicate which kind of style checking should be used. In general, it might be the type or value of anything accessible to the called operations, such as parameters and "global" variables.

Select integration thus provides multiple dispatch, even though UML itself does not model multiple dispatch. It is appropriate in situations where:

- Different subjects describe different behavioural *variants*.
- All variants should be available at run time.
- Only one (or a subset) of the variants should be executed at run time.
- The choice of which to execute is made upon each call, based on some runtime criterion.

If all variants should be executed, merge integration should be used; if one variant is to be chosen at design time and the others excluded entirely, override integration should be used.

Select integration is denoted by a multi-headed composition arrow, with the design elements to be included in the composition at the ends of the arrow. Annotations must be attached to the branches of the arrow to specify the selection criteria.

3.3 An Open-ended Set of Specifications

There are many ways to specify correspondence, reconciliation and integration, of which we have described only a few, and developed only a few more. Our goal is to describe a useful basic set, but that set cannot be complete. An open-ended approach is therefore needed that allows sophisticated users to define their own specifications. Tools that support composition of implementations or the visualisation of synthesised equivalents of a composite design can support an open-ended set, a framework with pluggable pieces for implementing different approaches, as in the subject-oriented programming implementation [OK+96]. The extension of UML to include open-ended concepts (whether composition specifications or simply types of relationships) is beyond the scope of this paper.

[3] In subject-oriented programming, the process of making all inherited information explicit is called *flattening*. Definitions are copied down from where they are defined into each class that inherits them [OK+96]. Flattening is also used to define the meaning of specialised elements in subject-oriented design.

3.4 Composition Patterns and Guides

In some contexts, composition relationships are quite complex, potentially involving multiple correspondences among multiple subjects and their constituent elements, with reconciliation specifications and detailed integration specifications attached to them. Common patterns of such composition relationships might emerge, or be expected, when the same or similar subjects are used in different contexts. It is convenient to identify, name and define such *composition patterns*. Portions that might vary in different contexts can be separated out as parameters. Relationship patterns can thus be thought of as lambda abstractions over sets of composition relationships. They can be instantiated whenever needed by naming them and supplying the parameters. This provides an important level of abstraction when specifying composition relationships; comprehensibility is improved and duplication of detail is avoided, thereby reducing the probability of errors.

A collection of design subjects might represent the design of not just a single system, but of a family of related systems. In this case, some design subjects might be alternatives, whereas others are written to collaborate. In addition to composition patterns specifying useful compositions, it might also be valuable to specify *composition guides*, identifying properties of composition relationships that should normally hold for any composition. For example, a composition guide might specify that only one of a set of alternative design subjects should normally be included in a composition.

The author of a subject, or a coherent set of subjects, often has some common forms of use and some guides in mind. Composition patterns and guides provide a means for these to be expressed for the benefit of future users of the subject(s), and might be supplied by the authors along with the subjects themselves. Users then have a number of choices. They can use one or more of the patterns exactly as specified, by instantiating them. They can also, if desired, specify additional composition relationships that enhance or override details specified in the patterns. Alternatively, they can avoid using the patterns entirely, and specify their own composition relationships. Composition patterns and guides, provided by the authors of design subjects, are intended as aids, not as mandates. The author of a subject cannot know all the ways in which that subject will be used in the future. The developer responsible for creating an integrated design from a number of subjects, on the other hand, is expected to understand enough about all the design subjects to be able to specify details of their relationships. This developer therefore has the last word, including the ability to override composition constraints. Such overriding must be done explicitly; in its absence, constraint violations are considered to be errors.

The use of composition patterns and constraints is illustrated in Section 4. Further details are beyond the scope of this paper.

4. APPLYING THE MODEL

To illustrate how subject-oriented design addresses the misalignment problem and achieves better, more flexible system design, we apply it to the construction and evolution of the expression SEE.

4.1 The Initial Software System

To align design with requirements, we define one design subject per feature identified in the requirements specification. Thus, we define a *kernel* subject supporting the representation of expressions; an *evaluation* subject; a *check* subject; a *display* subject; and a subject responsible for logging of operations. We discuss these subjects individually in Section 4.1.1., and their integration by means of composition relationships in Section 4.1.2.

4.1.1 Design Subjects

Figure 3 shows the Kernel subject. As in the original design, we represent expressions as abstract syntax trees. Notice, however, that the kernel design subject only defines the AST classes and their primitive accessor methods—it does not tangle support for any of the required SEE features with the expression representation.

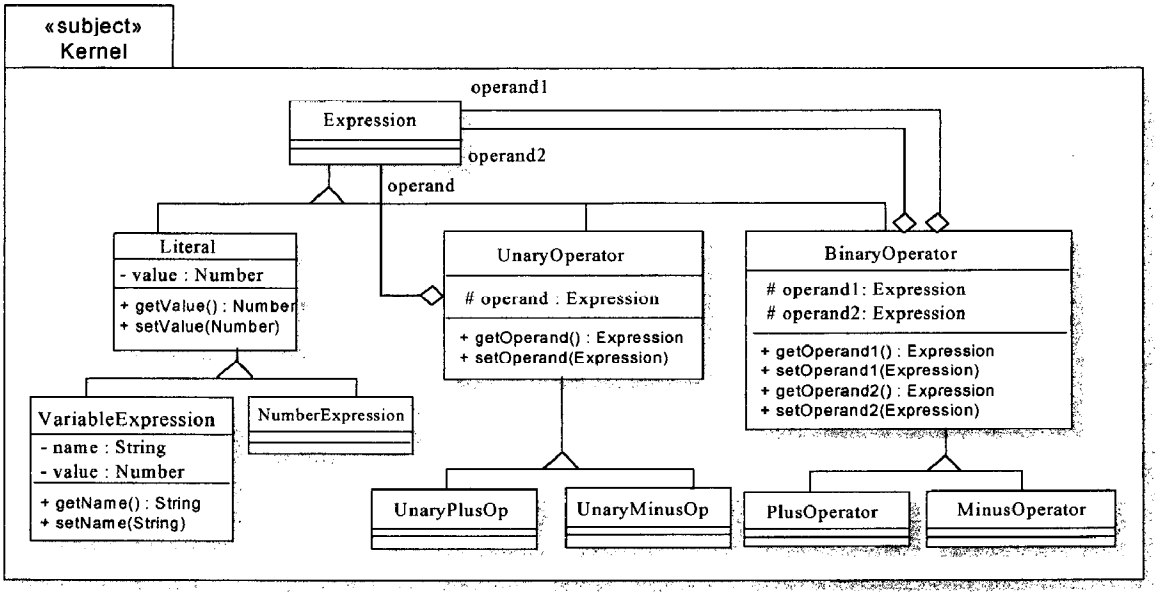

Figure 3: Kernel Subject Structure Diagram

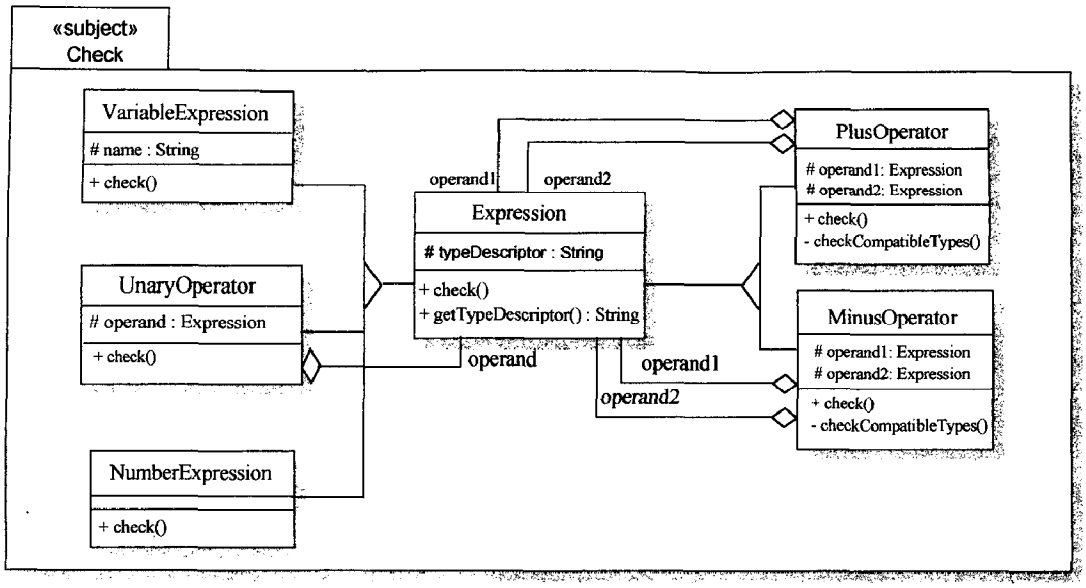

Figure 4: Check Subject Structure Diagram

Similarly, the design subjects for each of the other required features, a representative example for the Check subject which appears in Figure 4, are "pure"—one subject each supports checking, evaluating and displaying expressions, and operation logging. For space reasons, we show the structure diagrams of only the Kernel subject (Figure 3), the Check subject (Figure 4) and the Logger subject (Figure 5).

These design subjects illustrate some important features of subject-oriented design. First, the kernel, check, evaluate and display subjects realise and encapsulate their respective SEE tools in a standard object-oriented manner, with appropriate methods in each of the AST classes. While there is unavoidable scattering of tool support across classes within each subject, encapsulation is nonetheless achieved by each subject *as a whole*. This provides clear alignment of the design to the requirements, as each subject represents the design of a particular feature in total, and contains no reference to any other feature; all cross-feature interactions are specified by means of composition relationships. Encapsulation of the logger feature also avoids tangling of logger functionality with the rest of the design.

A second important feature of this subject-oriented design approach is that each of the subjects specifies its own view of overlapping design elements. For the SEE, the AST structure of an expression is manifested in every subject, except the Logger subject. Yet each subject defines a slightly different view of the AST class hierarchy; for example, the Check subject does not define the `BinaryOperator`, `UnaryPlusOp`, and `UnaryMinusOp` classes in its hierarchy, as they are not affected directly by the checking methods. Similarly, the Evaluation subject and the Display subject, not illustrated here, do not include the `BinaryOperator` and `UnaryOperator` classes. The designers of the individual subjects need not be concerned about these differences, as identification and resolution of any differences is supported by composition relationships. This increases the amount of concurrent design that is possible. It also enables each subject to include whatever model of AST it finds most appropriate to its task, rather than requiring premature commitment to a single AST definition. This property helps to improve the individual subjects, to insulate each designer from the effects of changes in other subjects, and to eliminate coupling across subjects.

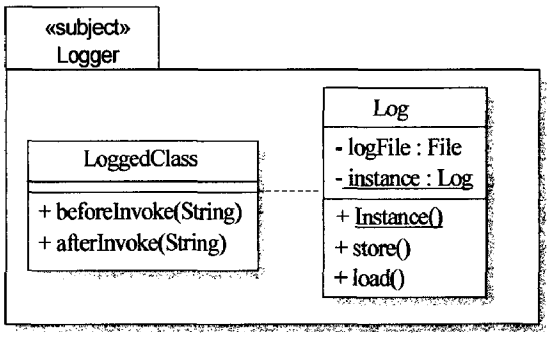

Figure 5: Logger Subject Structure Diagram

The Logger subject illustrates another interesting feature of subject-oriented design. The SEE requirements specification imposed a requirement for optional logging of operations. The ability to log operations is not particular to expressions or ASTs, however, so the logger subject can be designed independently of the operations to be logged (Figure 5). Composition relationships will establish connections between the SEE subjects (or any others) and Logger, thereby specifying exactly when logging is to take place. This approach has the advantage of separating the design of logging from that of the SEE, addressing the tangling problem that manifests itself primarily in the behavioural specifications for operations that are to be logged (not shown). It also results in a subject that is generally reusable for any application that requires logging of operations.

This is a good scenario for the use of composition patterns. We might define a pattern called *TotalLogging* as an abstraction for composition relationships. It specifies that when Logger is composed with other subjects, all operations of all classes within those subjects are to be logged by invoking Logger's `beforeInvoke()` method before execution of each operation,

and Logger's `afterInvoke()` method immediately afterwards. This pattern, provided by the author of Logger along with the subject itself, makes specification of standard logging especially simple, as we shall see later in this section.

4.1.2 Composition Relationships for Design Synthesis

Taken together, the collection of design subjects described above actually defines a *family* of SEEs. That is, the set of features encapsulated in the individual design subjects can be integrated in a number of different combinations—e.g., some versions of SEEs might include the evaluation feature but not the checking feature, and some might include logging while others might not. This ability to "mix and match" features is another benefit of subject-oriented design. It requires only the specification of composition relationships among whatever design subjects are to be included in any given member of the SEE family. For example, Figure 6 illustrates the composition relationships required to define a SEE that includes all of the possible features (display, check, evaluation, and logging). The relationships between the kernel, check, evaluate and display subjects indicate *match[name]* correspondence with *merge* integration, while the logger subject is composed based on the *TotalLogging* pattern. This set of composition relationships is complete and sufficient to either derive the appropriate composition rules in the subject-oriented programming domain, or to produce the composed design as illustrated and described in the Appendix. A *match[name]* correspondence with *merge* integration means that in a composed design subject, if produced, classes and attributes having the same name in different design subjects would appear once, and operations having the same name would be aggregated. Since no inheritance anomalies result from the merge integration in this case, composed generalisation relationships can be constructed easily. The composition relationships involving Logger use the composition pattern, *TotalLogging*, introduced in Section 4.1.1, which specifies concisely that all operations are to be logged.

Producing a SEE that excludes any of the features is equally simple—we need only exclude the subject supporting that feature from all composition relationships. Because each requirement is encapsulated in a separate subject, removal of a feature does not impact the design of any other feature. An interesting example of how this is useful is in the design of the logger. In the original logger design, two methods, `turnLoggingOn()` and `turnLoggingOff()`, had to be included to support this feature. Two alternative approaches to achieving this feature optionality are possible with subject-oriented design. One is to include or exclude the Logger subject from compositions, depending on whether or not logging is required. Another alternative is have composition relationships with Logger in both cases, but with a *select* integration specification that provides an appropriate runtime selection criterion for the inclusion or exclusion of logging functionality dynamically. Both approaches have the benefit of not requiring any modifications to the design subjects.

4.1.3 Producing Code from the Design

We have already described how the subject-oriented design just described aligns with requirements. There are two approaches to aligning it with code. The first approach is to code each individual design subject as a code subject in the subject-oriented programming paradigm, and then compose the code subjects with a *composition rule* [OK+96] derived from the composition specifications in the design. The second approach is for the designers to construct an integrated design, and then write standard object-oriented code based on it. In either case, however, this two-faced alignment of subject-oriented design supports the realisation of one of software design's primary purposes—to bridge the gap between requirements and code. The first approach is preferred, however, because it results in code that is directly aligned with requirements, and that therefore has the same properties of traceability and, especially, evolvability, described earlier for subject-oriented designs. It also simplifies the process of round-tripping.

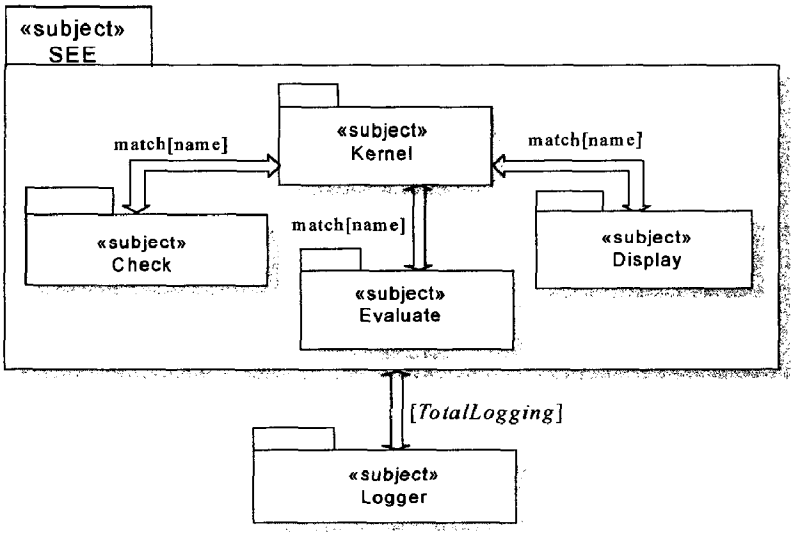

Figure 6: High-Level Subject Composition Relationships

4.2 Evolution Made Easy

The original SEE design suffered from the problem that what appeared to be simple, additive changes ended up being pervasive and invasive. Specifically, clients requested the inclusion of different forms of optional checking, thus rendering the check feature a "mix-and-match" capability. The solutions considered either resulted in combinatorial explosion of classes (using a non-invasive, subclassing approach), or required invasive changes to all of the AST classes (retrofitting design patterns). The subject-oriented design avoids all of these problems. Each different kind of checking is designed in a separate subject. Effecting the change request simply requires the definition of two new subjects: one to support the design of a def/use checker, and one to support verifying conformance to local naming conventions. Selective use of composition relationships permits designers to decide what kind(s) of check(s) are to be performed in any particular system produced from the design; the use of *select* integration defers this decision to users at run time.

This example illustrates the general point that subject-oriented design facilitates *additive*, rather than *invasive*, changes, significantly increasing the ease of system evolution.

5. RELATED WORK

There is a rapidly growing realisation that decomposition of object-oriented systems by class is necessary, but not sufficient for good software engineering. Classes often contain intertwined code pertaining to many different concerns. The work on subject-oriented programming [HO93,OK+96] provided a means of separating these concerns at the code level, and is the foundation of the work presented in this paper. The primary additional contributions of this paper are: more detailed analysis and illustration of requirements-design-code alignment problems; the application of the approach to design in general and to UML in particular; and introduction of the notions of composition patterns and guides and *select* integration.

Many other approaches have also provided improved separation of some kinds of concerns in object-oriented software, including aspect-oriented programming [KL+97], role modelling [RW+95, DW98, KØ96], contracts [HH+90, Hol92], propagation patterns [Lie96], composition filters [AB+92] and views [SS89, SU96]. The relationships of these and others to subject-oriented programming have been previously described in [HO93,OK+96,TO+99]. In the remainder of this section, we concentrate on two approaches that have particularly emphasised design rather than code: role modelling and contracts.

Role modelling. The goals of subject-oriented design and those of the role modelling work from the OORam software engineering method [RW+95] are similar. OORam shows how to apply role modelling by describing large systems through a number of distinct models. Derived models can be synthesised from base role models, as specified by *synthesis relations*. Synthesis relations can be specified both between models and between roles within the models, much like our composition relationships. The synthesis process is equivalent to the synthesis of subjects defined with a merge interaction specification. The subject-oriented design model distinguishes itself with its notion of *override* and *select* integration, and more particularly, with the open-ended semantics of subject integration. In addition, the potential provided by composition patterns and guides provide for more sophisticated, complex possibilities for combination patterns.

Role modelling with Catalysis [DW98] is based on UML, using *horizontal* and *vertical slices* to separate a package's contents according to concerns. Composition of artefacts is based on a definition of the UML *import* relationship, called *join*. The designer is instructed to form a new design containing the simple union of design elements, with re-naming in the event of unintended name clashes. This approach is similar to the meaning of a merge interaction specification with property matching by name. Catalysis encourages a design strategy in which an initial design is gradually modified to produce a completed one, which is a single, fully integrated design. We, instead, encourage a design strategy in which pieces (subjects) are identified and designed separately, and remain separate in the completed design, though related by composition relationships. This enhances the traceability of requirements that lead to various artefacts. For example, Catalysis describes the rules and decisions a designer should (might) follow to form the result of joining two packages, while we retain the original packages and, instead, define a way of specifying the rules and decisions as annotations on the composition relationship(s) relating them. Making a more complete specification of the sort we advocate necessitates the introduction of a wider and more sophisticated set of composition concepts, like matching and integration specifications. Reusable design components are supported in Catalysis with template frameworks, containing placeholders that may be imported, with appropriate substitutions, into model frameworks. This is similar to merging reusable design subjects that have no overlapping design elements, possibly using composition patterns.

Roles from Kristensen [KØ96] extend an object's intrinsic properties and may contain additional state and behaviour. Roles may be dynamically attached to objects, and more than one role can be bound to an object at a given time. Support for subject composition [Kris97], which is based on [HO93], combines class hierarchies by re-establishing the hierarchies to be composed as role hierarchies of a new intrinsic hierarchy. This maintains a clean separation of the different roles of objects, but does not address well the specification of concerns that cut across multiple hierarchies.

Other approaches to roles, of which MON [MD94] is an example, separate role specification from the intrinsic object specification, supporting the changing behaviour of objects in different stages of their lifecycles. In general, roles are defined at the object level of granularity, not for groups of collaborating objects.

Contracts: An approach to *interaction-oriented* design using contracts is described in [HH+90, Hol92], where contracts specify behavioural compositions and obligations on participants. They capture explicitly and abstractly the behavioural dependencies amongst co-operating objects. *Contract specification* identifies the participants in a behavioural composition and their contractual obligations. *Contract conformance* checks classes to ensure that they behave appropriately relative to all contracts in which they participate. *Contract instantiation* creates objects at run time that interact as described by the contract. Programming language constructs are required to use contracts fully. There is a difference of emphasis between contracts and subject-oriented design that might be thought of as "object composition" (concern with how functionality is provided by interacting objects), versus "class

composition" (concern with synthesising full class definitions from partial definitions reflecting different points of view). An individual contract describes an object composition; combination of contracts involves class composition. An individual design subject, on the other hand, is a collection of classes, not of objects. Other differences between the approaches include the fact that a design subject is a (partial) design in a standard language (e.g., UML) rather than a new kind of construct, the openness of composition relationships versus the specific, though more precisely defined, contract combination rules, and the fact that composition relationships can specify combination of methods and sharing of instance variables across design subjects.

6. CONCLUSIONS AND FUTURE WORK

Standard object-oriented designs do not align well with requirements. Requirements are typically decomposed by function, feature, property or other kind of user-level concern, whereas object-oriented designs are always decomposed by class. This misalignment results 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.

In this paper, we proposed and illustrated *subject-oriented design* as a means of achieving alignment between requirements and object-oriented designs, and hence alleviating these problems. This alignment is possible because requirements criteria can be used to decompose subject-oriented designs into *subjects*, which can then be synthesised as specified by composition relationships and their reconciliation and integration annotations. Subjects can be designed independently, even if they interact or cut across one another.

Support from the subject-oriented programming domain ensures that this level of alignment can be followed through to the code implementing the system. Propagation of changes from the requirements through to the design is therefore greatly simplified. This increases the chances of development teams keeping the design specifications up-to-date with the code, and alleviates the problems associated with misalignment at the code level also.

Much work remains to be done. The syntax and semantics of composition relationships and their reconciliation and integration specifications supported within the current model are being defined, together with ensuring that the model is an open, extensible framework that supports composition patterns and guides. As a start, this work will be done in the context of, and integrated with, the UML meta-model [UML99], though over time, it would be interesting to pursue its semantics independently of any particular design language's constructs. Support for all kinds of UML design models, in addition to the structural and interaction models discussed in this paper, must be included in the framework. Automation of the link from subject-oriented design to subject-oriented programming is also an important area we are interested in pursuing. This has a number of compnents, including generation of subject code from subject designs, and generation of composition rules from composition relationships. Environment support for subject-oriented design that includes such automation is needed, and it will provide the opportunity for validating the approach and gaining experience with its use.

7. ACKNOWLEDGMENTS

We are grateful to John Vlissides for his help with the use of design patterns. We thank Rob Walker and the anonymous reviewers for their comments. Thanks are also due to IBM Ireland Ltd. for partially funding Siobhán Clarke's work.

8. REFERENCES

[AB+92] M.Aksit, L.Bergmans, S.Vural, *"An object-oriented language-database integration model: The composition filters approach"* In Proc. European Conference on Object-Oriented Programming (ECOOP) 1992

[Bch94] G. Booch, *"Object-Oriented Analysis and Design with Applications (2nd ed.)"* Benjamin-Cummings, 1994

[BR98] G. Booch,, J. Rumbaugh, I. Jacobson, *"The Unified Modelling Language User Guide"* Addison-Wesley, 1998

[CAB93] D. Coleman, P. Arnold, S. Bodoff, *"Object-Oriented Development: The Fusion Method"* Prentice Hall 1993

[CD94] S. Cook, J. Daniels, *"Designing Object Systems: Object-Oriented Modelling with Syntropy"* Prentice-Hall 1994

[DW98] D. D'Souza, A.C. Wills, *"Objects, Components and Frameworks with UML. The Catalysis Approach"* Addison-Wesley, 1998

[GH+94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *"Design Patterns. Elements of Reusable Object-Oriented Software"*. Addison-Wesley 1994

[GF+98] M. Griss, J. Favaro, M. d'Allessandro, *"Integrating Feature Modeling with the RSEB"* In Proc. International Conference on Software Reuse (ICSR) 1998

[HH+90] R. Helm, I. Holland, D. Gangopadhyay. *"Contracts: Specifying Behavioral Compositions in Object-Oriented Systems"* In Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1990

[HO93] W. Harrison, H. Ossher, *"Subject-Oriented Programming (a critique of pure objects)"* In Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1993

[Hol92] I.M.Holland. *"Specifying reusable components using contracts"* In Proc. European Conference on Object-Oriented Programming (ECOOP) 1992

[Jac94] I. Jacobson. *"Object-Oriented Software Engineering: A Use Case Driven Approach"* Addison-Wesley 1994

[KØ96] B.B.Kristensen, K.Østerbye. *"Roles: Conceptual Abstraction Theory and Practical Language Issues"* Theory and Practice of Object Systems, Volume 2(3), 143-160 (1996)

[Kris97] B. Kristensen *"Subject Composition by Roles"* In Proc. Object-Oriented Information Systems (OOIS) 1997

[KS98] R. Keller, R. Schauer, *"Design Components: Towards Software Composition at the Design Level"* In Proc. International Conference on Software Engineering (ICSE) 1998

[KL+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, *"Aspect-Oriented Programming"* In Proc. European Conference on Object-Oriented Programming (ECOOP) 1997

[Lie96] K. J. Lieberherr. *"Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns."* PWS Publishing Company, 1996.

[MS91] S. Mellor, S. Shlaer, *"Object Lifecycles: Modelling the World in States"*. Prentice Hall, 1991

[OH+95] H. Ossher, W. Harrison, F. Budinsky, I. Simmonds, *"Subject-oriented programming: Supporting decentralized development of objects"* In Proc. 7th IBM Conference on object-oriented technologies, Santa Clara, CA. March (1995)

[OK+96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, V. Kruskal, *"Specifying Subject-Oriented Composition"* Theory and Practice of Object Systems, Volume 2(3), 179-202, 1996

[Par72] D.L.Parnas. *"On the criteria to be used in decomposing systems into modules"* Communications of the ACM, 15(12):1053-1058, December 1972

[RW+95] T. Reenskaug, P. Wold, O.A. Lehne, *"Working with Objects: The OORam Software Engineering Method"*. Prentice Hall, 1995

[RL+90] J. Rumbaugh, W. Lorenson, M. Blaha, *"Object-Oriented Modelling and Design"* Prentice Hall 1990

[SM89] S. Shlaer, S. Mellor, *"Object-Oriented Systems Analysis: Modelling the World in Data"* Prentice Hall 1989

[SS89] J.J.Shilling, P.F.Sweeney. *"Three steps to views: Extending the object-oriented paradigm"* In Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA) 1989

[SU96] R.B.Smith, D.Ungar. *"A Simple and Unifying Approach to Subjective Objects"* Theory and Practice of Object Systems, Volume 2(3), 161-178 (1996)

[TF+98] C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L. Wolf. *"Feature Engineering."* Proceedings of the 9th International Workshop on Software Specification and Design, April 1998.

[TO+99] P. Tarr, H. Ossher, W. Harrison, S. Sutton. *"N Degrees of Separation: Multi-Dimensional Separation of Concerns"* In Proc. International Conference on Software Engineering (ICSE) 1999

[UML99] *"OMG Unified Modeling Language Specification (draft)"* Version 1.3 beta R7. June 1999

[Vli98] John Vlissides. *"Pattern Hatching: Design Patterns Applied"* The Software Patterns Series, Addison-Wesley 1998

APPENDIX: Composed Designs

As discussed in the body of the paper, tools can be applied to reduce composite designs to their fully expanded form. Such a transformation, while optional (as noted earlier), can be useful, particularly to a developer attempting to understand the full semantics of a composed design and all of the ramifications of a set of composition relationships. Figure 7 is an example of a fully expanded composed design—in this case, for the design shown in Figure 6. Not surprisingly, the fully expanded composed design shown in Figure 7 is very similar to the original SEE design depicted in Figure 1, except that classes are shown in *flattened* form, with all inherited members explicitly copied from superclasses to subclasses. The importance and use of flattening in composition semantics are discussed in [OK+96].

We can illustrate how it is derived with some examples. Consider the class `UnaryPlusOp`. It exists in the result because of its presence in the Kernel subject. In its flattened form, it contains (among others) the operations `check()`, `getOperand()` and `setOperand()`, which are derived by merging the flattened forms of `UnaryPlusOp` in the Kernel subject (which has `getOperand()` and `setOperand()` because it inherits them from `UnaryOperator`) and `UnaryOperator` in the Check subject (which contains `check()`). Classes `UnaryPlusOp` and `UnaryOperator` are merged here, despite their different names, because the Check subject does not contain a `UnaryPlusOp` class; in the case of no name match, the default is for a class to be composed with the same class as its superclass [OK+96]. In addition, composition with the Logger subject has caused the sequence of actions performed on operation calls to be updated. For example, when the `check()` operation is invoked on a `UnaryPlusOp`, it causes a sequence of methods to be executed: `beforeInvoke()`, `check()` and `afterInvoke()`, as shown. A similar situation is obtained for all other operations, but is not shown in the figure.

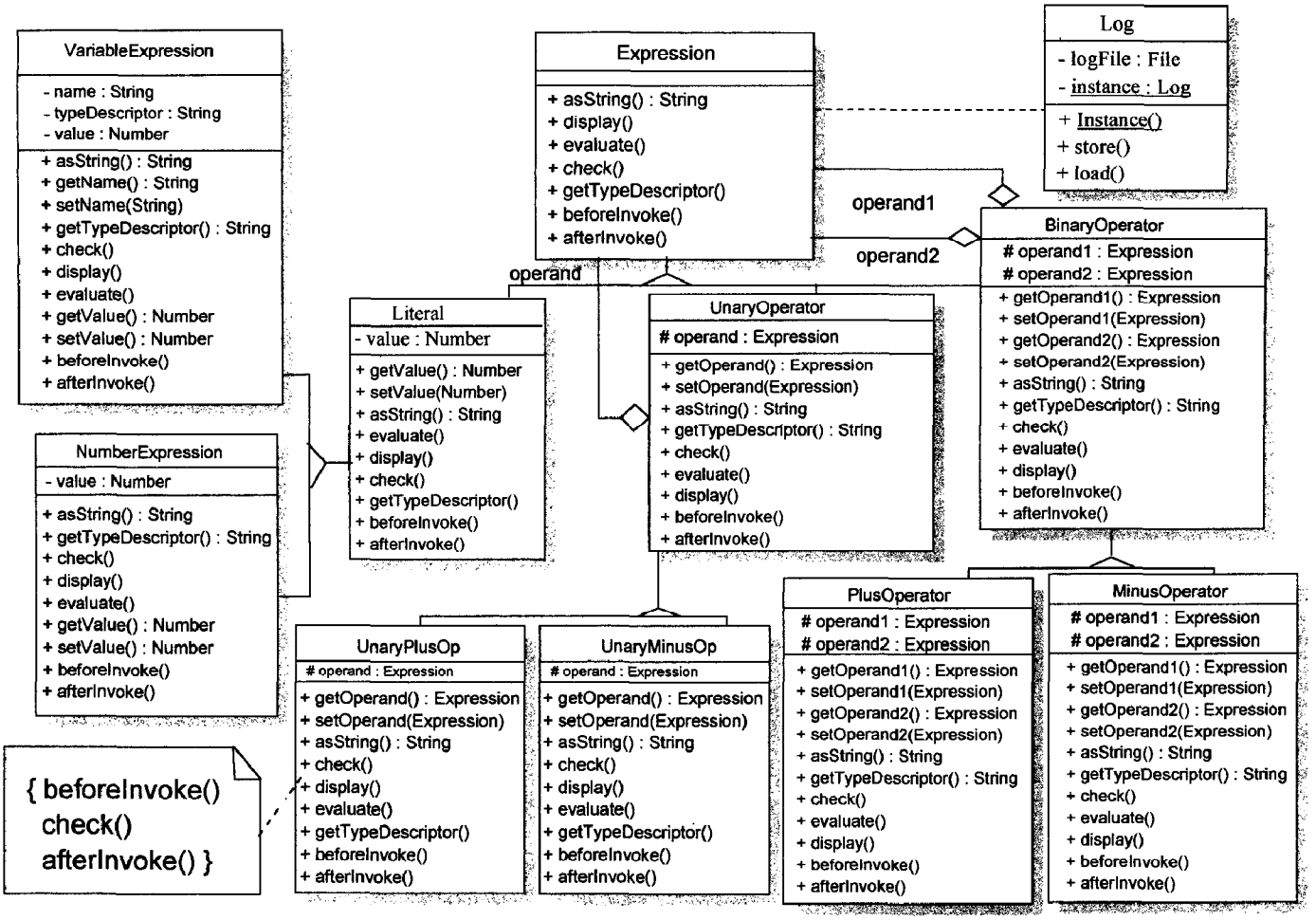

Figure 7: Composed SEE Design, Fully Expanded