Multi-View Software Evolution: A UML-based Framework for Evolving Object-Oriented Software

Robert France James M. Bieman
Computer Science Department
Colorado State University
Fort Collins
Colorado 80523 USA
970-491-6356, 970-491-7096
france@cs.colostate.edu, bieman@cs.colostate.edu

Abstract

It is well-known that uncontrolled change to software can lead to increasing evolution costs caused by deteriorating structure and compromised system qualities. For complex systems, the need to carefully manage system evolution is critical. In this paper we outline an approach to managing evolution of object-oriented (OO) software. The approach is based on a goal-directed, cyclic process, in which OO models are transformed and quantitatively evaluated in each cycle. Evaluation criteria guide developers in choosing between alternative transformations in each cycle. The process, transformations, and evaluation techniques can be used to develop systems from a set of baseline models.

Keywords Design transformations, design evaluation, object-oriented design, software evolution

1. Introduction

The large integrated software systems that are central to the missions of many organizations must change to satisfy continually evolving requirements. Adapting such software can be very difficult, because of the software's size and complexity and the variety of users or stakeholders with conflicting requirements. The cost of adapting systems includes both labor and time, and can be enormous. Furthermore, uncontrolled evolution of software can lead to deterioration of system architecture qualities.

We view software evolution as a process in which transformations are successively applied to software artifacts. Our notion of "evolution" includes (1) producing new software systems from scratch, (2) producing software artifacts from implemented software systems (e.g., producing models and documentation from legacy systems) and (3) producing new software system versions from previous versions. During the development of a new software system, requirements and design models evolve as developers gain a better understanding of desired functionality and qualities. These models are eventually transformed (evolved) to an implemented system. Evolution also occurs after release, as a result of corrective, adaptive, and perfective maintenance.

We propose an approach to software evolution based on transformations of object-oriented (OO) models and code. We are developing a framework, called *multi-view software evolution* (MVSE), that includes techniques to

1. characterize and represent alternative perspectives or views of an evolving software system,
2. transform system models and code to implement required modifications, and propagate the transformation effects across views and their implementations,
3. derive objective criteria from goals for evaluating the effects of a change, and
4. define a process that weaves these elements into a coherent, goal-driven approach to software evolution.

Our objective is to develop methods that scale up to large industry software systems (e.g., enterprise-wide information systems found in large telecommunication and retail organizations). The complex relationships among services and the diverse user base of these systems contribute to the difficulty of evolving them. Evolution can add new services, modify existing services, and update underlying support systems (e.g., databases, communication, and security systems). Such changes require: (1) assessing the impact on interacting services, support systems, and on user views, and (2) resolving possible quality conflicts arising from different views of the system, and minimizing potential degradation of system qualities by making appropriate tradeoffs. These requirements also apply to changes made during the development of new software systems.

MVSE has the following characteristics:

- *A multi-view, model-based approach*: Models of evolving systems make the evolution task more manageable. Changes are effected by evolving multiple views of a system represented as models, and propagating those changes to the implementation. Our notion of view is an adaptation of views and viewpoints as used in requirements analysis [25, 26]. Propagating changes, and assessing the impact of change across views requires well-defined relationships among and between the views and their implementations. Models expressed in languages such as the Unified Modeling Language (UML) [38], can identify dependencies and interactions among system components within and across views. Using well-defined relationships among models and implementations, changes can be propagated across views and their implementations.

- *A set of well-defined transformations*: The transformations are instantiations of evolution patterns, i.e., patterns reflecting types of software changes.

- *Objective evolution goals*: Selecting appropriate transformations depends upon objective determination of the success or failure of the transformations.

- *An iterative, goal-driven evolution process*: A process model that weaves the views, transformations, and goals together to manage evolution. Our model consists of cycles, where each cycle implements a well-defined change and consists of the following activities: (1) identification of change goals, (2) reduction of goals to objective evaluation criteria, (3) selection and application of appropriate transformations to satisfy the criteria, and (4) evaluation of transformation results.

In MVSE, evolution of complex systems is a process in which transformations are successively applied to multiple views of software (represented by models), until objective criteria are satisfied. This paper introduces the concepts underlying MVSE.

2. Evolving Software using MVSE

A stakeholder[1] view reflects the perspective a stakeholder has on a system's application and behavior. In MVSE, stakeholders initiate changes to systems and describe these changes in the context of stakeholder views.[2] For example, in a Telecommunication Information System (TIS), changes in switching and network technologies can trigger changes expressed in the context of a system/network view of the TIS, while changes derived from changing processes in a particular business sector (e.g., Customer Care or Billing) are often expressed in the context of service-oriented views of the TIS. Within a large organization, there are usually many stakeholders with different and sometimes conflicting views.

[1]A stakeholder is someone with a vested interest in how the system is used and evolved.

[2]We refer to a "stakeholder view" as a *view* when it does not cause confusion.

Precise relationships among different views of a system can help determine the impact of a change on different stakeholders. A good system architecture can provide a solid foundation for building relationships among multiple views: we treat a view as a sub-architecture that reflects a stakeholder's perspective on the system (see Fig. 1).

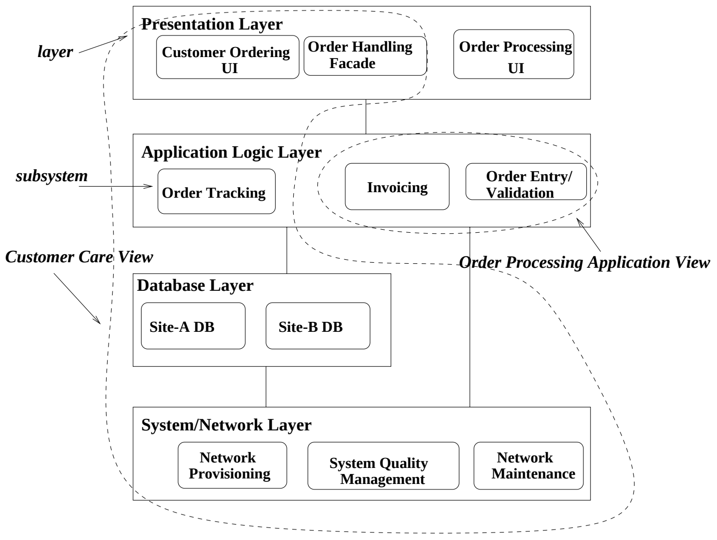

Figure 1. Views for a Distributed Order Handling System with a Multi-Tiered Architecture

MVSE treats a *system architecture* as a model of structure and behavior expressed in terms of *subsystems*, where a subsystem defines the behavior and structure of a clearly defined part of the system. A subsystem is developed and depicted in terms of UML models (e.g., Class Diagrams, Use Cases), where each model captures a particular aspect of the subsystem's structure or behavior (see the elaboration of *Subsystem 4* in Fig. 2). The integration of a subsystem's UML models (accomplished by relating concepts across the models) results in a comprehensive definition of the subsystem's structure and behavior.

A stakeholder view can be represented as an integrated set of UML models from the subsystems in the view. Fig. 2 shows an example of relationships between stakeholder views and subsystems.

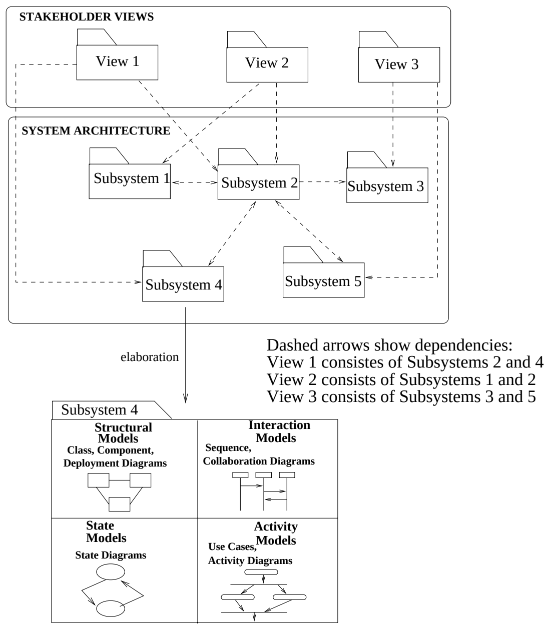
STAKEHOLDER VIEWS
View 1
View 2
View 3
SYSTEM ARCHITECTURE
Subsystem 1
Subsystem 2
Subsystem 3
Subsystem 4
Subsystem 5
elaboration
Dashed arrows show dependencies:
View 1 consistes of Subsystems 2 and 4
View 2 consists of Subsystems 1 and 2
View 3 consists of Subsystems 3 and 5
Subsystem 4
Structural Models
Class, Component, Deployment Diagrams
Interaction Models
Sequence, Collaboration Diagrams
State Models
State Diagrams
Activity Models
Use Cases, Activity Diagrams

Figure 2. Architecture and Views

MVSE treats software evolution as a process in which models in stakeholder views are iteratively transformed and evaluated. Well-defined relationships among the models in a view play key roles when propagating a change in one model to other models in the view. The propagation of change across different views is made possible by well defined relationships among the subsystems.

When evolution involves making a change to an implemented system, transformations eventually lead to the production of code that effect the change. When only code is available, transformations must generate models for stakeholder views from the code. During the development of a new product, transformations are carried out on the models to meet design quality goals, and to obtain implementations expressed in particular programming languages.

MVSE models are expressed using a precise form of the Unified Modeling Language (UML) [38], currently being developed by the *precise UML* (pUML) Group [27]. Class, Component and Deployment Diagrams provide structural views while Interaction and State Diagrams provide behavioral views of a system. The relationships among UML models in and across views are depicted using stereotyped dependency relationships (e.g., $\ll realize \gg$, $\ll refine \gg$, and $\ll trace \gg$). For example, $\ll traceOp \gg$ is a stereotyped dependency (more precisely, a stereotype of the $\ll trace \gg$ stereotype) that is used to depict the relationship between operations/methods in a Design Class Diagram and messages that request invocations of the operations/methods in Interaction Diagrams. Consistency checkers can ensure that operations/methods are located in classes as determined by the Interaction Diagrams.

Models can also include cross-referencing information to relate concepts across models. For example, Use Cases can include a cross-reference section with information about objects that are created, read, updated, and deleted (CRUD information) during the process described by the Use Case. CRUD information relates Use Case concepts to concepts in Class Diagrams, and can help determine the impact of a process change on underlying databases.

2.1. Horizontal and Vertical Transformations

We identify two broad classes of model transformations: *vertical* and *horizontal* transformations. A vertical transformation results in a target model that is at a different level of abstraction. Model refinement and abstraction are two forms of vertical transformations.

A horizontal transformation results in a target model that is at the same level of abstraction as the source model. Horizontal transformations can occur for a number of reasons:

- **To improve specific quality attributes of the model:** Transformations based on reusable experiences (e.g., design patterns [30]) can be applied to models to produce models at the same level of abstraction. Such transformations can result from a desire to (1) meet design goals, (2) address deficiencies uncovered by evaluations, or (3) explore alternative decision paths. We use the term *model refactoring* to refer to this type of horizontal transformation.

- **To support analysis of models:** During analysis, models can be transformed to make explicit properties that are being analyzed. Such transformations use the semantics of the models to determine what properties can be inferred from the models. We use the term *model inferencing* to refer to this type of transformation.

Fig. 3 illustrates horizontal and vertical transformations. The horizontal transformation is an example of model refactoring. It represents an application of a design pattern, the Bridge pattern, [30], to decouple the implementation of images (which can vary based on display resolution and availability of colors) from their abstraction. The vertical transformations shown in Fig. 3 are examples of class refinement and abstraction [21]. In a class refinement, a class object (the abstraction) is realized by a network of objects that interact to realize the public interface of the abstraction. In the figure, **DisplayImp** is a realization of **Display**, and **Display** is an abstraction of **DisplayImp**.

Effective management of evolution requires that the transformations be explicitly modeled; MVSE uses stereotypes of UML model relationships (e.g., realize, trace, re-

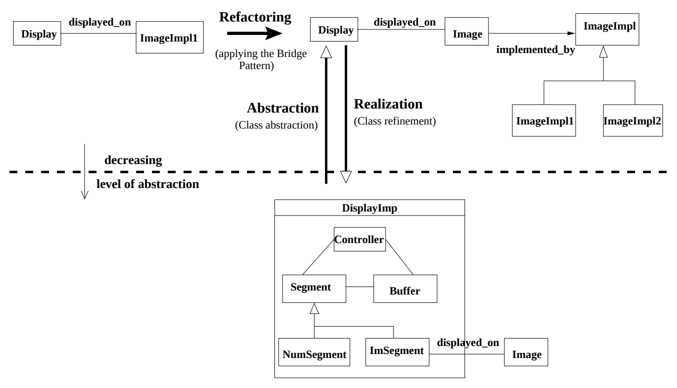

Figure 3. Model Transformations

fine). The stereotypes tailor the more generic UML relationships to the transformations types supported by MVSE. Information pertaining to transformation scope, measurable impact, and applicability are associated with the relationships. By explicitly modeling transformations one can track the evolution paths of OO software. This paper does not focus on the form of documented transformations. The focus in the remaining sections is on the process, transformation types and evaluation techniques.

3. A Generic Process For Managing Evolution

The generic development process weaves model transformation techniques together with model evaluation techniques. Explicit goals and evaluation criteria help to (1) determine when to perform transformation types and (2) select among alternative transformations of the same type.

We assume an *evolutionary life cycle*, where development is based on modifications to an existing system. Initial development is a special case where the base system is null. Development involves version-to-version transformations of any of the models that represent the various views of the system at various levels of abstraction. This assumption is consistent with a spiral life cycle [9], where development consists of cycles of activities which progress towards iterative development goals.

The generic process model is an adaptation of commonly recommended approaches to OO development [32]. Our adaptations include the following:

- We assume cycles (or iterations of a process spiral) that achieve specific interim goals. The process is goal directed.
- We assume that each cycle has an initial version with a set of models. The initial set of models can include from zero to many OO models of arbitrary types and levels of abstraction.
- The process supports both forward (refinement) and reverse (abstraction) engineering activities.
- Transformations achieve specific goals.
- The selection among alternative transformations is guided by evaluation criteria.

As in the general spiral model [9], our generic process model includes four general activities which represent the four quadrants of a cycle. However, our quadrants make use of the Goal-Question-Metric paradigm for quantitative assessment of software entities [2]; goal setting and determining mechanisms to evaluate changes play key roles in the process. The generic process model includes the following activities for each quadrant of a development cycle:

1. Identify cycle-goals for the current process cycle and derive a set of questions whose answers can determine the extent to which the goals are satisfied. The goal setting includes an assessment of the current state of the development process, and includes a review of the evaluation from the previous cycle.
2. Determine the transformations that are needed to satisfy the goals, thus identifying the target OO models to be created. Convert the questions from the prior step into more precise evaluation criteria which can answer the questions for the selected models created by the transformations.
3. Conduct the transformations, using the evaluation criteria to select between alternatives of the target OO model type.
4. Evaluate the generated OO models using the evaluation criteria developed in steps 1 and 2 to determine how well the goals have been met.

This generic process model is general enough to include a wide range of overall process types: The model fits development from scratch, where development starts with some loosely defined requirements, and cycles through a series of OO models as described by existing OO process models [21, 32]. In contrast with these models, the generic process model does not assume any particular starting point, and can be applied when there are incomplete and/or missing models. The model can be applied to iterative development which adds new features to an existing system. Such development may include a reverse engineering component, since OO models are often not available on existing systems. Then, an early cycle will abstract models from existing code so that adaptations can be made to appropriate OO models.

The goals for a particular process cycle can vary widely; the following are possible cycle goals:

- Create a (partial) design solution from a high-level analysis model. Cycle transformations will tend to be refinements.

- Gain understanding of an existing artifact or model. Cycle transformations will tend to be abstractions.

- Understand an orthogonal facet of an existing OO model. Inferences explore design consequences.

- Improve quality characteristics of a model. Refactoring transformations are likely candidates. For example refactoring can reduce model coupling, or introduce patterns to improve adaptability.

This generic development process depends on having the ability to rigorously refine the models vertically from higher levels of abstraction all the way down to program code. It also depends upon horizontal analyses at one level of abstraction to identify consistency between different views of the system. For example, an evaluation criteria might be that a Use Case can be "applied" to a Class Diagram to determine if the class model is sufficient for specified use scenarios. Such static testing of a Use Case against a Class Diagram can identify missing attributes and responsibilities. The generic process model formalizes the development process by providing well-defined connections between different models and at various levels of abstraction.

Application of the generic process depends on the development of appropriate goals and evaluation questions, the selection of transformations and target models based on the goals and questions, and the derivation of evaluation criteria that can be applied to OO models and answer evaluation questions.

4. Model Transformations

In this section we give an overview of the different types of transformations supported by MVSE.

4.1. Model Refactoring

Model refactoring occurs when a model is transformed to enhance quality attributes of the model. Using a design pattern (e.g., see [30, 16]) to improve a design results in model refactoring: the instantiated pattern, when applied to a model, results in a new model reflecting the solution defined by the instantiation.

To support effective management of transformations based on patterns, precise representations of patterns are needed. A problem with the current form of pattern descriptions is that the models of structure and behavior are given in terms of typical instantiations of the patterns. A more general, but precise, representation of pattern structure and behavior is needed if one is to objectively assess if a pattern has been applied appropriately.

In MVSE, a pattern specification determines the form of the UML models that are pattern instantiations. A pattern instantiation is a set of UML models (static structural and dynamic models) that reflect the pattern properties. A pattern specification in MVSE consists of two types of (meta-)models:

- **Static Role Models**: A *static role model* (SRM) characterizes the static structural models that can be part of pattern instantiations. It consists of classifier and relationship roles, where a classifier role defines properties that determine the UML classifer constructs that can "play the role" and a relationship role defines properties that determine the UML relationships (e.g., association, generalization, dependency) that can "play the role". An element that *plays (realizes) a role* has the properties specified by the role. Such an element is also called a *realization* of the role. Applying a pattern essentially involves associating modeling elements (e.g., a class construct) to roles specified in its role models. Given an SRM, a static structural diagram, and a "plays role of" mapping between the diagram constructs and the SRM role, the structural diagram is said to *realize* the SRM if the mapped constructs play the roles they are mapped to.

- **Dynamic Role Models**: A *dynamic role model* (DRM) characterizes the dynamic aspects of models that can be part of the pattern instantiations. For example, *behavioral roles* characterize the properties (in terms of parameterized pre- and post-conditions) that class operations must satisfy in realizations.

There are two types of properties that can be specified in a (Static or Behavioral) Role Model:

- *Metamodel-level constraints* restrict the form of constructs that can play the role. These properties are expressed as constraints over the UML metamodel elements [38] and thus limit the syntactic form of constructs that realize the role.

- *Model-level constraint templates* are parameterized constraints whose instantiations express semantic properties that must be expressed in models that realize the pattern. An instantiated constraint template is referred to as a model-level constraint. A constraint that requires an operation to have a particular effect on a system's state is an example of a model-level constraint.

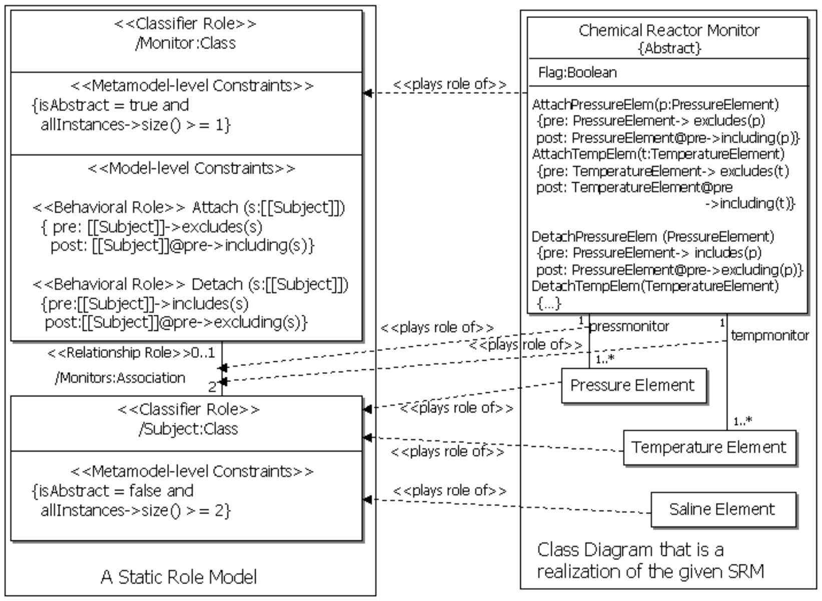
<<Classifier Role>>
/Monitor:Class
<<Metamodel-level Constraints>>
{isAbstract = true and
allInstances->size() >= 1}
<<Model-level Constraints>>
<<Behavioral Role>> Attach (s:[[Subject]])
{ pre: [[Subject]]->excludes(s)
post: [[Subject]]@pre->including(s)}
<<Behavioral Role>> Detach (s:[[Subject]])
{pre:[[Subject]]->includes(s)
post:[[Subject]]@pre->excluding(s)}
<<Relationship Role>>0..1
/Monitors:Association
2
<<Classifier Role>>
/Subject:Class
<<Metamodel-level Constraints>>
{isAbstract = false and
allInstances->size() >= 2}
A Static Role Model
<<plays role of>>
<<plays role of>>
<<plays role of>>
<<plays role of>>
<<plays role of>>
<<plays role of>>
Chemical Reactor Monitor
{Abstract}
Flag:Boolean
AttachPressureElem(p:PressureElement)
{pre: PressureElement-> excludes(p)
post: PressureElement@pre->including(p)}
AttachTempElem(t:TemperatureElement)
{pre: TemperatureElement-> excludes(t)
post: TemperatureElement@pre
->including(t)}
DetachPressureElem (PressureElement)
{pre: PressureElement-> includes(p)
post: PressureElement@pre->excluding(p)}
DetachTempElem(TemperatureElement)
{...}
1
pressmonitor
1
tempmonitor
1..*
Pressure Element
1..*
Temperature Element
Saline Element
Class Diagram that is a
realization of the given SRM

Figure 4. SRM and Static Structural Diagram

An example of a SRM and a static structural diagram that satisfies the SRM are shown in Fig. 4. The SRM shown on the left of the figure consists of two classifier roles ($Monitor$, $Subject$) and one relationship role ($Monitors$). The meta-model constraint for the $Monitor$ role states that only class constructs that are abstract can play this role. Furthermore, there can be only one such class in the diagram. The model-level constraints are behavioral roles that specify the required effects of behaviors on objects of the classes that can play the $Monitor$ role.

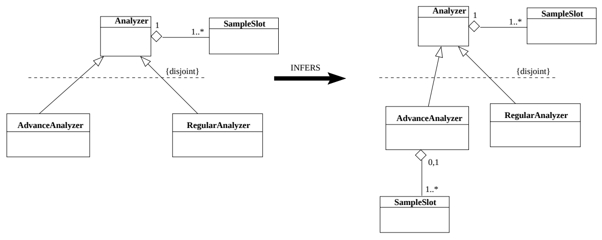

Figure 5. UML Inference Diagram

Model Inferencing A precise modeling language enables one to rigorously reason about properties captured by models. One can understand a precise model in terms of the properties that can be inferred from it. One approach to inferring properties from precise models is to transform the models in such a way that implicitly defined properties become explicit. These types of transformations are referred to as *inferences*.

To illustrate inferencing, consider the UML Class Diagram inference shown in Fig. 5.

In [29] we developed a technique for transforming CDs to Z specifications based on a precise semantics. By transforming the UML models in Fig. 5 to Z specifications we showed that the Z specification for the top model implies the Z specification of the lower UML model. We generalized the above transformation to a rule for demoting associations in single-level specialization hierarchies. Our early work defines UML Class Diagram inference rules [22, 28].

4.2. Model Refinement and Abstraction

Refinement/abstraction of a system model is accomplished by refining/abstracting one or more modeling views of the system model. In the framework, the notion of refinement is broader than more formal notions of refinement: new properties may be included in a refined element. For example, a refined Class Diagram may include structures (classes and relationships) not included in the more abstract Class Diagram. A more appropriate term for our notion of refinement is *detailing*. The notion of refinement supported in MVSE builds upon the refinement notions defined in Catalysis [21].

Vertical transformations on a model can trigger corresponding transformations on other models in order to maintain consistency among the models. For example, a refinement of an Interaction Diagram may introduce new classes of objects, in which case a corresponding refinement of the

Design Class Diagram is triggered, resulting in the addition of the new classes and appropriate associations. Well-defined relationships among models (represented as stereotyped UML dependencies) enables one to determine what models need to be transformed as a result of a model transformation. The following subsections provide examples of the types of model refinement supported by MVSE.

Class Diagram Refinement Class refinement results in an elaboration of class properties and may involve (1) refining class attributes and operations of the class (abstract data type refinement), (2) adding new properties (attributes, operations, relationships), or (3) decomposing a class into a substructure. For example, in moving from requirements to design, one may decide to centralize control of the system services by using a facade class to manage all interactions between a system and its users. This results in the addition of a class to the requirements Class Diagram.

Use Case and Activity Diagram Refinement Use Case refinement creates more detailed Use Cases, Activity Diagrams, or interaction diagrams. Activity Diagrams can be used to detail flows described in Use Cases. Use Cases are treated as informal entities in the framework, thus their refinement are guided by heuristics rather than formal rules. In MVSE, the relationship between a Use Case and its refinement can be documented by (1) relating Use Case steps to sections of Interaction Diagrams or Activity Diagrams that realize the steps, and (2) mapping Use Case concepts to elements in Interaction Diagrams (e.g., objects, links, messages) and detailed Use Cases.

Interaction Diagram Refinement Refinement can decompose a single interaction into a structure of interactions representing a complex protocol between objects. Conversely, abstraction can derive a single interaction that represents a complex set of related interactions.

5. Evaluation of OO Models

We evaluate models to assess designs and guide the development process. Ultimately, software artifacts should exhibit externally visible qualities such as maintainability, reusability, testability, reliability. These qualities depend upon the *architectural integrity* of the software — the desired structural quality of the software that should be embodied by the OO models that represent the software. MVSE evaluation techniques support detection of design situations that compromise architectural integrity.

Prior work on quantifying OO design structure focuses on lower level static design entities, most commonly classes or class models. Existing measures can quantify isolated intra-class attributes such as unit size, control flow, data flow, and cohesion, and inter-unit attributes such as coupling. Of the existing measures, those that quantify class coupling are the best predictors of one key quality attribute: class fault proneness [10, 14].

We focus on the architectural context of the software units — the higher level structures that a unit plays roles in. The quality of a design will depend on whether a class's interactions with the rest of a system is consistent with the intent of the architectural structures and design patterns that exist at a higher level of abstraction.

We merge class-level properties with the architectural contexts in which the program units exist. Architectural contexts are defined primarily through design patterns, and other well-known design structures, such as inheritance relationships. We quantify the architectural context in a way to insure that measures are internally consistent, and actually quantify the attributes of interest.

Design patterns represent preferred structures, and quantifying the existence of such structures can help in improving design quality. We quantify structure in terms of the use of preferred design structures as represented by design patterns. Components in a pattern are coupled in specific ways. Antoniol et al. found that measures of such connections can identify patterns in object models [1]. We can also derive measures of the specialized coupling exhibited within pattern instances.

Measurements can be based on the relationships between a class or other unit and the design patterns that it plays roles in. We can classify and quantify the patterns that contain a class and the pattern roles that a class plays.

A class may play a role in more than one pattern. Figure 6 shows the partial design of a system to model an abstract syntax tree (AST) of a programming language. Two classes of AST nodes are shown: `StmtNode` which represents AST nodes for program statements, and `DeclNode` which represents AST nodes for program declarations. The design supports two kinds of AST analysis, type checking and code generation. It makes use of two patterns from Gamma et al. [30]: Visitor and Abstract Factory. The Abstract Factory pattern supports the creation of instances of new AST node subtypes, and the *Visitor* pattern makes adding new kinds of AST analysis easier.

The two patterns in the design overlap. Class `AST,` `ASTNode,` `StmtNode,` and `DeclNode` in Figure 6 play roles in both patterns. Thus, these "dual role" classes are in context with two architectural structures. Modifying these classes may affect the expected behavior of both patterns, and thus these classes may be more difficult to adapt. Overlapping patterns may be very common in real industrial designs, especially those that are designed with patterns in mind [35]. We can readily quantify the number of design contexts for each class.

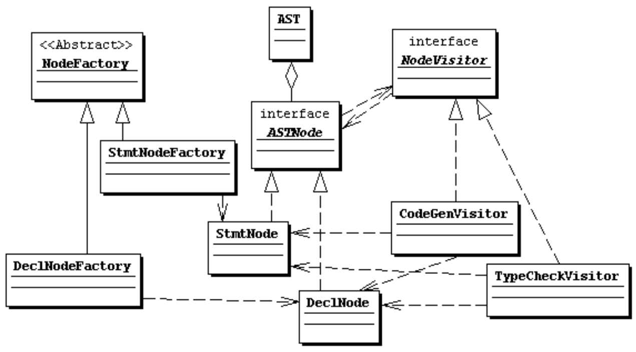

Figure 6. UML class model of a design with overlapping AbstractFactory and Visitor patterns.

We treat design patterns as architectural design units, just as classes are implementation units. From this perspective, a system is a collection of interacting patterns and independent classes — classes that are not part of any pattern. Pattern instances have attributes that are potentially measurable in a manner similar to class attributes.

Pattern coupling might be measured in terms of the connections between patterns. One pattern can be connected to other patterns through common classes — classes that play roles in more than one pattern, by referencing common objects, and by using methods in another pattern. Such pattern coupling could be viewed as a form of common coupling, an undesirable form of coupling according to Myers' ordinal coupling scale [37].

More desirable forms of coupling require connections via unit parameters. The most desirable form of coupling, data coupling, requires that a program unit pass simple data or a data structure as a parameter to another unit. Since design patterns are not yet supported as programming language units or programming environment units, connections through explicit pattern parameters cannot be made.

The internal structure of patterns and pattern instances can also be quantified. For example, we can measure the size of a pattern in terms of the number of its component classes. We can identify the sub-patterns contained within a pattern. Patterns may also have cohesion-like attributes, that are measurable in terms of the dependencies between pattern components, which may be classes or sub-patterns.

Other potential design measures can be based on some notion of distance from desired patterns. We can develop empirical relation systems [24] that model our intuition about relative distances of a structure of classes from a particular pattern. We can then define ordinal, and eventually ratio, scale measures from such empirical relation systems. This is the approach that we took to develop measures of functional cohesion and class cohesion [7, 5].

Our hypothesis is that components that play roles in a small number of well-defined architectural contexts will be easier to maintain, reuse, and test than components that play no role in these contexts or that play roles in many contexts. We are testing this hypothesis by examining commercial software data which includes software designs and process data, such as maintenance and reliability information [4].

6. Related Works

Lehman and his FEAST project show that transformations and system growth are inevitable, and it takes an input of effort to keep system complexity manageable [23]. We are developing a process and mechanism that guide us in applying transformations so that evolving system improve. Our work complements Lehman's by showing how to apply effort to manage system transformations.

6.1. Transformation Techniques

A number of techniques support the incorporation of patterns into development tools (e.g., see [15, 20, 36, 42]). These techniques use either imprecise pattern descriptions or base their descriptions on metamodels. Our work emphasizes precise representation of patterns in terms of role models. Like the Objecteering approach [20], we use transformations to incorporate patterns into designs. Unlike the Objecteering approach, our transformations are based on instantiations of role models, rather than mappings between metamodel elements.

MVSE uses notions of refinement and abstraction that builds upon the notions described in the Catalysis method [21]. Catalysis provides extensive guidelines for refining and abstracting OO models, but the descriptions are informal and there seems to be no support for rigorous application of refinement and abstraction techniques. One goal of our research is to develop formal notions of refinement and abstraction where possible.

6.2. Software Measurement and Evaluation

Briand et al, [13] identify five major quantifiable software design concepts: *size, length, complexity, cohesion*, and *coupling*. *Size* can be measured in terms of counts of entities, while *length* implies some kind of distance between entities. *Complexity* "depends on the relationships between entities" and is not a property of an isolated element. Here, complexity refers to "artifact complexity" rather than the psychological complexity of a human interacting with a design [19]. *Cohesion* refers to the relatedness of module components. Cohesive modules are difficult to split into separate components. *Coupling* refers to the connectedness of modules. A module with high coupling has many connections with other modules. Coupling differs from complexity in that coupling is generally measured with respect to an individual element or a pair of elements.

A commonly used heuristic is to design modules with high cohesion and low coupling [40]. Developers also try to minimize system complexity and component size.

No prior work addresses elements within specific architectural design contexts. Rather, published principles focus on elements within abstract, semantics-free, constructs.

6.3. Object-Oriented Design Measurement

Published measures of structure of object-oriented software primarily quantify properties of individual classes and their relationships to the rest of a system. Chidamber and Kemerer's suite of object-oriented measures include only single class measures [17]. Other researchers, including one of the authors, have developed cohesion measures for individual classes [5, 11, 31, 39].

Work on quantifying interclass properties (properties of collections of classes) includes Hitz and Montazeri's work to quantify coupling in object-oriented systems [31]. They classify individual dependencies between classes and between objects, and they propose an ordinal measure of the strength of coupling by a single dependency and the change dependencies between two components. Others study the coupling between two classes or objects: Coad and Yourdon [18] define inheritance coupling and design coupling between two classes; Wild [41] defines a hierarchy that classifies coupling between two classes. Briand et al. [12] describe the properties of 30 different object-oriented coupling measures. These coupling definitions and measures do not directly address coupling between classes that are components of specific architectural contexts or patterns.

One of the key reasons to use object-oriented methods is the often asserted claim that they lead to more reusable and adaptable systems. Measures of reuse and reusability are needed to evaluate these claims. Several relevant reuse abstractions, attributes, measures, and measurement tools are applicable to object-oriented systems [3, 6, 8]. The measures and tools are based on both the inheritance and calling structure of a system.

Inheritance and inheritance hierarchies are unique features of object-oriented systems and should have measurable attributes. Properties of inheritance hierarchies, or inheritance *clusters* [34], are subjects for analysis and measurement. The relationship between inheritance tree shapes and reuse through inheritance can be measured in terms of "code savings" [33]. These measured interclass properties are relevant to inheritance use, in general, but do not provide direct guidance for design pattern use.

7. Conclusions

We propose the MVSE framework to support managed evolution of OO software. MVSE supports: (1) objective evaluation of analysis and design models, and (2) rigorous analysis, refactoring, refinement, and abstraction of software models.

In MVSE, software evolution is a process in which models are iteratively evaluated and transformed. At the end of each development stage, the products are evaluated and goals for the next stage are refined. The refined goals are used in the next stage to drive decisions related to the types of transformations the models will undergo.

The development of MVSE is in the early stages. Our ongoing research has five major threads:

- Development of techniques for expressing patterns in terms of *role models*.
- Development of *model refactoring* techniques that (1) use patterns to transform models, (2) support the incorporation of components into designs, and (3) support the instantiation of designs from design frameworks.
- Precise characterizations of refinement and abstraction relationships.
- Development of objective criteria for evaluating OO models and of industrial-strength evaluation techniques that can be used to determine whether models meet the criteria.

8. Acknowledgements

This work is partially supported by U.S. National Science Foundation grants CCR-0098202 and CCR-9803491, and by a grant from the Colorado Advanced Software Institute (CASI) in collaboration with Storage Technology Corporation. CASI is sponsored in part by the Colorado Commission on Higher Education (CCHE), an agency of the State of Colorado.

References

[1] G. Antoniol, R. Fiutem, and L. Cristoforetti. Using metrics to identify design patterns in object-oriented software. *Proc. IEEE-CS Software Metrics Symp. (Metrics'98)*, 1998.

[2] V. Basili and D. Weiss. A methodology for collecting valid software engineering data. *IEEE Trans. Software Engineering*, SE-10(6):728–738, 1984.

[3] J. Bieman. Deriving measures of software reuse in object-oriented systems. In T. Denvir, R. Herman, and R. Whitty, editors, *Formal Aspects of Measurement*. pp. 63–83. Springer-Verlag, 1992.

[4] J. Bieman, D. Jain, and H. Yang. OO design patterns, design structure, and program change: An industrial case study. *Proc. Int. Conf. on Software Maintenance (ICSM 2001)*, To Appear Nov. 2001.

[5] J. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. *Proc. ACM Symp. Software Reusability (SSR'95)*, pp. 259–262, April 1995.

[6] J. Bieman and S. Karunanithi. Measurement of language supported reuse in object oriented and object based software. *J. Systems and Software.*, 28(9):271–293, Sept. 1995.

[7] J. Bieman and L. Ott. Measuring functional cohesion. *IEEE Trans. Software Engineering*, 20(8):644–657, Aug. 1994.

[8] J. Bieman and J. Zhao. Reuse through inheritance: A quantitative study of c++ software. *Proc. ACM Symp. Software Reusability (SSR'95)*, pp. 47–52, April 1995.

[9] B. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[10] L. Briand, J. Daly, V. Porter, and J. Wüst. A comprehensive empirical validation of design measures for object-oriented systems. *Proc. Int. Software Metrics Symp. (Metrics'98)*, pp. 246–257, 1998.

[11] L. Briand, J. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[12] L. Briand, J. Daly, and J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Engineering*, 25(1):91–121, 1999.

[13] L. Briand, S. Morasca, and V. Basili. Property-based software engineering measurement. *IEEE Trans. Software Engineering*, 22(1):68–85, Jan 1996.

[14] L. Briand, J. Wüst, S. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: an industrial case study. *Proc. Int. Conf. Software Engineering (ICSE'99)*, pp. 345–354, 1999.

[15] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems J.*, 35(2), 1996.

[16] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. Wiley, 1996.

[17] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, June 1994.

[18] P. Coad and E. Yourdon. *Object-Oriented Design.* Yourdon Press/Prentice-Hall, Englewood Cliffs, NJ, 1991.

[19] B. Curtis, S. Sheppard, P. Milliman, M. Borst, and T. Love. Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics. *IEEE Trans. Software Engineering*, SE-5(2):295–303, 1979.

[20] P. Desfray. Automation of design patterns: Concepts, Tools, and Practices. In *Proc. UML'98, Springer-Verlag LNCS 1618*, pp. 120–131, 1998.

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

[22] A. Evans, R. B. France, and E. S. Grant. Towards Formal Reasoning with UML Models. *Proc. Eighth OOPSLA Workshop on Behavioral Semantics*, pp. 67–73. Nov. 1999.

[23] FEAST/2: Feadback, Evolution and Software Technology. Web Site: http://www.doc.ic.ac.uk/˜mml/feast/

[24] N. Fenton. Software measurement: a necessary scientific basis. *IEEE Trans. Software Engineering*, 20(3):199–206, 1994.

[25] S. Fickas, A. V. Lamsweerde, and A. Dardenne. Goal-directed concept acquisition in requirements elicitation. *Proc. 6th Int. Workshop on Software Specification and Design*, pp. 14–21, 1991.

[26] A. Finkelstein, J. Kramer, and J. K. Goedicke. Viewpoint oriented software development. *Proc. of the 3rd International workshop on Software Engineering and its Applications*, pp. 37–51, 1990.

[27] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Computer Standards and Interfaces*, 19, 1998.

[28] R. B. France. A Problem-Oriented Analysis of Basic UML Static Requirements Modeling Concepts. In *Proc. of OOPSLA'99*, pp. 57–69, 1999.

[29] R. B. France, E. Grant, and J.-M. Bruel. FUZed: An UML-Based Rigorous Requirements Modeling Technique. Technical report, Colorado State University, Oct. 1999.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[31] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object oriented systems. *Proc. Int. Symp. Applied Corporate Computing*, 1995.

[32] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[33] B.-K. Kang and J. Bieman. Inheritance tree shapes and reuse. *Proc. Fourth Int. Software Metrics Symposium (Metrics'97)*, pp. 34–42, Nov. 1997.

[34] J. Mayrand, F. Guay, and E. Merlo. Inheritance graph assessment using metrics. *Proc. Third Int. Software Metrics Symposium (Metrics'96)*, 1996.

[35] W. McNatt and J. Bieman. Coupling of design patterns: Common practices and their benefits. *Proc. COMPSAC 2001.* To Appear October 2001.

[36] T. D. Meijler, S. Demeyer, and R. Engel. Naking design patterns explicit in FACE, A framework adaptive composition environment. In *Proc. ESEC/FSE'97*, 1997.

[37] G. Myers. *Composite/Structural Design*. Van Nostrand Reinhold, New York, 1978.

[38] T. O. M. G. (OMG). Unified Modeling Language. Version 1.3, OMG, http://www.omg.org, June 1999.

[39] L. Ott, J. Bieman, B.-K. Kang, and B. Mehra. Developing measures of class cohesion for object-oriented software. *Proc. Annual Oregon Workshop on Software Metrics (AOWSM'95)*, June 1995.

[40] W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Systems J.*, 13(2):115–139, 1974.

[41] F. Wild. Managing class coupling: Applying the principles of structured design to object-oriented programming. *UNIX Review*, 9(10):44–47, Oct. 1991.

[42] S. M. Yacoub and H. H. Ammar. Toward pattern-oriented frameworks. *J. Object-Oriented Programming*, 12(8):25–35, Jan. 2000.