Layered Class Diagrams: Supporting the Design Process

Scott Hendrickson, Bryan Jett, André van der Hoek

Institute for Software Research
University of California, Irvine
Irvine, California 92697-3455, U.S.A.
+1 949 824 6326
{shendric, bjett, andre}@uci.edu

Abstract. Class diagrams model a system's classes, their inter-relationships, operations, and attributes and are used for a variety of purposes including exploratory design, communication, and evaluation. However, traditional diagrams, and the tools used to create them, focus on capturing a single configuration – *the product of the design process* – rather than supporting the explorative *design process itself* that is used to create and evolve a design over time. This process involves iteration over multiple alternatives and evaluation of those alternatives. We present a layered approach and environment that encourages this process by capturing a design and its alternatives using layers. Layers may be combined with other layers to compose and explore new design alternatives for evaluation. Our tool provides mechanisms for creating, composing, and visualizing layers as well as detecting dependencies and conflicts among layers and managing semantic relationships among layers.

1 Introduction

Class diagrams are primarily used for two purposes: as *detailed design documents* that describe an implementation and as *conceptual models* that aid in designing that system [6]. In the former case, class diagrams sufficiently capture a *single design* of a corresponding system. However, as a conceptual model, class diagrams alone are insufficient. Designing nontrivial systems generally involves a design process that explores and evaluates *multiple design alternatives*. To better function as conceptual models, class diagrams need to support this *process of design* by capturing and organizing these alternatives, supporting their evaluation, and incorporating new ones.

Most class diagramming tools focus only on capturing class diagrams as finished products consisting of a single document that contains the result of all design decisions. Consequently, creating a new alternative requires creating a new document, and combining complementary alternatives requires manually merging each documents' contributions into yet another document. Although some diff and merge tools are starting to emerge that help this process [2, 3, 10, 18], these deal with documents as a whole, and do not allow a designer to deal individually with each design decision, which is often captured implicitly along with many others in a single document. For example, a new design document may contain both minor corrections to an old design combined with

major modifications incorporating a new piece of functionality. To incorporate one of these conceptual changes without the other requires a designer to determine which parts of the document map to which concept. Managing the design process in this way, without explicit support for explicitly modeling alternatives separately, is cumbersome and error-prone.

In this paper, we present an approach and supporting environment that encourages a *creative design process* by promoting a model of interaction relying on *layers*. Individual layers capture individual modifications to a design. These may include minor corrections, improvements to existing or additions of entirely new classes and associations, or even entirely different design approaches. By selectively composing layers on top of one another, different class diagrams are created that represent the accumulated modifications of the selected layers. In our approach, layers are first class entities that are independently manipulatable. This encourages capturing separate concerns in separate layers. The result is that a design consists of many individual alternatives and concerns that are properly separated and easily manipulatable, promoting a creative, explorative design process.

Clearly, some form of validity must be maintained in this process to ensure that the product of the design process produces consistent class diagrams. We use *relationships* for this purpose. Relationships allow a designer to explicitly set the rules according to which layers can be composed. Some relationships can be automatically detected, such as a class association created in one layer that points to a class created in another layer: the layer with the class association could not reasonably be applied without the layer that also creates the class it points to. Other relationships are semantic in nature, and must be specified by the designer, such as when two changes are logical alternatives, only one of which can be incorporated at a time. Relationships are modeled at the same level as, but independently from, layers. This allows a designer to easily specify and manage different compositions from the same set of layers.

We have implemented our approach in a layered design tool, EASEL. Our tool is similar to Rationale Rose [12] or ArgoUML [13], but is a layer and relationship centric design environment with special features to support these concepts. We view EASEL as a proof-of-concept prototype that shows that an approach in which alternative designs can be dynamically composed through the use of layers is indeed possible and that the issues involved in using such an environment can be addressed.

2 Motivating Example

To understand the problems in modeling class diagrams to date, we introduce a motivating example that we also use as the running example throughout the remainder of the paper. The example concerns class diagrams, shown in Figure 1, that each represents an alternative design for a hypothetical graph data model. As in a traditional design environment, each alternative is modeled separately: Figure 1a presents one design consisting of two classes representing edges and vertices in a graph and Figures 1b and 1c present alternative designs that reflect different design decisions.

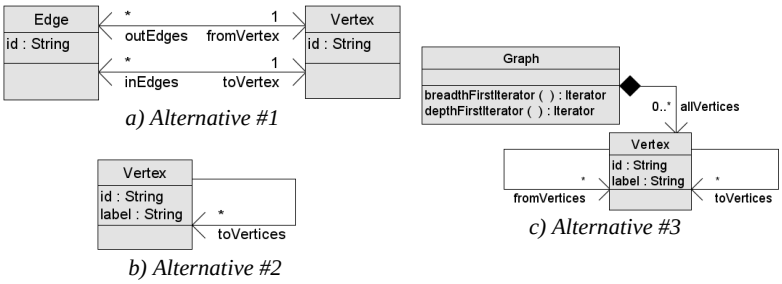


Fig. 1. Alternative designs for a hypothetical graph data model

Upon first glance, the only obvious difference between Figures 1a and 1b is that 1b is missing the *Edge* class. However, two distinct issues are actually addressed: (1) the *Edge* class was removed in favor of implicit out edges captured in the *toVertices* association, and (2) a *label* attribute was added to the *Vertex* class. Figure 1c differs in two distinct ways from Figure 1a as well: (1) a *Graph* class has been added that keeps track of all vertices, and (2) the *Vertex* class has an additional *fromVertices* association that keeps track of in edges.

Suppose that a designer decides that a fourth alternative would be best after examining the different alternatives shown in Figure 1. This fourth alternative would include the human readable label in Figure 1b, the *Graph* class in Figure 1c, the explicit edges in Figure 1a, and an additional *weight* attribute for each edge that is not present in these designs. The designer is now faced with the challenge of incorporating the desired parts from each original alternative into a single, coherent fourth design. This requires an understanding of the boundaries of each design concept in the first three alternatives, since creating and merging deltas of the alternatives is insufficient because the designer only wants a subset of the design concepts incorporated in each alternative.

Now, consider what happens if the designer later decides that a concept should be reintroduced that was originally rejected (or vice versa). Perhaps edges do not need to be represented explicitly and the implicit edges represented with the *toVertices* and *fromVertices* associations are adequate. Making such a design change would once again involve a manual revision of the design.

While the above is an overly simplistic example, these issues become more prevalent when one creates designs of a much larger scale and/or complexity. The example, then, shows the following needs we wish to address in this paper:

- *Separation of concerns*: we want to explicitly model different design concerns rather than implicitly mixing them together in a single document.
- *Composition*: we want to compose new, alternative designs from desired design concerns in order to explore and evaluate them.
- *Concern permanence*: we want design concerns to remain intact so that they may be (re)incorporated or removed at any time.
- *Variation*: we want to explicitly support alternative ways of realizing the same (or similar) functionality.

3 Approach

Our work was motivated by the observation that using class diagrams as conceptual models during the design process is hindered by tools that only record design documents *extensionally*. This means that resulting documents capture a single design without explicitly differentiating between the concepts that compose it, making the process of exploration cumbersome. Typically, a designer in exploratory mode necessarily must create and track multiple documents, concepts from which must be manually brought back and forth.

To facilitate an explorative design process, our solution incorporates two key insights. The first is that an *intensional* approach based on *layers* provides a natural mapping from conceptual intent and understanding to physical realization. The second key insight is that "straight" layers, as applied in Photoshop and similar tools for graphical editing, are not sufficient: explicit and detailed management of layer *relationships* must complement their use. Below, we detail these two insights and outline our solution in the context of the motivational example.

3.1 Layers

The discipline of *configuration management* (CM) has been primarily concerned with capturing the evolution of a software system at the *source code* level [5]. Of interest to this paper are the concepts of extensional and intensional versioning [4]. In *extensional versioning*, the entire configuration management system focuses on managing the versions of artifacts that result after changes have been made. That is, versions of artifacts are the primary "language" through which developers interact with the CM system. Extensional versioning ensures that each version is uniquely stored and accessible through revision numbers. Deltas may be used for storage optimization, but they are generally hidden from the user.

The key insight behind *intensional versioning* is to invert the relationship between versions and changes, making changes a first class entity, storing each change as a delta *independently from the other changes*. So, instead of requesting versions of artifacts, developers retrieve a set of changes and merge them together to create a particular "version." Similarly, when they have completed implementing a new "version" in their workspace, the delta between this new and the original version is stored as an individually-identifiable delta. Accessing an artifact, then, requires the developer to request a baseline (an initial, stable configuration) and a set of deltas.

There are two advantages to this approach:

1. Because a delta encapsulates a logically-related set of changes, it provides developers with a natural model of interaction. No longer must they mentally map desired conceptual features onto specific versions of artifacts. They can simply request features, bug fixes, and other kinds of changes by name.

2. Because each delta is built from the baseline, they are independent from each other. It is therefore possible to combine deltas in ways that they were not previously combined, creating new versions along the way.

At the same time, there is one major disadvantage:

1. Because each delta is independent, it is possible that certain combinations of deltas produce invalid or incomplete versions. Some of these conflicts can be automatically resolved, but others must be resolved manually.

The advantages of this approach are exactly what we would like to achieve with respect to class diagrams. We discuss how we address the disadvantage using the concept of relationships in Section 3.2.

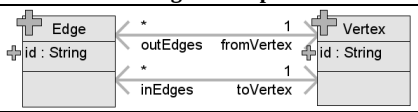
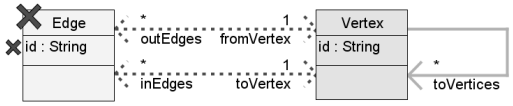
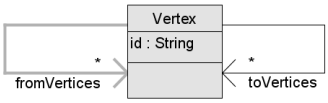
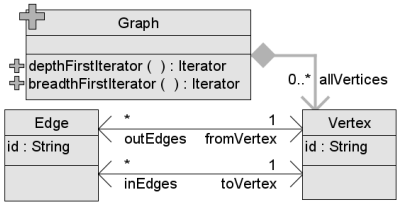
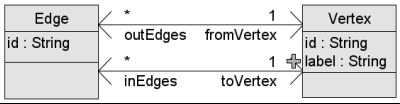
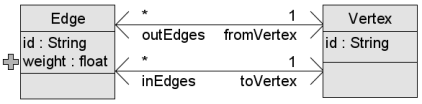
Our first step is to adopt the approach of using a baseline and deltas and apply it to class diagrams. Making this adoption requires adjusting the concepts of a baseline and deltas to operate at the level of design instead of lines of code. This is a straightforward adoption of the concepts of baselines and deltas, but applied to UML diagrams. We use layers as deltas, but with one exception. Because the process of design is highly iterative, we want a baseline to be editable in the same way that a delta is editable. We therefore start out with an empty, virtual baseline, and simply treat each layer as an increment from there. This is only a minor deviation, as the first layer could simply be treated as an imaginative baseline, emulating the original approach.

Consider the example presented in the previous section, but with the design and its alternatives captured using layers. Many possibilities exist for how the different designs might be partitioned over multiple layers. For instance, a very fine grained approach could be used where separate layers capture individual class operations and attributes. Alternatively, one could use a very coarse grained approach to capture the initial design and the two alternatives using just three layers. Both of these approaches technically work, but may not be as advantageous. The first is too fragmented, capturing point changes rather than design concepts; the second is too coarse grained, in essence reproducing the extensional approach that we are trying to overcome with our work.

A better way of capturing the design and its alternatives is presented in Table 1, where we capture each conceptual design feature in a separate layer. In each layer, added class elements are annotated with a "+" and added associations are shown using bold lines; removed class elements are annotated with an "×" and removed associations are shown using dashed lines. Unannotated elements are there for the sole benefit of the reader, placing the changes within context. For instance, in Table 1 the *Initial Design* layer adds an *Edge* and *Vertex* class and two associations. The *Use Only Vertex* layer removes the *Edge* class and its two associations, and adds another association. The other layers add, remove, or modify elements as depicted in Table 1.

Returning to the original three configurations in Figure 1, we can construct each of these designs by selectively merging layers from Table 1. The first alternative is represented by just the *Initial Design* layer. We compose the second alternative by merging the changes stored in the *Initial Design*, *Use Only Vertex*, and *Add Label* layers. We compose the third alternative by merging the changes stored in the *Initial Design*, *Use Only Vertex*, *Implicit In Edges*, *Track Vertices*, and *Add Label* layers. Finally, and this is where the power of our approach comes in, we can create the fourth alternative discussed in the text of Section 2 simply by merging the changes stored in the *Initial Design*, *Track Vertices*, *Add Label*, and *Add Weight* layers. No new changes needed to be made, we simply needed to compose a different set of existing layers.

Table 1. Layers capturing the design concepts of each design of Figure 1

Layer	Design Concepts
Initial Design	
Use Only Vertex	
Implicit In Edges	
Track Vertices	
Add Label	
Add Weight	

3.2 Relationships

Producing valid designs requires composing valid combinations of layers. From the layers presented in Table 1, we note the first basic relationships between layers: *structural dependencies and structural conflicts*. Structural dependencies arise when one layer's contents depend on elements introduced by another layer. For example, the *Add Label* layer adds an attribute to the *Vertex* class, which is created in the *Initial Design* layer. Consequently, in order to produce a valid design with the *Add Label* layer, the *Initial Design* layer must also be included. By the same reasoning the *Add Weight* layer also structurally depends on the *Initial Design* layer since it adds an attribute to the *Edge* class, which is created in the *Initial Design* layer.

Structural conflicts arise when one layer's contents depend on elements that are removed by another layer. For example, the *Edge* class that the *Add Weight* layer adds an attribute to, is *removed* by the *Use Only Vertex* layer. In order to produce a valid design with the *Add Weight* layer, the *Use Only Vertex* layer must *not* be included.

While structural dependencies and structural conflicts are of a syntactical nature, it is also necessary to support semantically meaningful relationships. For example, it does not make sense to have both the explicit edges from the *Initial Design* layer and the implicit edge from the *Implicit In Edges* layer. These layers can technically be merged, but would produce an undesired result. What is intended by the designer is for the *Implicit In Edges* layer to be included only when the *Use Only Vertex* layer has also been included, which removes the explicit edges it is replacing. The designer must be able to express such semantically meaningful relationships.

To support structural and semantic relationships, our work distinguishes three kinds of basic relationships through which layer relationships can be specified:

1. *and relationships*: this relationship states that if *all* of the layers *a*, *b*, and *c* are included, then layer *d* must also be included.
2. *or relationships*: this relationship states that if *any* of the layers *a*, *b*, or *c* are included, then layer *d* must also be included.
3. *variant relationships*: this relationship states that from a particular subset of layers *a*, *b*, and *c*, only a certain minimum and maximum number can be included at the same time.

The first two relationships are not necessarily singular: from any "source" layer(s) they can designate the inclusion of multiple layers (e.g., if *a*, *b*, and *c* are included, then *d*, *e*, and *f* must also be included) and exclusion of multiple layers (e.g., if *a*, *b*, and *c* are included, then *d*, *e*, and *f* must *not* be included). It is also possible to negate source layer(s) (e.g., if *a* is included and *b* is *not* included, then *c* must be included). Finally, the variant relationship may refer to an arbitrary number of layers, limiting the number included concurrently to one (making a group of layers mutually exclusive, creating a switch [11] or variant), or multiple (creating what COVAMOF [16] terms an alternative, which allows up to so many variants to be included at a time).

As with layers, different ways exist to choose and organize relationships. This is influenced by the choice of layers, but also by the personal preferences of the architect. Returning to our example, we could express the structural dependency of the *Add Label* and *Add Weight* layers on the *Initial Design* layer as "*Add Label* or *Add Weight* implies *Initial Design*." Similarly, we could express the structural conflict of the *Add Weight* layer with the *Use Only Vertex* layer as "*Add Weight* implies not *Use Only Vertex*."

Semantic relationships are expressed using the same rules. The semantic relationship that the *Implicit In Edges* layer is intended to be applied only when the *Use Only Vertex* layer is applied could be expressed as "*Implicit In Edges* implies *Use Only Vertex*," or alternatively as "not *Use Only Vertex* implies not *Implicit In Edges*."

Composition layers support the grouping of individual layers. They do not have any changes of their own. Instead, they use relationships to group layers in particular ways. For instance, to model the second alternative of Figure 1, we define a composition layer called *Alternative 2*, and a relationship stating: "*Alternative 2* implies *Initial Design*, *Use Only Vertex*, and *Add Label*."

3.3 Composition through Merging

The principal goal of our approach is to allow the designer to explore alternative designs with minimal interference from tools; we want a layer composition process that reflects this. While exploring different designs, we envision a designer turning on and off layers frequently and editing "earlier," previously created layers at any time. In fact, the strength of using layers in exploratory design is that the designer has the flexibility of not only creating a new layer to modify the outcome of other layers composed before it, but the designer can alternatively go back and modify a layer at any time to change the base of everything composed after it as well. This flexibility also enables a designer to select conflicting layers in order to produce a design which is "close to" what they want, then fix the design and build upon it using an additional layer that brings back the design to a consistent, non-conflicting state.

These goals are in contrast to configuration management systems that disallow revising previously committed deltas and whose process of merging deltas may require the user to manually correct conflicts. We want our merge process to be flexible enough to allow *strictly incompatible* layers to be merged in a predictable way and the process to be automated so that we do not unnecessarily interrupt the designer every time incompatible layers are selected. While the designer should be *aware* of incompatible layer selections, we do not want to prohibit the designer from selecting them or unnecessarily burden the designer in such cases.

We, thus, base our composition algorithm on traditional merge tools, but make a few adjustments. Specifically, we need to address ghost additions and removals. The first problem, "ghost additions", may occur when a class association from a first layer relies on the presence of a class from a second layer and, vice versa, when a class association from the second layer relies on the presence of a class from the first layer. Regardless of which layer is applied first in the merge process, a requisite class will not be there. The second problem is "ghost removals": when two mutual layers each remove a class established by the other layer, one of those removed classes is bound to erroneously reappear. Both problems arise, because we want to explicitly allow editing of "earlier," previously created layers at any time (as we discussed previously).

To address the problem of ghost additions and ghost removals, our merge process first applies all additions of all layers, in the order of classes first and then associations, and then performs all of the removals, in the order of associations first and then classes. The result is that all necessary classes are always present, avoiding ghost additions, and that classes that are intended to be removed are always removed, avoiding ghost removals. While the result is different from what one would expect from a traditional merge process, from the perspective of layer composition it makes sense to support a behavior that predictably shows added elements and hides removed elements. As an alternative solution, it would have been possible to disallow circular dependencies, but that would greatly restrict the flexibility of our layered approach during the exploration of design alternatives. We recognize that alternative composition behaviors may be desired. We address this in Section 5.

Even with the specialized merge process, small conflicts may still occur when layers make changes to the exact same element, i.e., two layers that each rename a class, but to a different name. In such cases, the order in which layers are merged, from first to

last, is used to resolve the conflict (which, because layer ordering is specifically supported in our EASEL tool, makes sense as a choice).

3.4 Summary

To summarize, our approach to modeling class diagrams with layers and relationships adheres to the following properties:

- *A class diagram is specified as a series of layers.* No longer is a class diagram captured as a monolithic design specification; it is instead a group of loosely coupled layers, each addressing a particular design alternative or concern.
- *Layers consist of sets of additions and removals of design elements*. The elements can be of any granularity, from classes, to associations, to properties (properties are not shown throughout the paper, but are supported by our infrastructure as discussed in Section 4).
- *An individual design is composed from layers*. Instead of working with different, complete designs, a design is created by selecting a set of desired layers and merging their additions and removals to construct the design.
- *Relationships capture structural and semantic dependencies, including potential conflicts among layers*. To avoid invalid designs, these relationships must be taken into account when composing a design from layers.
- *Complex relationships and layer hierarchies can be expressed using a combination of composition layers and relationships.* This allows a designer to deal with higher level concepts and express any Boolean expression.
- *Cyclic dependencies are resolved by using an adjusted merging process*. First, all additions of all of the layers are merged and only then are all of the removals applied.

We conclude by noting once more that relationships are expressed explicitly at the level of layers. This promotes variability to be the key mechanism and representation for design and allows the designer to reason about and manage differences among design concepts during the process of designing.

4 EASEL

To demonstrate our approach, we have implemented it in EASEL, a layered design environment for class diagrams. As illustrated in Figure 2, EASEL is partitioned into two separate areas: a drawing canvas for specifying class diagrams and a variability spreadsheet for managing layers and relationships.

The drawing canvas of EASEL operates similarly to Rationale Rose [12] or ArgoUML [13] in allowing a designer to add, remove, and change classes, their attributes and operations, associations, and properties. If none of the special features of EASEL are used, EASEL simply acts as a design tool for capturing individual class diagrams

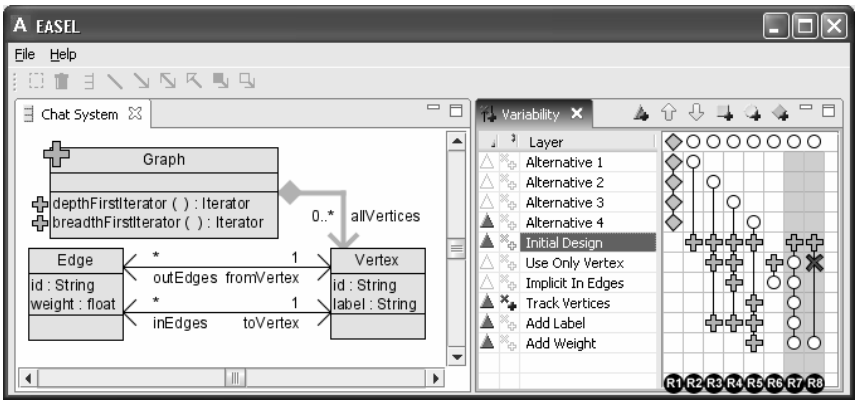


Fig. 2. Screen shot of EASEL. The relationships are labeled *R1* through *R8* for reference

much like these existing design environments. However, this is not EASEL's purpose. The drawing canvas has special behaviors that change it from a tool that merely captures a design product, to one that also supports the design process. First, the design on the drawing canvas is composed from the layers selected in the first column of the variability spreadsheet. For example, the design shown in Figure 2 is composed by merging the *Alternative 4*, *Initial Design*, *Track Vertices*, *Add Label*, and *Add Weight* layers. Layers are merged with the specialized merge process discussed in Section 3.3, avoiding large conflicts introduced by ghost additions or ghost removals and resolving any minor conflicts that are left using the order in which layers are listed from top to bottom. Changing this order is done by dragging layer names up or down.

The second special behavior is that each of the elements on the drawing canvas may be annotated with icons that explicitly illustrate each specific change recorded by the layers. This is what the second column of the variability spreadsheet is for: by selecting one or more layers in this column, the effects of the layers are shown. Currently, the *Track Vertices* layer is selected as such. This annotates the *Graph* class and its attributes with a "+" and bolds the association between the *Graph* and *Vertex* classes to indicate that these elements are added by that layer. If the layer were to remove elements, as, for instance, the *Use Only Vertex* layer does, the removed elements would be annotated with a "×" or dashed as discussed previously. Of course, when explicit display is not turned on, any elements that a layer removes are no longer visible; the drawing canvas only displays the results after merging all of the selected layers.

The third special behavior lies in how modifications made by a designer are stored. In EASEL, they are incorporated in the layer that is currently selected for editing. In Figure 2, this is the *Initial Design* layer, as highlighted. This means that each addition or removal made by a designer is added to the set of additions and removals of that layer. This is key to the power of EASEL in supporting the process of designing and modeling class diagrams. Had we enforced an incremental model in which layers are frozen once they have been created, much design flexibility would be lost making it impossible to revisit and update a feature without creating an additional layer. The

drawback is that inconsistencies may arise. However, as we will see below, EASEL has features to deal with this problem.

The final special behavior is that EASEL allows a designer to explore new layer combinations even if they produce invalid designs. This behavior allows the designer to freely explore new designs produced by new layer compositions. As we discuss below, however, EASEL does *inform* the designer of compositions that violate relationships. Thus, relationships act as design critics [14] rather than hard constraints. Without allowing invalid designs, exploring new design alternatives would be difficult.

The variability spreadsheet shown on the right hand side of Figure 2 provides a designer with a graphical representation through which they can edit the relationships that exist between layers. The rows of the variability spreadsheet represent layers and the columns relationships. Seven different symbols are used (please read the expressions carefully):

- A white *circle* represents a source of an *or* relationship (e.g., A in "A or not B implies C" or "A or not B excludes D").
- A white, *slashed circle* represents a negated source of an *or* relationship (e.g., B in "A or not B implies C" or "A or not B excludes D").
- A yellow *square* represents a source of an *and* relationship (e.g., A in "A and not B implies C" or "A and not B excludes D").
- A yellow *slashed square* represents a negated source of an *and* relationship (e.g., B in "A and not B implies C" or "A and not B excludes D").
- A cyan *plus* represents an implied destination (e.g., C in the examples above).
- A red *X* represents an excluded destination (e.g., D in the examples above).
- An orange *diamond* represents a variant in a *variant* relationship (e.g., A, B, or C in "variant(A, B, C)"). The minimum and maximum number of variants that may be selected concurrently is viewable and editable using context menus.

Returning to the example in Figure 2, the way to read some of the relationships, then, is as follows:

R1. The *Alternative 1*, *Alternative 2*, *Alternative 3*, and *Alternative 4* layers are variants of each other, that is, only one can be included at a time.

R3. The *Initial Design*, *Use Only Vertex*, and *Add Label* layers are implied by the *Alternative 2* layer, and they should always be included in the overall selection of layers whenever the *Alternative 2* composition layer is included (and, in fact, EASEL performs this inclusion for the designer upon selection of the *Alternative 2* layer).

R7. The *Initial Design* layer is implied by the *Use Only Vertex*, *Implicit In Edges*, *Track Vertices*, *Add Label*, and *Add Weight* layers.

R8. The *Add Weight* layer implies the *Initial Design* layer, but should not be included with the *Use Only Vertex* layer.

EASEL automatically detects a number of relationships, adding them to the variability spreadsheet with a slightly darker background. In Figure 2, relationships *R7* and *R8* were automatically added by EASEL. In general, EASEL automatically detects layers

that structurally depend on or structurally conflict with other layers (i.e., a layer links to a class created in another layer, or a layer removes a class that another layer links to).

In addition to detecting direct dependencies and conflicts between two layers, EASEL searches for additional layers that may affect these dependencies or conflicts. For instance, if a layer creates an association to a class that is created in a second layer, the first layer depends on the second. However, if a third layer removes this association, the dependency between the two layers would be removed. EASEL examines the contents of all layers when creating relationships, and in such cases generates appropriate relationships.

Finally, the implementation mechanism for automatically detecting relationships is extensible. EASEL could, for example, be easily extended to create variant relationships among layers that add classes with the same name – thereby addressing one of the minor conflicts that the specialized merge algorithm cannot automatically handle.

The current selection of layers shown in Figure 2 produces a valid design. However, if the designer were to make a selection that was invalid, EASEL would inform the designer of the invalid selection by highlighting violated relationships in red. For example, if the designer were to additionally include the *Use Only Vertex* layer in the selection shown in Figure 2, EASEL would highlight the violated relationship, R8. In general, when a designer is content with a particular selection of layers, despite violated relationship(s), then they have a few options to resolve the semantic and/or structural conflicts: (1) the designer could create a new layer that adds and removes elements that resolve the conflicts, (2) the designer could go back and modify the problematic layers so that they are compatible when merged, or (3) the designer could change the explicitly defined semantic relationships so that they are no longer violated. The choice will be influenced by the existing layers, their impact on other compositions, and by personal preferences.

Of note is that automatically-detected relationships are continuously updated while the design at hand is being modified. These relationships serve as critics [14] and disappear when particular dependencies or conflicts no longer exist. Hence, automatically detected relationships assist a designer during the design process, as they inform them of the fact that their current design is exhibiting some relationships that may or may not have been intended.

5 Discussion

At this point, a full-fledged validation of EASEL and our approach versus other design editors and notations is unavailable. However, our implementation of EASEL demonstrates that a layered approach is feasible and can be used to represent monolithic designs as compositions of layers. It, in fact, was meant as such: a proof-of-concept prototype exploring the feasibility of the technology.

To date, we have found in our early explorations that the explicit support of design alternatives in our layered approach is convenient and non-intrusive. We found it less prohibitive to make changes in EASEL than with tools without layered support, particularly since we could easily engage and disengage related collections of changes while

we explored various design alternatives. One strength in this process is that layers are generic and can capture any type of change (i.e., improvements, features, or even entirely different directions of design choices). While on the one hand this could be said to mix metaphors and perhaps end up being confusing, on the other hand, once one understands how layers compose, it represents a much more agile attitude in which the designer can flexibly use layers to their best convenience.

Using our tool also has revealed some weaknesses. We found that it would be useful to allow a designer to, after exploring many different alternatives, somehow indicate the nature of the changes stored in a layer or to otherwise organize them by content or status (e.g., "critical baseline", "stable", "in flux", "some changes still needed"). We also found a need to allow a designer to split, merge, and rearrange layer content. Since our tool focuses on supporting the design process (and thereby discovering the "right" design incrementally, necessitating frequent restructuring and redistributing concepts over layers), this is functionality that is necessary and will be implemented soon. Additionally, we recognize that alternative composition behaviors may be desired by designers. For example, a designer may wish that layers be applied in the order specified and fail if there are inconsistencies, or a designer may want the option to resolve those inconsistencies manually.

Finally, we found the automatically created relationships helpful in indicating when we had overlooked the impact of one change on another layer's contents. However, we found that as the number of relationships could grow very large quickly, and the number of relationships relevant to our particular task was frequently small. We will explore automatic filters that display only relationships relevant to the current design and will investigate approaches to grouping and summarizing relationships to reduce the cognitive demands on the designer.

6 Related Work

Other research has worked towards supporting the design process as well. ArgoUML [15], for instance, eases the cognitive challenges of the design process through the use of design critics, task organization and prioritization, and supporting a designers' natural tendency to switch tasks during the design process. The overall focus of ArgoUML is on *guiding* a designer through the design process, which is different than, but complimentary to, the focus of this paper.

Differencing and merging algorithms have been applied to UML [18] and generically to diagrams [2, 10]. Our approach, as we discussed in Section 3.3, uses differencing and merging algorithms internally to capture and apply layers. However, our approach makes some specific adjustments to address ghost additions and removals in order to avoid continuously bothering the user with manual resolution requests during the merge process.

Aspect-oriented modeling [8], programming [9], and aspect-oriented design [17] are also related to our work. Aspects are also compositional and used to separate concerns. In fact, aspects have already been applied to UML diagrams. Symmetric approaches [1, 7], which do not differentiate between "aspects" and a "base", but treat these as the

same, are more closely related to our approach. However, the focus of our approach is different and the resulting needs of the technology differs from those capabilities offered by asymmetric and symmetric aspects: (1) we need both additive and subtractive capabilities, (2) we want the ability to freely make edits rather than through specific kinds of joinpoints, and (3) we need relationships to track how layers are compatible to one another. Nonetheless, with sufficient work, our approach could probably be made to support an aspect-oriented approach, and vice versa.

7 Conclusion and Future Work

The contribution of this paper is an innovative modeling approach that supports the naturally explorative design process. Rather than forcing the specification of a monolithic design that intermingles many implicit design concepts, our approach enables a mode of work in which design concepts are separated as individually manipulatable layers. Key to our work is the application of intensional techniques to model class diagrams. This is supported with a specialized merge algorithm and the ability to capture and manipulate both structural and semantic relationships.

In addition to addressing the issues raised in the discussion section, our future work involves several different strands. First, we wish to apply EASEL to a real system and obtain feedback from real designers. Our work to date demonstrates the feasibility of the layered approach, but now we wish to move beyond our own experiences to evaluate whether others experience the same benefits as we do. Second, we would like to explore how layers could further aid a designer by capturing additional types of information, such as example designs or templates upon which one can overlay a design under construction. Finally, we would like to explore how EASEL could be used to support collaborative design, using layers as a means of isolating and including contributions from different designers.

8 Acknowledgements

We thank the anonymous reviewers for their insightful comments and suggestions. Effort partially funded by the National Science Foundation under grant number DUE-0536203.

9 References

[1] alphaWorks. HyperJ. http://www.alphaworks.ibm.com/tech/hyperj, IBM.

[2] Briand, L.C., Labiche, Y., et al. Impact Analysis and Change Management of UML Models. In Proceedings of the 19th International Conference on Software Maintenance (ICSM'03), p. 256-265, Amsterdam, The Netherlands, September 22-26, 2003.

[3] Chen, P.S., Critchlow, M., et al. Differencing and Merging within an Evolving Product Line Architecture. In Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5). p. 269-281, Siena, Italy, November 4-6, 2003.
[4] Conradi, R. and Westfechtel, B. Version Models for Software Configuration Management. ACM Computing Surveys. 30(2), p. 232-282, 1998.
[5] Estublier, J., Leblang, D.B., et al. Impact of the Research Community on the Field of Software Configuration Management. Software Engineering Notes. 27(5), p. 31-39, 2002.
[6] Fowler, M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd ed. Addison Wesley: Reading, MA, 2003.
[7] Harrison, W.H., Ossher, H.L., et al. Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition. IBM Research Division, IBM Research Report RC22685 (W0212-147), December 30, 2002.
[8] Kienzle, J., Gray, J., et al. Report of the 7th International Workshop on Aspect-Oriented Modeling. In Satellite Events at the MoDELS 2005 Conference, Bruel, J.-M. ed. 3844, p. 91-99, Lecture Notes in Computer Science, Springer: Montego Bay, Jamaica, 2005.
[9] Lopes, C.V., Kiczales, G., et al. Aspect-Oriented Programming. In Proceedings of the European Conference on Object-Oriented Programming. Jyväskylä, Finland, June 9-13, 1997.
[10] Mehra, A., Grundy, J., et al. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In Proceedings of the International Conference on Automated Software Engineering (ASE 2005). p. 204-213, Long Beach, CA, USA, November 7-11, 2005.
[11] Ommering, R.v., Linden, F.v.d., et al. The Koala Component Model for Consumer Electronics Software. IEEE Computer. 33(3), p. 78-85, March, 2000.
[12] Rational Software Corporation. Rational Rose: Using Rose. IBM Corporation, Report 800-024462-000, p. 258, 2003.
[13] Robbins, J., Hilbert, D., et al. Extending Design Environments to Software Architecture Design. In Proceedings of the Conference on Knowledge-Based Software Engineering (KBSE'96). p. 63, 1996.
[14] Robbins, J. and Redmiles, D. Software Architecture Critics in the Argo Design Environment. Knowledge Based Systems. 11(1), p. 47-60, 1998.
[15] Robbins, J.E. and Redmiles, D.F. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. Information and Software Technology, Special Issue: The Best of COSET '99. 42(2), p. 79-89, 2000.
[16] Sinnema, M., Deelstra, S., et al. COVAMOF: A Framework for Modeling Variability in Software Product Families. In Proceedings of the Third International Software Product Lines Conference (SPLC 2004). p. 197-213, Springer Berlin / Heidelberg. Boston, MA, USA, August 30-September 2, 2004.
[17] Stein, D., Hanenberg, S., et al. A UML-based Aspect-Oriented Design Notation For AspectJ. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development. p. 106-112, Enschede, The Netherlands, April 22-26, 2002.
[18] Xing, Z. and Stroulia, E. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In Proceedings of the Automated Software Engineering (ASE 05). p. 54-65, Long Beach, CA, November 7-11, 2005.