Maintaining Life Perspectives During the Refinement of UML Class Structures

Alexander Egyed[1], Wuwei Shen[2], and Kun Wang[2]

[1] Teknowledge Corporation, 4640 Admiralty Way,
Suite 1010, Marina Del Rey, CA 90292, USA
aegyed@teknowledge.com

[2] Dept of Computer Science, Western Michigan University, USA
{wwshen, kwang}@cs.wmich.edu

Abstract. Models provide an alternative perspective for the understanding of a software system. However, models reflect the state of the system at the time of their creation (or last updating) but they do not reflect intermediate changes during the system's evolution. Depicting perspectives without showing changes is like watching a movie through a small set of still pictures (i.e., no motion). This paper demonstrates this problem on an existing technique for the automated simplification (abstraction) of class diagrams. We will show that it is computationally feasible to maintain a set of abstract perspectives of a class structure such that evolutionary changes to the class structure are instantly perceived through its perspectives. For developers, this provides the ability to understand changes to systems from the modeling perspectives they care about. It also gives the developers the confidence that their modeling perspectives remain up-to-date with the system even while the system evolves.

1 Introduction

Software is more than source code and software development is more than programming. Software development generates and maintains a wide range of artifacts, such as documentation, requirements, or design models; all of which are valuable to the understanding of a software system. These artifacts help developers in understanding the software system through different perspectives (i.e., representing different goals or problems). In doing so, these perspectives emphasize certain development concerns and ignore others that are momentarily not of interest. For example, the design is an abstraction of the implementation and it often omits language-specific programming details that are not necessary to the understanding of the system. Our notion of perspectives is similar to the notion of views, however, a view typically hides parts of a model whereas our perspectives interpret the hidden information.

Perspectives separate concerns and thus cope with the complexity of software development. Perspectives reduce the complexity of software development as they limit the amount of information the developers have to be aware of at any given time (i.e., instead of having to understand the entire system, developers only need to understand the perspectives). In this paper, we discuss perspectives of UML (v1.3) class structures [14]. With modern software systems becoming increasingly complicated, developers can easily lose their vision of the structure of the system while diving into the implementation details. It is thus common practice to retain abstractions of the class structure (sometimes referred to as higher-level designs or architectures [15]). These higher-level perspectives typically represent snapshots of the lower-level design, omitting lower-level details. It is not uncommon to retain different perspectives of the same lower-level design, to, say, represent different requirements, development concerns, or aspects (aspect oriented software development [11]).

While developers derive tremendous value from perspectives, they are not free. There is a cost in creating perspectives and a cost in maintaining them (i.e, new or changing goals or needs [4]). If a perspective cannot be updated promptly based on the changes made in a software system then the perspective no longer correctly reflects the system. This lack of correctness may then mislead developers.

Like many others [5, 12], we have investigated techniques for creating and maintaining perspectives. This paper builds on one such technique for the automated class abstraction [7]. This technique simplifies (=abstracts) UML class structures where developers can decide which classes to keep and which ones to temporarily "hide." This technique solves a range of concerns that will be discussed in Section 2. For example, the hidden classes have to be reinterpreted in terms of their effect on the remaining, non-hidden classes. Our technique has the benefit that developers may derive perspectives when they are needed. However, our technique does not maintain the correctness of the perspectives thereafter (i.e., during evolutionary changes). Of course, perspectives could be recreated instantly after design changes but this is computationally infeasible because class abstraction is not cheap computationally, there are potentially many perspectives, and iterative software development [4] encourages changes to be frequent. Relatively minor but frequent changes thus lead to costly re-transformations.

As an alternative, this paper discusses on how to efficiently update perspectives by only propagating changes (additions, removals) [3]. This paper thus contributes a technique for the instant and incremental abstraction of class structures to keep perspectives up to date continuously at a low cost. It works on the same rules as the original abstraction technique (batch abstraction) but it only updates changes. That is, any change to the system is evaluated in terms of its impact onto all perspectives. The change is then propagated to every perspective separately such that only those parts of the perspectives are updated that have changed.

This paper also contributes a new philosophy to working with perspectives. Since the perspectives are updated instantly, they provide developers with an instant understanding on the high-level effect(s) of their changes. Developers now instantly become aware about the impact of their low-level changes in context of the perspectives they care about. Previously, this could only be done after a costly batch re-abstraction and comparison. Even then, it was often not possible to tell exactly what had changed (e.g., if the name of two classes are swapped then an after-the-fact comparison might confuse this with the movement of its relationships).

This paper is organized as follows: Section 2 defines perspectives for class abstraction and provides background for generating them automatically. Section 3 discusses our approach to the incremental and instant class abstraction. Section 4 shows results of several case studies. Finally we draw our conclusions in Section 5.

2 Background

2.1 A Need for Perspectives

Fig. 1 shows a class diagram of a simplified hotel management system (HMS) taken from [7]. The role of the HMS is to provide support for hotel reservations, check-in/check-out procedures, and associated financial transactions. The class diagram depicts details on how a guest is a person (inheritance), how every person has an account or how payment and expense transactions are associated with accounts.

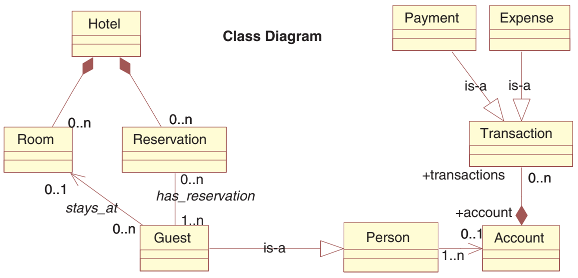

Fig. 1. Refinement of a Class Structure

While this class diagram is simple enough for human comprehension, we have worked with class diagrams that include thousands of classes and many more relationships [13]. It is impossible for humans to comprehend such class structures and developers resort to abstraction as a means of coping with this complexity. Abstraction depicts a class structure from a particular point of view, concern, requirements, or other form of interest. We refer to such an abstraction as a perspective of the class structure. Fig. 2 depicts four such perspectives of the class structure in Fig. 1.

Naturally, the perspectives in Fig. 2 are class structures themselves albeit simplified ones. A trivial form of a perspective is to represent a subset of the class structure only. For example, Fig. 2 (a) simply depicts the classes Guest, Reservation, and Hotel (and their relationships) from Fig. 1 by omitting all other classes and their relationships. These forms of trivial "perspectives" (i.e., sometimes referred to as views) are supported in many modeling tools, i.e., in form of diagrams. Yet, it must be understood that deriving perspectives is not just about eliminating details but also about re-interpreting the hidden details. For example, Fig. 2 (b) depicts the classes Guest, Payment, and Expense (as taken from Fig. 1) but it also depicts relationships among these three classes that are not to be found in Fig. 1. These relationships are the abstract interpretation of the hidden information. Fig. 2 (c) and (d) depict yet other perspectives that "slice" across the classes in Fig. 1. Clearly, there are a range of benefits associated with working with perspectives. Each perspective is easier to understand than the original class diagram.

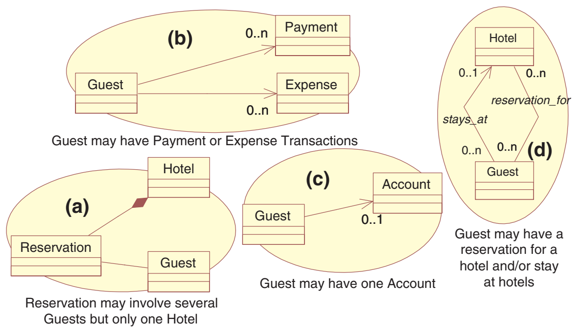
(b)
Guest
0..n
Payment
Expense
0..n
Guest may have Payment or Expense Transactions
(a)
Hotel
Reservation
Guest
Reservation may involve several Guests but only one Hotel
(c)
Guest
Account
0..1
Guest may have one Account
Hotel
0..1
0..n
reservation_for
stays_at
0..n
0..n
(d)
Guest
Guest may have a reservation for a hotel and/or stay at hotels

Fig. 2. Perspectives of the HMS system

Yet, without instant and incremental abstraction, it would be computationally infeasible to maintain these perspectives consistent with the system while the system evolves. That is, a change in Fig. 1 instantly renders all perspectives obsolete unless this change is propagated to all affected perspectives. Such propagation has the benefit that the perspectives continue to reflect the system accurately (i.e., important for decision making); and it has the benefit that the developers understand their system change(s) in terms of its impact onto the various perspectives. For example, if the cardinality from Person to Account changes from 0..1 to 0..n in Fig. 1 (i.e., a person may have many accounts and not just one) then which perspectives need updating? Does this change affect the Guest-Payment relationship in Fig. 2 (b)? Or does it change the Guest-Account relationship in Fig. 2 (c)?

2.2 Automated Abstraction

We previously developed a transitive reasoning technique in collaboration with Rational Software [10]. The technique takes arbitrary complex class structures and infers transitive relationships among its classes. A transitive relationship is the semantic equivalent of a collection of normal relationships. For example, if

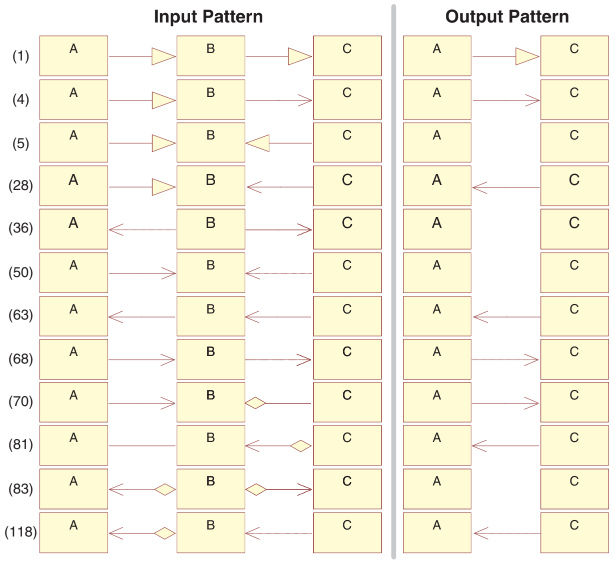

Fig. 3. Subset of Transitive Abstraction Rules for UML relationships [6]

A calls B and B calls C then, transitively, A calls C. Transitive relationships are thus indirect relationships between classes.

A transitive relationship is always the result of a collection of direct relationships. By composing the properties of a collection of direct relationships one can infer properties of the transitive relationship. Properties of relationships include the direction of the call, the type of relationship, or the cardinality of association ends. If, say, two relationships have the same type and the same calling direction then transitively the two relationships can be composed into a single one of the same type and direction (see Rule 70 in Fig. 3). Transitive relationships are thus a form of abstraction where the transitive relationship is semantically equivalent or weaker (less constrained) than the direct relationships it composes. Fig. 3 gives an excerpt of about 121 transitive relationships defined in [6]. For instance, rule 1 states that if A inherits from B and B inherits from C (input pattern) then, transitively, A inherits from C (output pattern). Or Rule 118 states that if C depends on B, A is a part of B (diamond head), and A is called by B (arrowhead) then, transitively, C depends on A.

The given transitive abstraction rules are simple in nature. Most rules describe a collection of two input relationships that are composable into a single output relationship (or not composable if the output pattern does not have a relationship). What makes this abstraction technique powerful is the large number of simple rules (121 rules for three types of class relationships and various properties). Given the simplicity of the rules, the abstraction algorithm is fast (see empirical studies in [6]); however, at the expense of precision. UML relationship semantics are not well-treated in the current UML specification which may lead to uncertainties during transitive reasoning (e.g., A calling B and B calling C may not imply A calling C always; see validation in [6]). While we cannot guarantee the correctness of all abstraction results, we found that we can guarantee completeness. That is, the lack of an abstraction result true means that there is no transitive relationship. Furthermore, validation showed that it was a two-orders of magnitudes (100 fold) saving in checking the correctness of abstraction results manually versus having to abstract by hand.

As input, the algorithm takes an arbitrary complex class structure and a list of "important classes". The list of important classes emphasizes the classes that should not be hidden. In Fig. 3, the classes A and C are important and the class B is not important as it gets replaced (together with its relationships) by a higher-level relationship. A human has to make the decision what classes are important as it depends on the circumstances and usage of the perspectives.

Important classes are not used for transitive reasoning during abstraction. They remain untouched during abstraction but their relationships to other, important classes are derived through transitive reasoning by hiding and re-interpreting unimportant classes (=helper classes). Fig. 4 shows the use of transitive reasoning in understanding the relationship between the important classes Guest and Payment (from Fig. 1). Although the two important classes are not directly related to one another, a transitive relationship can be derived by eliminating the helper classes Person, Account, and Transaction. Fig. 4 shows that the application of Rule 4 eliminates the class Person, the subsequent application of Rule 70 eliminates the class Account, and, finally, rule 28 eliminates Transaction. This results in an incremental abstraction where the previous result is then abstracted further if needed. The resulting abstraction is depicted in the bottom of Fig. 4. It depicts the two untouched, important classes and a single relationship between them that is semantically equivalent to the now-hidden helper classes.

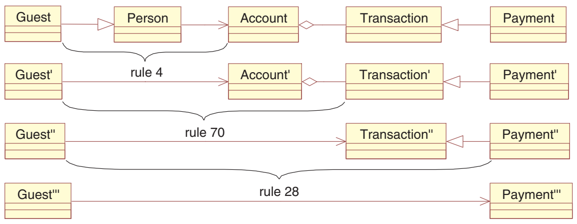

Fig. 4. Transitive Relationship between Classes

In summary, transitive reasoning merges low-level classes and relationships into higher-level relationships. This form of abstraction is necessary in cases where lower-level classes are the result of refining a relationship. For instance, the low-level class Account is important for implementing the HMS system but it is not needed on a higher-level abstraction to convey the point that a guest may have payment transactions. The class was thus hidden together with other classes and the hidden information was then re-interpreted through higher-level relationships. It is also possible to merge classes into higher-level classes (instead of relationships) and our approach is capable of doing so but its discussion is not of importance in this paper [8].

In the remainder of the paper we refer to the class structure in Fig. 1 as the design and to the abstractions in Fig. 2 as the perspectives. Abstraction assumes the existence of the design and a list of important classes in order to compute perspectives. The design and list of important classes must be provided by the developer.

3 Approach

Automated abstraction gives the developer the ability to create one perspective at a time. This perspective is then consistent with the design (assuming the rules for abstraction are accurate) but any change to that design may render any and all perspectives obsolete. Naturally, developers may re-compute perspectives to make them consistent again; however, many automated techniques, such as ours are computationally not cheap. It is thus infeasible to update perspectives continuously while the design changes.

This paper extends our previous work through incremental abstraction. Instead of updating perspectives in their entirety (batch abstraction), we only update changes. The basic goal of our approach is depicted in Fig. 5. Incremental abstraction understands both the design and its perspectives such that it can reason about a change in the design in terms of its impact onto the perspectives. It then updates the perspectives by deleting obsolete information or adding new ones. Compared to the abstraction of entire perspectives, we find this incremental approach to be much more efficient.

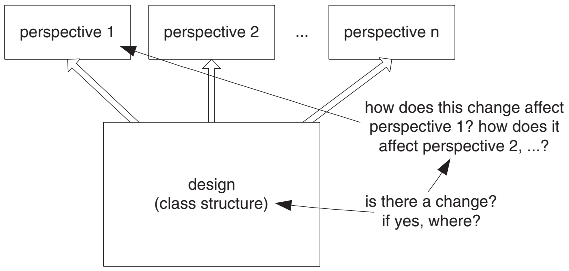
perspective 1
perspective 2
...
perspective n
how does this change affect perspective 1? how does it affect perspective 2, ...?
design (class structure)
is there a change? if yes, where?

Fig. 5. Instant and Incremental Abstraction to maintain the Consistency between a Design and its Perspectives

Incremental abstraction is a two-step process in that one has to understand 1) when and where changes happen in the design (class structure) and 2) how such changes affect the given perspective(s). While a change to the design is a constant, its impact is dependent on the particular perspectives at hand. This section explores these two issues and discusses our solution.

3.1 When and Where Changes Happen

To understand when and where changes happen in a design, we need to instrument its drawing tool (e.g., the design capture tool). Of interest is information about the creation, modification, and deletion of classes, their relationships, and associated properties (e.g., methods, attributes). This task is only moderately complex if the source code of the drawing tool is available. However, we previously demonstrated a capability for "spying" into commercial-off-the shelf tools to elicit these kinds of information [9]. In particular, we demonstrated on IBM Rational Rose [1] and Matlab/Stateflow [2] how to convert low-level keyboard and mouse events into the kinds of events discussed above (e.g., class creation, renaming, and relationship moving).

This technology has been published in [9] and is not described in more detail here aside to say that we have built a tool support, called the UMLInterface, that enables us to use the commercial tool IBM Rational Rose as a drawing tool for class structures and is able to observe developer changes. Fig. 6 depicts the architecture of our tool schematically where design changes from inside Rose are forwarded to our abstraction tool, which then responds by updating the perspectives in Rose. It is important to note that Rose maintains both the design and its perspective(s) and our tool simply propagates the changes. Therefore, all the visible activities happen inside Rose and the developer is never aware of our tool.

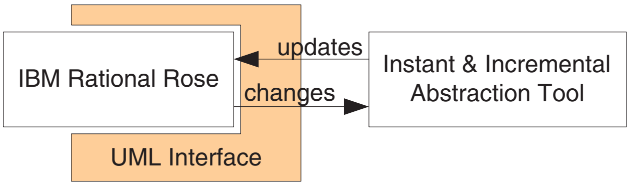

Fig. 6. IBM Rational Rose and the Instant & Incremental Abstraction of Changes

Since Rose is used as both a design drawing tool and a perspective visualization tool, we had to define logical structures for separating them. We also had to define a way for developers to designate "important classes." Recall from Section 2 that perspectives must define lists of important classes. We found a way of capturing this information inside Rose. However, these details are not discussed here as they do not contribute to the main topic of this paper.

3.2 How Changes Affect Perspectives

Incremental abstraction assumes the existence of a design and its perspectives. Changes to the design then cause updates to the perspectives. This approach assumes that perspectives are initially consistent with the design. The change to the design then causes an inconsistency and the simple propagation of the change is sufficient to re-establish consistency. However, there are situations

Table 1. Perspective Changes in Response to Design Changes

Impact of design onto perspectives			Perspective			
			Class		Relation	
			Add	Remove	Add	Remove
Design	Class	Add	no	no	no	no
		Remove	no	yes	no	yes
		Upgrade	yes	no	yes	yes
		Downgrade	no	yes	yes	yes
	Relation	Add	no	no	yes	no
		Remove	no	no	no	yes

where it is incorrect to assume initial consistency. For example, if a developer loads an existing class diagrams then we need to ensure initial consistency by abstracting all perspectives in their entirety. We refer to this process as the initial batch abstraction which is, in our case, the same as the normal class abstraction discussed in Section 2.

After the initial consistency between design and perspective is ensured, a change to the design requires no more than the abstraction of the change to again guarantee consistency. The kinds of changes in a design made by developers during software include: adding a class, removing a class, upgrading a class in a design from a helper class to an important class and downgrading a class (there are also other changes but are not discussed here). Table 1 depicts design changes in the rows. In response to a design change, the perspective may change by adding/removing classes and adding/removing relationships. Table 1 depicts these perspective changes in the columns.

We do not have a mechanism to prove the consistency between the batch transformation and incremental transformation. Thus, we tested batch abstraction and incremental abstraction concurrently such that we could compare differences. Fortunately, changes in the design have limited ways on how they affect a perspective. The fields in Table 1 indicate what kinds of perspective changes are caused by what kinds of design changes. For example, removing a class from the design may remove classes and/or relationships from the perspective (e.g., if the class was important then the perspective may loose a class; if the class was unimportant then the perspective may loose relationships). It is interesting to observe that class and relationship changes in the design have few effects onto the perspectives but class upgrades/downgrades are more complex. Table 2 summarizes and discusses these impacts in more detail.

Changing a Class in the Design

Adding a new class to a design does not add any new relation to that design. Therefore the perspective remains unchanged. However, deleting a class from a design may result in a change in the perspective. For example, if a developer decides that the class Person (Fig. 1) is no longer required in the design then this also changes some of the perspectives. For example, the perspective "Guest may have Payment or Expense Transactions" becomes out of date because Person is

Table 2. Design Changes and Impact onto Perspectives in More Detail

User Action	Changes based on perspective
Add a class	No change
Remove an important class	Delete class from the perspective. Also remove the relations between the class and all other important classes.
Remove a helper class	Delete relations from the perspective whose abstractions used the helper class.
Add a relationship between two classes	Find paths between important classes that pass through the relationship. Abstract these new paths into relations.
Remove a relation between two classes	Delete relations from the perspective that were abstracted from the removed relation.
Upgrade a class from a helper class to an important class	Add class to the perspective. Delete relations from the perspective that were abstracted from the upgraded class (the previous helper class). Also find paths between the new important class and other important classes. Abstract these new paths into relations.
Downgrade a class from an important class to a helper class	Remove class from perspective. Find paths between important classes that pass through the downgraded class. Abstract these new paths into relations. Also delete relations from the downgraded class to other important classes.

a helper class in that perspective and its removal affects the paths from Guest to Expense and Payment (recall Fig. 4). Consequently, there are no longer abstract relations among these classes and the perspective needs to be updated.

Since the removed class is a helper class, incremental abstraction only removes those relations in the perspective that were abstracted from it. Fig. 7 shows a design (left) and its perspective (right) with X and Y being important classes. If developers remove the helper class A in the design then our approach deletes the relation (2+A+3)' because this helper class was used to derive that relation.

Removing an important class in a design obviously results in its removal from the perspective. As a side-effect of the removal of an important class, all relations connecting to the important class must be removed also. For example, in Fig. 7 the removal of the design class X instead of A would delete the perspective class X' and all its relationships.

Changing a Relationship in the Design

There are two situations related to changing a relation in a design: adding a relation and deleting a relation. The addition of a design relation implies that there are potentially new paths among the important classes. This in turn may result in new relationships in the perspective. For example, if a developer adds a

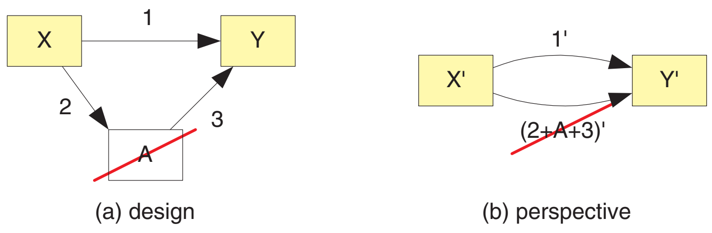

Fig. 7. Remove a helper class from a design

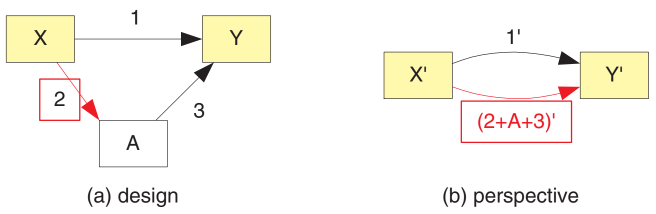

Fig. 8. Add/delete a relation in a design

relation between the Transaction and Account classes then this again affects the perspective "Guest may have Payment or Expense Transactions". The addition of the relation creates new paths among Guest, Expense, and Payment. Thus, we need to find all new paths between the important classes that pass through the new relation and abstract them.

Fig. 8 shows a general case for adding a new relation. If the new relation "2" is added between classes X and Y (Fig. 8 left) then we need to search for new paths among the reachable important classes (X and Y in this case) that pass through the relation. There is one such path (2+A+3)' which is then abstracted and added to the perspective.

The deletion of a relation is similar to the deletion of a helper class. If a developer deletes a relation from the design then incremental abstraction only removes those relations in the perspective that were abstracted from it. For example, if relation 2 in Fig. 8 is now removed from the design then the relation (2+A+3)' in the perspective must be removed also. While the removal of a design class may affect both perspective classes and relations, the removal of a design relation only affects perspective relations.

Upgrading and Downgrading a Class in the Design

Upgrading a class changes it from a helper class to an important class. Upgrading affects perspectives more than class and relation changes (recall Table 1). However, upgrading is not complex but simply the concatenation of class and relation changes. If a developer changes a helper class to an important class then three things have to be done. First, the newly-important class has to be added

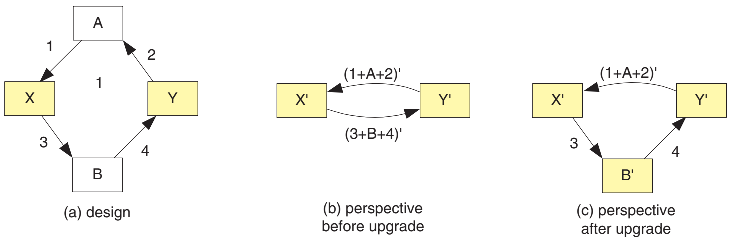

Fig. 9. Upgrade a class A

to the perspective (i.e., because important classes are not hidden). Second, all relations in the perspective that were abstracted from it need to be removed. And, third, all paths between it and other important classes must be found, abstracted, and added to the perspective.

For example, if developers are not only interested in “Guest may have Payment or Expense Transactions” but also in “Guest may have Transaction” then the helper class Transaction should be upgraded. The paths among the classes Guest, Expense, and Payment have to be removed from the perspective because they all pass through Transaction which is no longer a helper class. Transaction also needs to be added to the perspective and all paths between Transaction and Guest, Expense, and Payment need to be found and abstracted.

Fig. 9 (a) shows a general case for upgrading the class B in a design where classes X and Y are important. Before the upgrade, the perspective had two classes (X’ and Y’) to reflect the important classes of the design, and it had two relations between them to reflect the two paths through A and B (see Fig. 9 (b)). After class B is upgraded to an important class, incremental abstraction adds B’ to the perspective, eliminates all relations in the perspective that were abstracted from B (e.g., (3+B+4)’), and adds relations from B’ to all other important classes if it can find abstractable paths (see Fig. 9 (c)).

Downgrading an important class to a helper class is the exact opposite of upgrading a class (simply reverse the before/after picture in Fig. 9). The downgrading of a class removes that class from the perspective and with it all its relations to other important classes. Furthermore, it searches for paths among the important classes that pass through the downgraded class.

4 Validation

Software development changes have side effects. Yet these side effects are localized in that single changes in the design typically only cause small changes to their perspective(s). It is thus computationally wasteful to dispose of abstractions in their entirety simply because of small changes in the design. We evaluated the design models of four software systems (see Table 3) ranging between 9 classes and 127 classes to investigate this trade-off.

Table 3. Design Models used for Cases Study

	Design			Perspective	
	Classes	Relations	Model Size	Classes	Relations
Hms	9	9	104	3	3
Vod	65	199	1683	7	15
Visualizer	50	92	823	6	17
iTalks	107	127	1270	11	25

Fig. 10 (left) depicts the impact of design changes onto perspectives. As was discussed previously, there are essentially three types of changes of interest: up/downgrading, class changes (add/remove a class) and relationship changes (add/remove a dependency, association, or generalization). We subjected these models to over 800 random changes and observed their impact. For example, a change to a trace dependency in the iTalks design impacted in average 2.5 perspective elements (at least one and at most 8); or a change to a relationship impacted in average 0.9 perspective elements (0-5). The other three systems exhibited similarly small impact numbers which confirms our initial claim that design changes typically only have small impacts onto the perspectives for class abstraction. This observation is important for scalability.

While Fig. 10 (left) depicts up/downgrading, class, and relationship changes independently, there are situations where they occur together. For example, in IBM Rational Rose, the deletion of a class also causes the deletion of all its relationships and knowledge of its important/unimportant markers. Or the copy-and-paste action supported in many modeling tools allows a set of classes, relationships, and, perhaps, important/unimportant markers to be pasted at once. We thus conducted over 280 random changes that involved the deletion and creation of classes with relationships and important/unimportant markers. Fig. 10 (right) depicts the averaged results of these tests. For example, a typical deletion of a class in the vod system also deleted three design relations and 0.2 "important" markers (i.e., every 5th class deleted an important class). We refer to this deletion as a group deletion. In response, a group deletion changed elements in the perspectives. In the vod system, in average 0.2 classes and 2.3 relationships were changed per group deletion. These numbers demonstrate that the grouping of elements has only mild negative effects onto the impact of changes.

It must be noted however that a single copy-and-paste could involve an entire class structure which would then result in a change to the entire perspective. However, these kinds of changes are rare and so is their associated computational penalty.

Fig. 10 (left) also depicted another interesting observation. In all four systems, the up/downgrading had more severe effects onto the perspectives than relationship changes, which in turn had more severe effects than class changes. The differences are quite strong in that a relationship change impacts in average 3.5 times more perspective elements than a class change; and a up/downgrade impacts in average 2.9 times more elements than a relationship change (one order

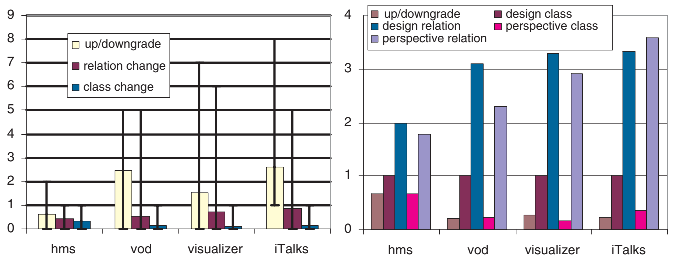

Fig. 10. Average, Min, Max Number of Perspective Changes per Design Change (left) and Grouping of Design Changes and Perspective Changes (right)

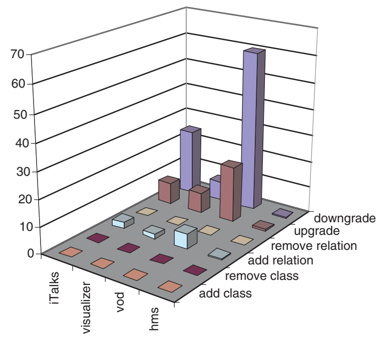

Fig. 11. Average number of Path Re-Evaluations for Design Changes

of magnitude difference). The differences are caused by the path re-evaluations that are part of class abstraction.

Fig. 11 depicts the average numbers of path re-evaluations for all four systems and it is obvious that up- and downgrading are particularly expensive. However, it must be noted that the path re-evaluations during incremental abstraction are cheap in comparison to the existing approach to abstraction (the batch abstraction of entire class structures). Also, we believe that trace changes are much less likely than relationship or class changes. After all, the four systems contain only 27 traces but almost 700 classes and relationships.

5 Conclusion

Abstraction, as in the simplification of complexity, plays an important role during software development. This paper demonstrated on UML class structures that it is possible and computationally feasible to maintain"life" perspectives that change instantly with system changes. The foremost advantage of instant transformation is that perspectives never become obsolete. Another advantage is that developers can observe system changes through their perspectives. The latter, in particular, is not common practice today because of the enormous cost of instant transformation. Yet, we believe that it is vital for the uninterrupted work flow of software developers to maximize the instant transformation of all kinds of information. Some of this capability is already transitioning into software development today. For example, many programming environments are capable of keeping the source code consistent with GUI modeling tools. We therefore see this work as another step in the same direction; and as the first step in doing so for software modeling and its model perspectives. The approach discussed in this paper is fully tool supported.

It is our future work to investigate the coupling of the instant and incremental abstraction discussed in this paper with the instant and incremental comparison for scalable consistency checking. That is, instead of checking the validity of entire models, we believe it is computationally much cheaper to check the consistency of models incrementally. Using instant and incremental abstraction to guide this instant and incremental consistency checking has not been attempted yet. It is our future work to investigate how to formally prove the consistency between batch and incremental transformation.

References

1. IBM Rational Rose. http://www.rational.com.
2. Matlab and Stateflow by Mathworks. http://www.mathworks.com.
3. R. Arnold and S. Bohner. Software Change Impact Analysis. In *IEEE Computer Society Press*. 1991.
4. B. Boehm, A. Egyed, J. Kwan, and R. Madachy. Using the WinWin Spiral Model: A Case Study. In *IEEE Computer*, pages 33–44. 1998.
5. B. Cheng, E. Y. Wang, R. H. Bourdeau, and H. A. Richter. Bridging the Gap Between Informal and Formal Approaches to Software Development. In *Proceedings of Software Engineering Research Forum, November 1995*. 1995.
6. A. Egyed. Semantic Abstraction Rules for Class Diagrams. In *Proceedings of the 15th IEEE International Conference of Automated Software Engineering (ASE)*, Grenoble, France, 2000.
7. A. Egyed. Automated Abstraction of Class Diagrams. In *ACM Transactions on Software Engineering and Methodology*, volume 11, pages 449–491, 2002.
8. A. Egyed. Compositional and Relational Reasoning During Class Abstraction. In *Proceedings of the 6th International Conference on the Unified Modeling Language (UML)*, pages 121–137, San Francisco, USA, 2003.
9. A. Egyed and B. Balzer. Integrating COTS Software into Systems through Instrumentation and Reasoning. In *Journal on Automated Software Engineering (JASE), accepted for publication.*
10. A. Egyed and P. Kruchten. Rose/Architect: A Tool to Visualize Architecture. In *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS)*, 1999.
11. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

12. F. D. Racz and K. Koskimies. Tool-Supported Compression of UML Class Diagrams. In *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML)*.
13. W. Roll. Towards Model-Based and CCM-Based Applications for Real-Time Systems. In *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 75–82, Hakodate, Hokkaido, Japan, 2003.
14. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley.
15. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.