

Emerging Design: New Roles and Uses for Abstraction

Christopher Van der Westhuizen, Ping H. Chen, and André van der Hoek

Department of Informatics
University of California, Irvine

Irvine, California 92697-3440, U.S.A.

+1 949 824 6326

{cvanderw, pchen, andre}@ics.uci.edu

ABSTRACT

Most abstractions in software engineering are used for one of two purposes, either 1) for *guidance*, in which an abstraction created up-front serves as a roadmap for the next activity, or 2) for *understanding*, in which an abstraction serves to explain the current state of the system at a given point in time. In either case, the abstraction tends to be static: once it has been created, it is not updated very often. Our research distinguishes itself by developing a dynamic abstraction, emerging design, that both guides and helps in understanding, while still able to fulfill new roles in the development process. In this paper, we will focus on the following three roles: (1) *coordination*: allowing developers to understand how their work influences that of others and vice versa, (2) *detecting design decay*: preventing unintended, undiscovered, and unauthorized design changes, and (3) *project management*: knowing which parts of the code are stable, incomplete, or in flux.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques – *computer-aided software engineering, user interfaces*; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement – *restructuring, reverse engineering, and reengineering*; H.5.3 [Information Interfaces and Presentation]: Group and Organizational Interfaces – *collaborative computing, computer-supported cooperative work*.

General Terms

Design, Management, Documentation.

Keywords

Design, reverse-engineering, coordination, awareness, abstraction, emerging design

1. INTRODUCTION

Software abstractions have long been used during software development for two main roles: guidance and understanding. First, the

abstraction can be generated up front as a guide to a particular activity (usually in the next phase in the lifecycle). Usage scenarios, for example, are a kind of system abstraction that can be used to help elicit further requirements as well as aid in developing the system design [7,12,14]. Secondly, a developer can use an abstraction to help understand an implemented system. Reverse engineering can be used to produce system models that aid developers in understanding the interactions that exist in a software system [5].

Regardless of the purpose of the abstraction, the result is more often than not a static document. The abstraction is static in the sense that it only captures the snapshot at the time at which the abstraction was generated. Unfortunately, continuous updating of the abstraction as the system evolves is often a cumbersome task and thus the abstraction is rarely updated. Therefore, an abstraction whose initial role was to guide system development cannot be successfully used to understand the system should the abstraction and the system fall out of sync with one another.

Like other abstractions, design can be used up front to guide implementation effort and assist in understanding an existing software system. Since a design illustrates the interactions among the various modules, the design can be used to assist in dividing up the implementation effort along logical module boundaries [3]. Similarly, when developers need to modify a system during maintenance, they can study the design document in order to gain an understanding of the high-level structure of the system and its various interactions [11]. Design is also often a static representation, unable to evolve automatically with the system [15]. There are a few noteworthy exceptions [1,10] that dynamically update the design as the code evolves. This is a critical step in moving from an abstraction that serves a singular role to one that can help in both understanding and guidance.

In this paper we propose a new dynamic abstraction that not only serves both roles of guidance and understanding, but also allows us to explore new roles for abstraction. The additional roles that we are most interested in exploring are:

1. *Coordination*. Allowing developers to be continuously aware of how their code relates to other developers' code. They can then see how their code depends on others as the system evolves.
2. *Detecting design decay*. Highlighting where the implementation diverges from the original design.
3. *Project management*. Allowing one to understand: what parts of the code are finished, what parts are incom-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ROA '06, May 21, 2006, Shanghai, China.

Copyright 2006 ACM 1-59593-085-X/06/0005...\$5.00.

plete, and in which tasks various developers are currently involved.

We call this new abstraction “emerging design,” and at its heart it is essentially an up-to-date representation of the design as it exists in the code. Whereas a conceptual design is a separate document that is typically created by a small set of architects or designers before any code is written, the emerging design is the incarnation of that design in the code as implemented by all of the involved developers. The emerging design comes into existence as the developers implement each part of the code and will change automatically as the code evolves. Furthermore, it can be overlaid on top of the conceptual design, thus highlighting which portions of the system have been implemented. We are currently integrating the concepts of emerging design and conceptual design into an Eclipse plugin called Lighthouse.

The paper is organized as follows. In Section 2 we explore how emerging design fulfills the three roles of coordination, detecting design decay, and project management. Section 3 looks at related areas of work. Finally, we conclude and describe our future work in Section 4.

2. APPROACH

The abstraction that we are introducing is emerging design, which is an up-to-date representation of the design as it exists in the code. As the individual developers change their code, every developer’s emerging design view is updated appropriately. The view is updated in real-time without the need to check in the changes into a versioning system. For instance, if a developer adds a method to a class then the emerging design view will be updated to reflect this change on all of the developer’s views. Therefore, each developer is presented with an accurate representation of the system as it evolves.

Figure 1 presents an initial version of emerging design. Note that the emerging design is very similar to a UML-class diagram in that it has three major sections: class name, fields, and methods. We chose UML class diagrams as the basis for emerging design since their representation remains close to the source code. However, the emerging design diagram has been enhanced with evolution information. Emerging design uses arrows to indicate related changes to the same code element. As can be seen in Figure 1, there is an arrow leading from the first item, STORE, to the second item, ~~STORE~~, indicating that the class was deleted. A second arrow exists leading again from the first occurrence of Store to a third

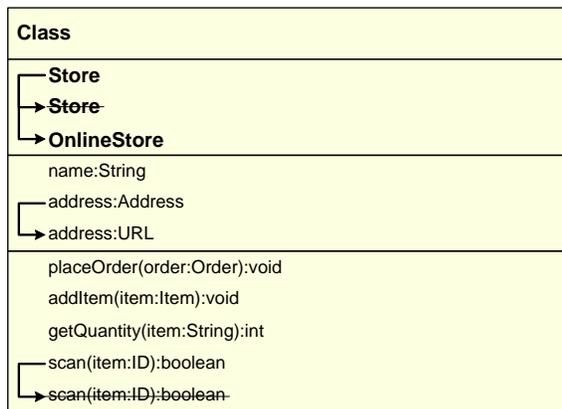


Figure 1. Emerging Design.

item, ONLINESTORE, indicating that the class was renamed. These two changes were made in parallel since both arrows originated from the first item, STORE. These parallel changes can indicate potential conflicts that developers need to be aware of and correct.

Immediately, we see that the emerging design can fulfill the role of understanding as it provides an accurate design representation of the system at the current point in time. Although the emerging design shown in Figure 1 does not support guidance in the sense of dividing implementation effort, it is still able to guide the developers by providing the interfaces and modules that are currently available. This in turn helps developers by allowing them to see which classes exist to be instantiated and which methods can be called.

In the following subsections, we explore ways in which we can augment the initial version of emerging design so that it can be applied to the three new roles that we wish to explore, namely coordination, detecting design decay, and project management.

2.1 Coordination

Imagine a situation in which two developers, John and Susan, part of a larger team, have each been assigned a task that involves changing some set of files. They each use the available configuration management system to first populate their respective workspace with the necessary files, and then begin to make their changes. John finishes relatively fast, and checks his changes into the repository. Susan takes a while longer, but eventually completes the task. When attempting to check in the result, however, Susan finds that she made modifications that conflict with the modifications by John. Some of the conflicts are within the same files, others span different files but lead to compilation and testing problems. While some of these conflicts can be resolved using an automated merge tool, others must be resolved by hand, a time-consuming and complicated task.

From the above scenario, one immediately notices that a critical requirement for supporting coordination is awareness. Developers should be aware of which other developers are contributing to the project and what changes they are making (or have made) to the code. Furthermore, it is useful for developers to be aware of whether or not changes have been committed to the repository, been checked out by other people, or are still in only the author’s workspace.

Figure 2 shows the emerging design as enhanced with two additional columns containing coordination information. The first column indicates the status of each specific change: whether the change is any combination of (1) in my workspace, (2) in the repository, or (3) in other people’s workspaces. A line is drawn connecting all three dots when all the developers have checked out the change. For example, the ADDITEM method exists in both the repository and in some other developers’ workspaces, but has not been checked out in the local workspace. On the other hand, the PLACEORDER method exists in the repository, this local workspace, and all other workspaces as denoted by the line connecting all three dots.

The set of columns on the far right includes a picture for each of the authors contributing to the class and also symbols denoting what types of changes they have made. The three symbols used are plus, minus, and triangle, and they represent addition, subtraction, and modification, respectively. Affixed to each of the change

Class	My Workspace	Repository	Other Workspaces		
<ul style="list-style-type: none"> Store → Store → OnlineStore 				+	
<ul style="list-style-type: none"> name:String → address:Address → address:URL 	●	●	●	+	▲
<ul style="list-style-type: none"> placeOrder(order:Order):void addItem(item:Item):void getQuantity(item:String):int → scan(item:ID):boolean → scan(item:ID):boolean 	●	●	●	+	▲

Figure 2. Coordination Annotations.

symbols is an arrow that, over time, rotates around in a clockwise fashion. The position of the arrow indicates roughly how long the associated change has been in effect. For example, when the ADDRESS field was first added by developer number one, an addition symbol was added to that developer’s column. When the ADDRESS field’s type was later changed to URL by the second developer else, a modification symbol was added to that developer’s column. The arrow associated with the addition has rotated almost entirely around, whereas the arrow associated with the modification has not rotated very much. This signifies that the addition took place a lot earlier than the recent modification of the field’s type.

With these coordination mechanisms the developers in the scenario would be able to immediately notice the changes that the other developers were making. Therefore, when Susan notices in the emerging design that the changes made by John may conflict with her changes, they can quickly discuss and resolve the issues before either of their changes are checked in. When Susan notices that John checked in his changes, she knows that his code is ready to be checked out and used by other developers. At this point Susan can merge her code with John’s changes thus minimizing any conflicts that she would otherwise introduce later. The severity of the conflict is reduced because the use of emerging design breaks developer isolation and allows them to coordinate their tasks in a timely fashion.

2.2 Detecting Design Decay

A second role of emerging design is to detect design decay. The following scenario illustrates a problem that can result from design decay. Bob, part of a larger team, has been assigned a task that involves making changes to a part of the system with which he is relatively unfamiliar. Unfortunately, the original developer of that part of the system has just left the company and is no longer available for questions. However, Bob remembers that a detailed UML design was made before the system was implemented, including some notes on rationale for some of the structural decisions. Bob consults the design, happily finds the information needed, and uses the configuration management system to check out the necessary files into his workspace. Ready to work, however, Bob finds that the design as studied can no longer be found in the source code. Major elements are missing, other elements have changed, and no rationale is provided as to why this

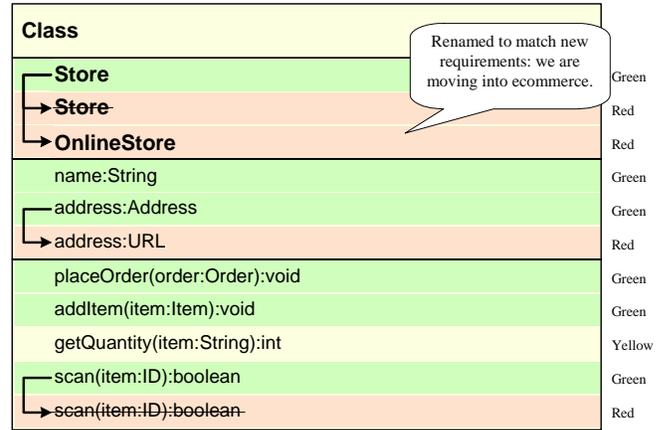


Figure 3. Detecting Design Decay.

happened and whether or not it happened intentionally or accidentally. Bob is left to study the source code in detail to try and understand how to make his changes, an unfortunate situation.

As can be seen from the above scenario, a critical requirement for addressing design decay is a mechanism for indicating whether or not the current implementation matches the design and, if not, where the deviation takes place (all assuming that an original design is available). Figure 3 depicts the emerging design overlaid on top of the original conceptual design. This overlay is shown by color coding each item in the emerging diagram based on whether or not it exists in the original design. If the item was not specified in the conceptual design, the row will be shaded light red, whereas the row will be shaded light green if the item is in the original design. Finally, if an item in the original design has not been implemented yet, it will keep the default background color of yellow.

Based on the color coding, the developers can, at a glance, see if the current implementation deviates from the originally specified design and by how much. Additionally, in the case that design deviations are knowingly performed by the developers, they are able to annotate the emerging design with rationale explaining why the deviations occurred. As can be seen in Figure 3, a developer used annotations to justify the choice for renaming the class. Developers are then able to discuss the deviations and vote on whether or not to keep the deviations. If the developers agree that the deviations improve the overall system design they will be able to automatically promote those agreed upon changes into the conceptual design.

Through the use of emerging design, the developer is able to quickly gain an understanding of the system’s current structure and where the current implementation differs from the original design. This saves the developer from having to carefully study the code, which can be a time-consuming task, in order to find out where to make their changes. In addition, had the original developer used emerging design in the first place, it is likely that he or she would have kept the original design and the source code in sync with one another by promoting deviations as they occurred.

2.3 Project Management

Finally, we envision that the emerging design can be applied in a project management setting. Specifically, we are interested in

Class	My Workspace	Repository	Other Workspace	Dev 1	Dev 2	Dev 3	Status
Store							Green
→ Store							Red
→ OnlineStore	●	●	●				Red
name:String	●	●	●				Green
→ address:Address		●	●				Green
→ address:URL	●						Red
placeOrder(order:Order):void	●	●	●				Green
addItem(item:Item):void		●	●				Green
getQuantity(item:String):int							Yellow
scan(item:ID):boolean			●				Green
→ scan(item:ID):boolean	●	●					Red

Figure 4. Project Management.

tracking project progress and overcoming the difficulty seen in the following scenario. A team of developers, working on a large project, is having its status meeting to determine how things are going. Each developer is asked to provide an update on their part of the code resulting in comments such as “I just fixed a major bug, and it seems the code is stable now,” and “I have progressed significantly, and am about 90% done.” Overall, the team leaves the meeting content and confident that they are almost done with the project. The reality of it turns out to be quite different. They did not see that two parts of the project had not been integrated as needed, were overly optimistic with respect to the impact of some agreed-upon design changes, and failed to recognize one part of the code was very unstable and had been so over the past few weeks. While useful, the meeting did not identify some critical issues that are likely to cause problems down the road.

From the scenario described above, the critical requirement of project management is project visibility. The developers should be aware of what changes have been made (and when), as well as which parts of the system still need to be implemented. All of these features can be achieved via the combination of coordination information and overlaying the emerging design on top of the conceptual design as seen in Figure 4 (in essence combining Figure 2 and Figure 3).

When emerging design is overlaid on top of the conceptual design, it is possible to gauge which parts of the project have been completed and which have not. For example, if most of a class’ rows are not shaded yet, it would mean that implementation on the class is far from complete. Furthermore, recency indicators show which parts of the code are still in flux. Code that has not been modified for quite some time is likely to be more stable than code which is currently undergoing changes. Lastly, since the emerging design is an accurate reflection of the code, the impact of agreed-upon design changes can be more easily analyzed.

2.4 Lighthouse

We are currently exploring the concepts of emerging design and conceptual design in a tool called *Lighthouse*, a plugin for Eclipse [4]. Lighthouse monitors a developer’s implementation through the use of Eclipse listeners and correspondingly updates the emerging design of that user and notifies other Lighthouse clients of the change so as to keep everyone in sync. A distinguishing feature of Lighthouse is that the visualization is updated without

the need for saving files or checking in and out the changes. This provides all the developers with a real-time view of the development as it evolves.

Our vision is to have a side-by-side view of the code and emerging design as shown in Figure 5. For the emerging design view to be most effective it is imperative that developers have constant awareness of changes being made to it. A single monitor setup would be far from ideal since developers would have to constantly perform context-switching from one application (such as their development environment) to Lighthouse in order to see the emerging design view. While the emerging design view is not in focus, the developers could miss crucial real-time updates. To this end we envision developers using a dual-monitor setup in which a main monitor would have their primary coding environment and an auxiliary monitor would be dedicated to Lighthouse. Therefore, the developers are able to maintain peripheral awareness of ongoing changes made to the project by the team members.

3. RELATED WORK

Our work is related to a number of areas including version control, coordination, and reverse engineering. While our proposed approach illustrates how the system has evolved, it does not allow developers to manage the versions (e.g. reverting back to a previous version). Since version control is not a focus of our work, in this section we focus on the areas of coordination and reverse engineering.

The coordination tool that is most similar to our work is Palantir [13]. Palantir is able to provide real-time awareness of the files that developers are changing and how severe those changes are without requiring developers to first check in those changes. In this respect, Palantir is similar to Lighthouse in that a developer’s view contains information pooled together from all the other developers’ workspaces and not just one’s own. However, Palantir differs from our work in that it provides a much lower-level abstraction mechanism, dealing with files, whereas our work utilizes the higher-level abstraction of design.

Jazz [2] is another workspace awareness tool that is integrated into Eclipse. It specifically provides information of which arti-

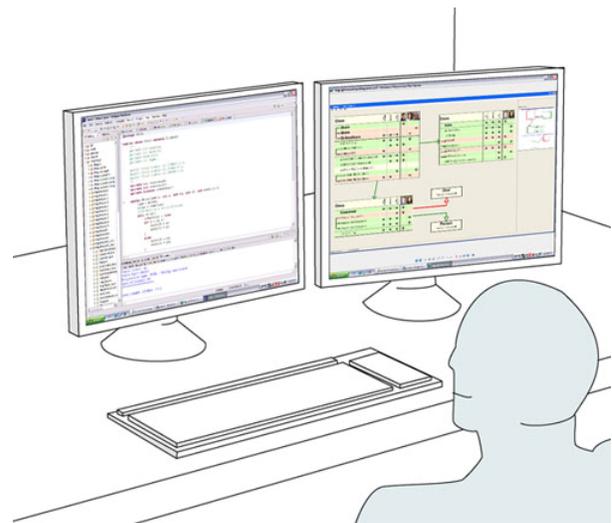


Figure 5. Side-by-Side View of Code and Emerging Design.

facts are checked out, which are undergoing changes, and which have been checked in. Additionally, Jazz provides presence awareness of developers and chat facilities for communication embedded within the IDE. Unlike Jazz, Lighthouse does not provide chat support and presence awareness for developers. However, where as Jazz uses files as the primary level of abstraction, Lighthouse uses design as the abstraction.

There are a number of reverse engineering tools that are able to generate a UML class diagram from the source code [1,6,10]. While some of these tools do not automatically update the class diagram as the source code evolves, a number of them do, including Omondo EclipseUML [10] and Green [1]. What these tools provide is essentially an update-to-date design based on the local source code without any annotations. Our work takes this idea a step further in that the design generated is not just based on the local source code, but the code of all the developers. Furthermore, we annotate the design with evolution and coordination information. Lastly, we are able to compare our abstraction against a conceptual design provided by the developers so as to identify where the source code differs from and agrees with the conceptual design.

While Omondo EclipseUML and Green are not able to compare the current design with an initial design, there are a number of design differencing and merging tools available [8,16]. These tools can be used to calculate the difference between a provided model and an initial design in order to understand the evolution of the system. Unlike emerging design, however, these tools are not automated and require the developer to manually generate a model and perform a diff.

The work that is most similar to ours is Gail Murphy's work on reflexion models [9]. With the reflexion models tool, a developer is required to provide the tool with a high-level model of the system as well as a mapping specifying how the model maps to the source code. The tool then outputs a reflexion model that shows where the engineer's high-level model differs from and agrees with the model of the source. This is similar to Lighthouse in that it accepts an initial high-level design model that is then used to identify differences between that model and the current source code. Our notion of emerging design is different in that it does not always require an initial high-level design to be effective. Without a conceptual design, emerging design can still be used to aid in coordinating developers' efforts and, at the very least, provides them with a design where previously there was none. Additionally, emerging design is a continuously updated representation whereas reflexion models are a one-time snapshot and, should the developer wish to understand the system at a later date, the developer is required to recompute a new reflexion model.

4. CONCLUSION AND FUTURE WORK

In this paper, we presented the novel concept of emerging design, which is a continuously-updating design representation of the code. Whenever a developer changes part of his/her source code, that change is propagated to everyone's emerging design view. Therefore, everyone maintains an accurate design representation of the system without needing to check-in or check-out the changes. We envision emerging design to not only be able to satisfy traditional roles of abstraction, guidance, and understanding, but also to support new roles such as coordination, detection of design decay, and project management.

Currently we are working on the initial prototype of Lighthouse to demonstrate the benefits of emerging design. While the prototype will implement the emerging design as described here, there are still a number of open questions to explore. For instance, what would be the right "level" of abstraction for emerging design? While we currently use UML class diagrams, we are planning to investigate different levels of abstraction for emerging design, such as software architecture and patterns. Another issue is the scalability of the emerging design. Specifically how can we display a large number of elements at once without overwhelming the user with too much information. We plan to investigate a number of user interface techniques such as fish eye view or advanced filtering methods. Thirdly, what other kinds of "coordination information" could we use to annotate the emerging design in order for developers to more effectively coordinate? In addition to the annotations that indicate what kinds of changes were made and by whom, we plan to investigate such concepts as "spheres of influence" that indicate that changing one part of the code would have an influence on other classes, thus warning developers of the potential impact of their change. Lastly, what are the right kinds of interaction and coordination mechanisms to present to the developers? Currently we plan to allow developers to annotate design elements with comments, vote on changes, and promote changes into the original design. We also intend to investigate how we can push the abstraction much further and incorporate configuration management functionality turning it into a central coordination portal for all technical coordination needs.

5. ACKNOWLEDGMENTS

This research has been funded by a 2005 IBM Eclipse Technology Exchange grant, a 2006 IBM Technology Fellowship, and by the National Science Foundation under grant numbers CCR-0093489 and IIS-0205724. We would like to thank Anita Sarma for her insightful input in the project as well as the rest of our research group for their valuable feedback.

6. REFERENCES

- [1] C. Alphonse and B. Martin. Green: A Pedagogically Customizable Round-Tripping UML Class Diagram Eclipse Plug-in. Proceedings of the Eclipse Technology Exchange Workshop at OOPSLA, 2005.
- [2] L.-T. Cheng, et al., Building Collaboration into IDEs. Edit -> Compile -> Run -> Debug -> Collaborate? ACM Queue, 1(9): p. 40-50, December/January 2003-2004.
- [3] C.R.B. de Souza, et al. How a Good Software Practice thwarts Collaboration - The Multiple roles of APIs in Software Development. Proceedings of the Foundations of Software Engineering, 2004.
- [4] Eclipse, Eclipse, <http://www.eclipse.org>
- [5] D.R. Harris, H.B. Reubenstein, and A.S. Yeh. Reverse Engineering to the Architectural Level. Proceedings of the International Conference on Software Engineering, 1995: p. 186-195.
- [6] IBM, Rational Rose, <http://www-306.ibm.com/software/rational>

- [7] M. Jarke, X.T. Bui, and J.M. Carroll, Scenario Management: An Interdisciplinary Approach. Requirements Engineering Journal, 3(3-4): p. 155-173, 1998.
- [8] A. Mehra, J.C. Grundy, and J.G. Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. Proceedings of the Automated Software Engineering, 2005.
- [9] G. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. Proceedings of the Third Symposium on the Foundations of Software Engineering, 1995.
- [10] Omondo, Omondo EclipseUML, <http://www.omondo.com>
- [11] D.L. Parnas and P.C. Clements, A Rational Design Process: How and Why to Fake It. IEEE Transactions on Software Engineering, 12(2): p. 251-257, February 1986.
- [12] C. Rolland, et al., A proposal for a scenario classification framework. Requirements Engineering Journal, 3(1): p. 23-47, 1998.
- [13] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: Raising Awareness among Configuration Management Workspaces. Proceedings of the Twenty-fifth International Conference on Software Engineering, 2003.
- [14] A. van Lamsweerde and L. Willemet, Inferring Declarative Requirements Specifications from Operational Scenarios. IEEE Transactions on Software Engineering, 24(12): p. 1089-1114, December 1998.
- [15] K. Wong, et al., Structural Redocumentation: A Case Study. IEEE Software, 12(1): p. 46-54, January 1995.
- [16] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-Oriented Design Differencing. Proceedings of the Automated Software Engineering, 2005.