# CASI: Preventing Indirect Conflicts through a Live Visualization

Francisco Servant, James A. Jones, André van der Hoek
University of California, Irvine
Department of Informatics
Irvine, CA, U.S.A. 92697-3440
{fservant, jajones, andre}@ics.uci.edu

## ABSTRACT

Software development is a collaborative activity that may lead to conflicts when changes are performed in parallel by several developers. Direct conflicts arise when multiple developers make changes in the same source code entity, and indirect conflicts are produced when multiple developers make changes to source code entities that depend on each other. Previous approaches of code analysis either cannot predict all kinds of indirect conflicts, since they can be caused by syntactic or semantic changes, or they produce so much information as to make them virtually useless. Workspace awareness techniques have been proposed to enhance software configuration management systems by providing developers with information about the activity that is being performed by other developers. Most workspace awareness tools detect direct conflicts while only some of them warn about potential indirect conflicts. We propose a new approach to the problem of indirect conflicts. Our tool CASI informs developers of the changes that are taking place in a software project and the source code entities influenced by them. We visualize this influence together with directionality and severity information to help developers decide whether a concrete situation represents an indirect conflict. We introduce our approach, explain its implementation, discuss its behavior on an example, and lay out several steps that we will be taking to improve it in the future.

## Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming environments – *Graphical environments.* D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *version control*. D.2.9 [**Software Engineering**]: Management – *software configuration management.*

## General Terms

Algorithms, Reliability, Human Factors.

## Keywords

Software configuration management, parallel work, conflicts, workspace awareness, software visualization.

## 1. INTRODUCTION

One of the biggest challenges that the field of software engineering faces today is collaboration. Most software projects are built by teams of developers. One of the main tools that these developers use in order to coordinate their work is Software Configuration Management (SCM) [1,5]. Most of these systems allow developers to make changes in parallel, but, as a result, conflicts may occur.

Conflicts are classified in two groups [11]: direct conflicts and indirect conflicts. Direct conflicts happen when two or more developers modify the same version of a source code file at the same time. When they decide to save their new version to the repository, changes may overlap and need to be integrated. An indirect conflict is an error that is produced as a result of two changes that are performed in parallel by two different developers in two different source code files. In this case, both developers will be able to save their changes correctly, but the final result might be a system that is inconsistent. These inconsistencies could lead to compilation errors, build errors, runtime errors, or just erroneous situations in which they do not necessarily receive an error message, even though the program exhibits an unexpected behavior.

Conflicts are typically detected after they have been introduced. Direct conflicts are normally detected when a developer decides to save (check in) the changes to the repository. In the case of indirect conflicts, however, the amount of time that passes before they are detected may vary. Sometimes they will be detected when compiling the application, but other times they might go unnoticed until the bug that they caused manifests itself in the field. This is the reason why solving the problems introduced by indirect conflicts can be a very complex and time consuming task, especially if a long time has passed since they were introduced.

The central problem addressed in this paper is to work towards providing developers with tool support through which they can detect and perhaps even avoid altogether potential indirect conflicts. The idea is that, if developers can detect a potential conflict earlier, it may be much easier and less costly to resolve it then, instead of at a much later time.

Several research projects address the issue of conflict detection by raising awareness among developers of their activity [14]. These approaches let developers put their activities in the context of those of others, thus enabling them to proactively plan and execute their activities in a more informed way in order to reduce their interference with the activities of others. Most of these approaches simply highlight changed source code entities [2,4,10] and require developers to judge when a conflict is introduced.

To date, awareness solutions either only detect direct conflicts [2,4], only warn about indirect conflicts after they have been introduced [13], focus on a limited set of potential indirect conflicts [11,12], or provide no indication about how risky a potential indirect conflict is [8]. In an attempt to fill these gaps, we introduce our tool CASI (Conflict Awareness through Spheres of Influence). A Sphere of Influence is a visualization that contains the set of source code entities that may be influenced by a set of changes, as well as other characteristics of that influence, such as its severity. CASI dynamically shows developers the source code entities that are being modified by any developer together with their Spheres of Influence. Just like changes, the Spheres of Influence evolve over time, and they normally grow as developers make more changes.

Developers will be able to use the Spheres of Influence to help them judge whether their changes might produce conflicts. This way, they will have more information to plan their changes in order to try to avoid conflicts. As an example, they may decide to hold off changes in some areas of the code until they are not influenced by other developers. They may also decide to apply their changes in a different way to try to reduce the amount of source code entities influenced by them.

At some point, the Spheres of Influence of several developers might overlap. Overlaps inform developers of who might be affecting their changes or whom they may be affecting with theirs. Being aware of overlaps also helps developers to judge if an indirect conflict may have been inserted. Once a developer decides that there might be a conflict, the possible reactions may vary, such as communicating with the conflicting developer via IM or telephone, or checking the source code themselves.

With our approach presented here, we contribute a new concept, the Spheres of Influence, and a novel heuristic algorithm to calculate them. Our algorithm is designed to warn about both syntactic and semantic indirect conflicts, since it is inspired in program slicing. We also contribute the notion of severity in the Spheres of Influence. With the severity, we intend to provide an indication of which source code entities are more likely to be involved in an indirect conflict.

## 2. APPROACH

Previous awareness approaches, such as Palantír [2] or Jazz [4], perform precise analyses over the source code to determine conflicts. These approaches are ineffective when trying to warn about indirect conflicts originating in semantic changes. Such precise analyses can find mismatching definitions and uses of APIs, but they cannot predict, for instance, run-time problems by only analyzing (partial and parallel) code changes.

Thus, we provide a different kind of solution. The goal of CASI is to help developers detect *situations that may turn into conflicts* as early as possible by providing them with an advanced visualization of the activity of other developers. Consequently, they will be able to react by taking the necessary proactive steps to prevent these conflicts from being introduced. Our assumption is that, even though developers would periodically monitor this visualization and at times undertake action to reduce the number of potential indirect conflicts, this proactive work involves less effort than resolving indirect conflicts after their introduction and discovery.

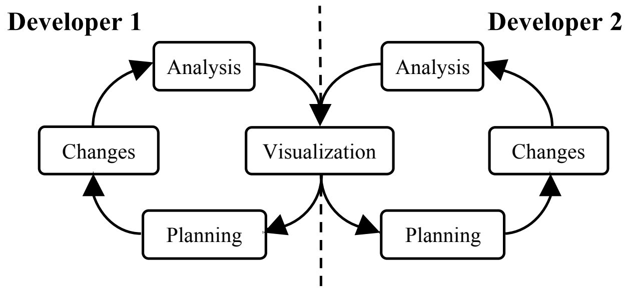


**Figure 1. Approach of CASI.**

This process is represented in action for two developers in Figure 1. It cycles through four steps: code changes (performed by developers), analysis (computed by our tool), visualization (rendered by our tool), and planning (performed by developers). These steps are described in more detail in the sections below.

### 2.1 Changes

CASI detects live code changes as they are being made in the developers' environments. Capturing changes *some time after they happen* would result in a reduction of the developers' ability to take proactive decisions about how to perform their changes. Therefore, the immediacy of this detection is a key factor for our approach.

CASI monitors changes in fields, methods, classes, and interfaces. For simplicity, we use the term *source code entity* to designate any of these. The addition, deletion, or modification of a source code entity is considered a change. In addition, we consider that a source code entity is modified when its set of relationships with other source code entities changes.

### 2.2 Analysis

To approximate where indirect conflicts may occur, our analysis informs developers of how "far" the influence of their changes reaches in the code base, using the changed source code entities as the seed. Program slicing [16] is an approach that can be used for such a purpose; however, slices generally cover vast swaths of code and do not provide any degree of *distance* or *strength* of the dependency. We want the influence of code changes to taper off as artifacts are further away from them in the dependency chain. Additionally, we want this information to help developers in judging the risk of a situation to turn into an indirect conflict by giving them some indication of *severity* or *distance from the change*.

CASI's technique operates over the entity-relationship metamodel used by dependency slicing [3], which provides the set of dependencies among all source code entities, including transitive ones. We use the parsing algorithm from dependency slicing to get the set of source code entities and relationships that represent the source code of the application. Then, we traverse this dependency graph in order to obtain all the reachable source code entities from each changed source code entity. Each reachable source code entity is an *influenced source code entity* as a result of the *influence* that the changed source code entity holds over it.

When a source code entity is changed, it produces two kinds of influence: forward influence over source code entities that it uses, and backward influence over source code entities that use it.

Therefore, our algorithm traverses the dependency graph in both directions in order to calculate these two different sets of influenced source code entities.

Our algorithm also assigns a heuristic value of severity to each influenced source code entity, with which we intend to express a relative likelihood of a source code entity to be involved in an indirect conflict. We assume that the influence of a change declines as the influenced source code entity is further from the change in the dependency chain. Thus, the severity value is driven by the distance from the influenced source code entity to the changed source code entity and by the kinds of relationships that form the chain of dependencies between them.

During the traversal of the dependency graph, the changed source code entity is assigned a severity of 1.0, and every time that we visit a new one, we assign it a severity value $s = s' \cdot w$, where $s'$ is the severity of the previous source code entity and $w$ is the weight of the relationship between them. Weights take values in the range 0.0 – 1.0, and different weights are considered for different relationships. As an example, we assign higher weight to CALLS relationships than to OF TYPE relationships because we consider that the likelihood of being involved in an indirect conflict is higher for CALLS relationships than for OF TYPE relationships. These weights are currently based on our intuition, but are also configurable; they are the subject of our ongoing research.

Commonly, a source code entity is influenced by many changes or by the same change along multiple paths. In such cases, all the severity values along each path are combined to calculate the severity of that influenced source code entity. We combine the severity values at a target source code entity along each path to recognize that influence traversing along multiple paths may be more likely to affect the target than a source code entity reachable by only one path. Thus, whenever a source code entity is influenced by two different changes, we assign it a severity value $s = (a + b - (a \cdot b))$, where $a$ and $b$ are the two different severity values that would correspond to the influence of each path. If more than two changes influence the same source code entity in the same direction, we first calculate the combined severity for the first two, and then iteratively apply the formula to the combined severity until all the severities have been processed.

To demonstrate how severity propagates in a program, consider the example depicted in Figure 2. The changed method (represented as the leftmost node) defines a field's value, (represented as the topmost node) which is of the type of the rightmost node. The changed method calls another method (the bottommost node) that throws an exception of the type of the rightmost node. The weight and type of each of the relationships joining the source code entities is depicted on the edge labels. The severity of the changed method is assigned a value of 1.0. Along the top path, the assigned field is given a severity value of $1.0 \cdot 0.7 = 0.7$. The called method is given a severity value of $1.0 \cdot 0.8 = 0.8$. The severity value of the class is calculated as $(0.7 \cdot 0.4) + (0.8 \cdot 0.1) - (0.7 \cdot 0.4) \cdot (0.8 \cdot 0.1) = 0.3376$.

In order to provide an up-to-date visualization, our algorithm is executed dynamically to update the set of influenced source code entities as developers make changes.

## 2.3 Visualization

An important aspect of this kind of solution is that it has to be shown in a clear and non-intrusive way, so that it does not represent a distraction from the coding activity, and it is only invasive when it is relevant. In order to fulfill this requirement, we implemented CASI as a plug-in for the Lighthouse project [2]. Lighthouse, itself, is a plug-in for Eclipse [7] that shows the *Emerging Design: "an up-to-date representation of the design as it exists in the code"* [15]. This view is represented in a separate window as a UML-like class diagram, which is annotated with events in the source code entities that have changed. Within the class nodes, each change event is represented to the right of the affected source code entity by a change icon (a plus sign for addition, a minus sign for deletion and a triangle for modification) together with the name of the developer that produced it. The Emerging Design is dynamically updated as developers make changes. At any particular moment, the Emerging Design contains all the changes that have not yet been transferred to all the developers' workspaces, i.e., if a developer makes some changes and checks them in, these changes will not be cleared from the visualization of the Emerging Design until all developers have checked them out from the repository. Thus, the Emerging Design represents the union of all the differences between all developers' workspaces.

CASI's visualization enhances Lighthouse with influence events, and it also dynamically updates as developers make changes. An influence event signals an influenced source code entity. These events are represented to the right of the influenced source code entity by a double arrow icon together with the name of the developer that produced the influence. A double arrow pointing right represents forward influence and a double arrow pointing left represents backward influence. The color of the double arrow informs developers of the severity of the influence, ranging from yellow to red. Yellow corresponds to minimum severity and red corresponds to maximum severity. A developer's Sphere of Influence is composed of the set of all the influence events (forward and backward) that correspond to that developer. Examples of CASI's visualization are portrayed in Figures 4 – 6.

The direction of the arrows was chosen to ease the interpretation of the visualization. Thus, if a developer named Alice produces forward influence over the method getScreen(), the visualization displays "getScreen() >> Alice", which could be read as "If you change getScreen(), Alice may be affected". However, if a developer named Bob produces backward influence over the method execute(), the visualization displays "execute() << Bob", which could be read as "Bob's changes may be affecting the

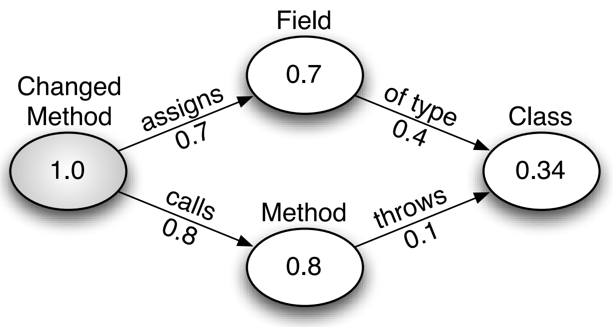
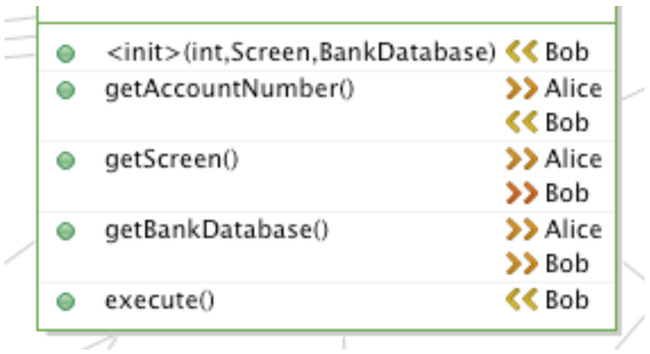


**Figure 2. Severity propagation among source code entities. The calculated severity is shown in the nodes.**



● <init>(int,Screen,BankDatabase) << Bob
● getAccountNumber() >> Alice << Bob
● getScreen() >> Alice >> Bob
● getBankDatabase() >> Alice >> Bob
● execute() << Bob

**Figure 3. Different types of influence in CASI's visualization.**

method execute()". These examples can be seen in Figure 3, which is zoomed in from Figure 6 and contains influence events with different directions and severities.

The influence, both forward and backward, is displayed for each developer so that they can see both whom they may be affecting and who may be affecting them. This way, both affected parties will have a chance to react in the event of an indirect conflict. We expect developers to normally focus their attention on the source code entities that they are editing and the close ones surrounding them. In such a case, they can see at a glance whether or not other developers' Spheres of Influence extend over their changes.

## 2.4 Planning

CASI enables developers to make intelligent decisions about how to implement their changes. Initially, their own Sphere of Influence can help them to better understand what parts of the code may be affected by their changes. This gives them hints about which source code entities they might want to review before checking in their changes in order to make sure that they will behave correctly after their changes have been performed. They can also use the severity indicator to decide which of the source code entities in their Sphere of Influence are worth reviewing.

Additionally, the Spheres of Influence corresponding to other developers can be used as a planning device. Developers often know where they will be making changes. The areas of the code that they intend to change may be included in the Spheres of Influence of others. If they are, developers can interpret this visualization as a risk of those areas actually being involved in an indirect conflict should they be modified. In this situation, developers may decide to: (1) hold off on their changes until the other developers have completed and checked in their changes, (2) communicate with the potentially conflicting other developers to avoid conflicts, or (3) perform the changes despite the warning. The severity indicator of the influenced source code entities is intended to help them make this decision.

If the number of dependencies involved in a set of changes is high, the corresponding Sphere of Influence can grow very large. A large size of the Spheres of Influence serves as a warning of high chances of them to overlap, which increases the risk of an indirect conflict being inserted. In this case, developers may decide to apply their changes in a different way. Consequently, they may reduce the number of source code entities influenced by their changes and/or the severity value assigned to them.

The Spheres of Influence may eventually overlap. Even though overlaps do not always signal the introduction of an indirect conflict, their presence should encourage developers to take action and talk to each other in order to understand their changes and whether they are compatible. This way, they may be able to avoid an indirect conflict even before it is introduced.

As different kinds of overlaps may arise, developers can use the severity and direction of the overlapping influences to judge how high the risk of indirect conflict is and whether communicating with others is necessary. In general, a higher severity in the overlapping influences denotes a higher risk of indirect conflict. An overlap of several backward influences over the same source code entity also increases the risk of that source code entity being involved in an indirect conflict if it is modified. An overlap of several forward influences over the same source code entity tells which developers would be involved in a potential indirect conflict if that source code entity were modified. Finally, the overlap of a forward influence with a backward influence means that the two developers involved in the overlap have made changes in areas of the source code that were part of the Sphere of Influence of the other developer. This means that somewhere in the visualization, there is an overlap between a developer's change and the other developer's Sphere of Influence. This is the most serious overlap and the one to which we expect developers to be most likely to react.

## 3. EXAMPLE

We tested CASI in some situations in which we knew that an indirect conflict was being introduced. These scenarios affected a small code base: the ATM example taken from a programming book [6]. In this section, we describe one of the ATM change scenarios step by step to show how CASI and its visualization provide guidance to developers.

In this example, developers Alice and Bob make changes in parallel over the source code of the ATM application. This code base is small enough for our example to be simple to understand, but of enough size for the visualization of CASI to be representative. The structure of the source code can be seen in Figure 4. The *ATMCaseStudy* class contains the main method of the application, which executes methods in the *ATM* class. The *Transaction* class contains the common functionality for all transactions, and *BalanceInquiry* and *Withdrawal* inherit from it. *Withdrawal* uses the *CashDispenser* class to execute its functionality. Both transactions *BalanceInquiry* and *Withdrawal*, as well as their parent class *Transaction* use the *BankDatabase* class. *BankDatabase* contains a list of all the accounts, each of which is represented by the *Account* class. Finally, the *Keypad* and *Screen* classes contain the input and output functionality of the application.

Alice's task is to modify the application so that the balance information is stored in cents instead of dollars. Currently, the balance information is stored in the *availableBalance* and *totalBalance* fields inside the *Account* class. These fields represent the dollar amount with decimals as a value of type *double*. The Java documentation recommends using a different type of variable for currency [9], so she plans to change the type of these variables to *long*.

Bob is in charge of adding a new transaction to allow deposits. He plans on creating a *DepositSlot* class for the functionality of the deposit slot, and a *Deposit* class for the functionality of the transaction. Thus, *Deposit* will inherit from *Transaction*. Since Bob is not aware of Alice's plans, he expects the balance stored in

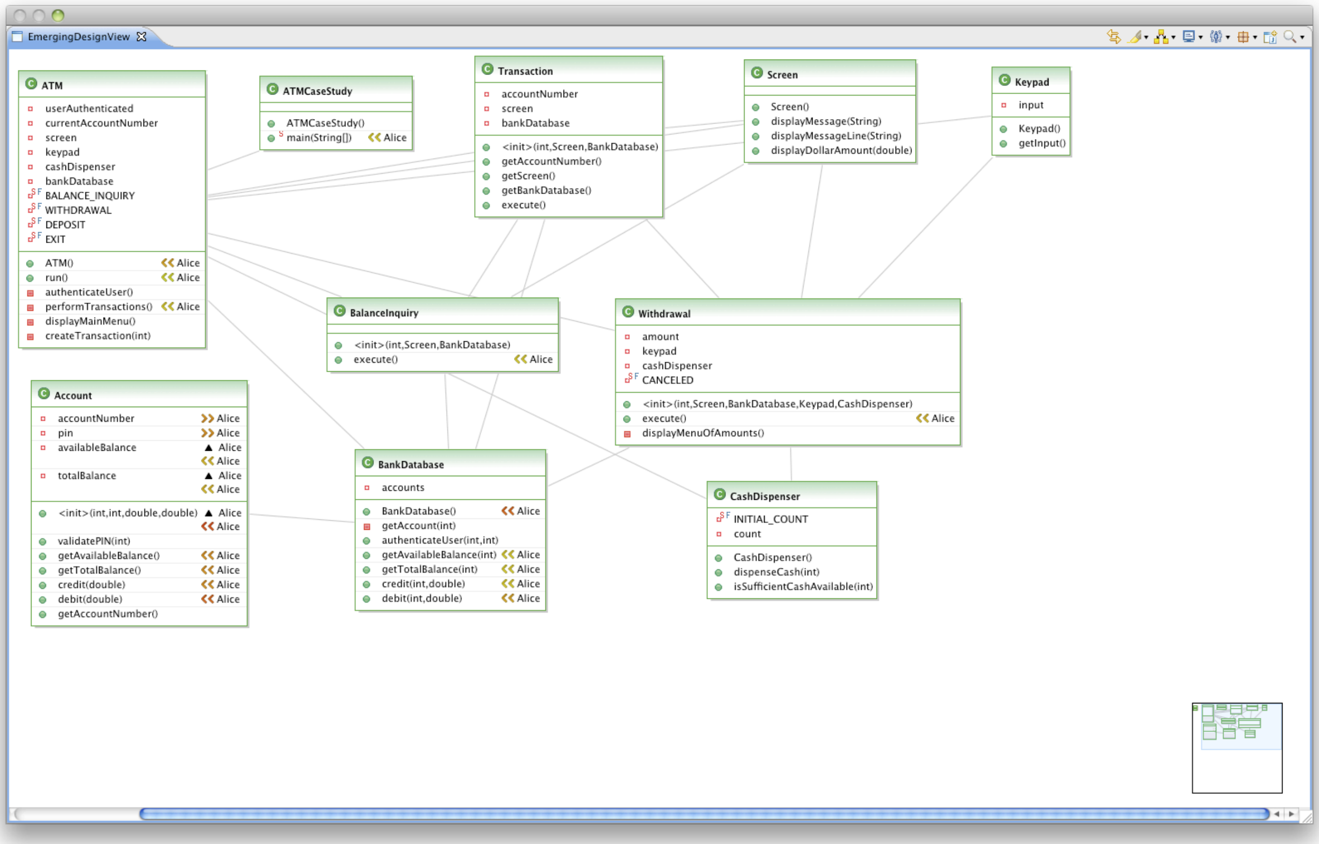


**Figure 4. Alice's Sphere of Influence after she starts making changes.**

the *Account* class to be represented by a dollar amount with decimals.

Both Alice and Bob check out the same version of the source code and make their changes in parallel. When they finish their changes, they compile and test the version in their workspace, which works correctly. Because the changes made by Alice affect different files than the changes made by Bob, the SCM system will allow both of them to check in their changes without any warning. However, there is an indirect conflict between Alice's changes in the *Account* class and Bob's changes in the *Deposit* class. This indirect conflict introduces an error in the version in the repository. The *Deposit* transaction implemented by Bob will credit the dollar amount in the corresponding account, but it will be interpreted as cents by the rest of the application. As a result, all the deposit transactions will be accounted for 100 times less than the actual amount of money deposited.

If Alice and Bob use CASI while making their changes, Alice will see the visualization in Figure 4 when she starts coding. At the beginning, she changes the type of *Account.availableBalance* and *Account.totalBalance* from *double* to *long*. By company policies, she is not allowed to change the signature of methods. So, she also inserts a cast in the constructor for *Account* to convert the *double* values received to *long*. At this point, CASI shows her "how far" the influence of her changes reaches.

Alice's next step is to review the rest of the source code to make sure that all transactions consider the balance amount as cents instead of dollars. CASI's visualization can help her with this task because all the source code entities involved in a dependency with her changes belong to her Sphere of Influence. So, she reviews *BalanceInquiry.execute()* and *Withdrawal.execute()*, and decides to adapt them to the new way of measuring currency. As she starts doing this, Bob also starts implementing his changes in parallel. CASI then shows both Alice and Bob the visualization in Figure 5. As a result of Alice's additional changes, her Sphere of Influence has grown, and it covers a considerable percentage of the source code entities of the application. This should serve as a warning that the risk of indirect conflict will be high if other developers start making changes.

Given that a large portion of the source code is influenced by Alice, Bob might decide to hold off his changes until Alice finishes hers. However, he decides to continue coding, hoping that his Sphere of Influence will not overlap with Alice's before one of them finishes making changes. He starts by writing the *DepositSlot* class, which is not yet used by any of the other source code entities. As a result, Bob's Sphere of Influence is also part of CASI's visualization, as can be seen in Figure 5.

Alice and Bob continue writing code. When Bob finishes implementing the *Deposit* class, CASI shows both developers the visualization in Figure 6. In this figure, all source code entities are covered by either Alice's or Bob's Sphere of Influence. In fact, in many of them, both Spheres of Influence overlap. This visualization should encourage Alice and Bob to communicate with each other in order to ensure that their changes are

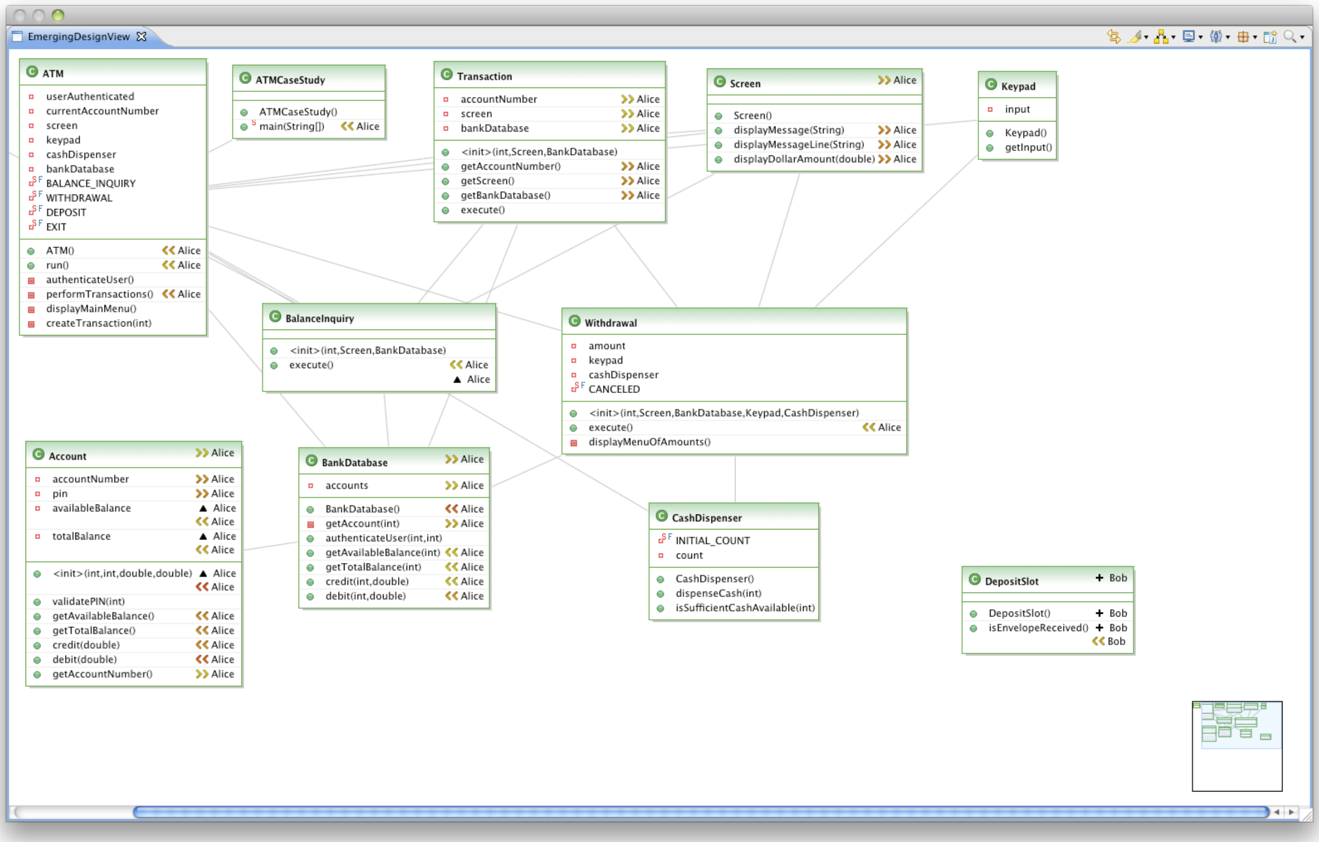


**Figure 5. Alice's Sphere of Influence grows and Bob starts making his changes.**

compatible. As a consequence of this communication, they would be able to avoid the indirect conflict before checking in their changes.

Normally, indeed, Bob and Alice should start talking before the point reached in Figure 6. As they see that their Spheres of Influence start to overlap more and more, it is a clear sign that their changes might be incompatible. A simple check over IM or per phone, or even in person, should alert them that their changes are not compatible. In response, Bob can simply change the assumption of dollar figures in his code, and the resulting work will no longer lead to a problem. Note that, after Bob makes this modification, the Spheres of Influence will still overlap, indicating the close relationship between their changes.

## 4. DISCUSSION AND FUTURE WORK

Our example in Section 3 demonstrates the utility of CASI on a small software system. We anticipate that a major theme of our future research will be addressing the scalability of the approach and visualization.

For a large source code base, we expect the Spheres of Influence to potentially grow large. Because the size of the Sphere of Influence for a developer is, in part, determined by the number of dependencies involved in a change, a small change can still produce a large Sphere of Influence if the changed code is involved in many dependencies. If the Spheres of Influence grow too quickly, they might overwhelm developers and cause them to stop paying attention to the visualization.

We believe that filtering mechanisms could help mitigate the issue of scalability. Some filters that we anticipate being useful are: (1) limit the maximum distance between the changed and influenced source code entity, (2) limit the maximum or minimum severity displayed, (3) show only a developer's changes and other developer's Spheres of Influence. In general, it might be useful to allow developers to choose what they want to display for each developer: changes, forward influence, backward influence, or a combination of them.

We could apply filtering mechanisms to also help developers identify the location of overlaps. We could show only overlaps, or only some kinds of overlaps. Also, we might help developers find the most dangerous overlaps through modifications of the visualization or querying mechanisms integrated into the user interface. In addition, we intend to experiment with new visualization modes in which the graph layout can be arranged such that the Spheres of Influence or their overlaps are more apparent.

We are also considering entirely new and complimentary visualizations of Spheres of Influence. Such visualizations may abandon the UML class diagram — presenting the program and the Spheres of Influence in a more scalable fashion. While such visualizations are a current topic of discussion in our research group, we imagine that they may present information in a way that

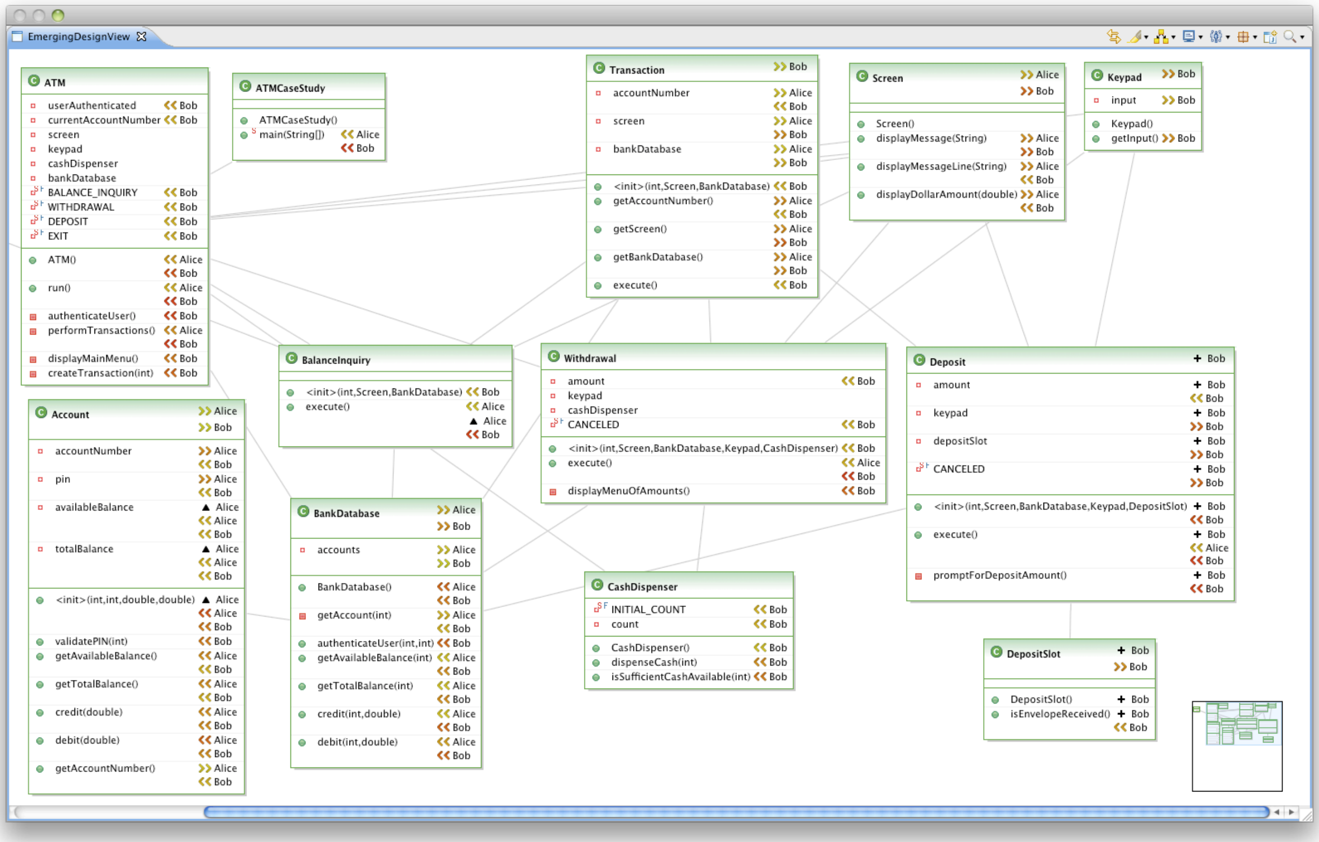


**Figure 6. Alice's and Bob's Spheres of Influence overlap.**

addressed other development tasks, such as providing overviews of the current state of a project for project managers.

It would also be interesting to apply CASI in real world projects and capture the history of changes and influence to try to learn more about how indirect conflicts are produced. Additionally, we plan to explore the use of CASI for test case selection. If we consider test cases as part of the source code of the application, the Spheres of Influence might indicate which test cases need to be re-run after developers make changes. The selected test cases would be those that were covered by the Spheres of Influence.

## 5. CONCLUSIONS

In this paper, we proposed a new kind of approach to the problem of indirect conflicts. The traditional approaches rely on code analysis and therefore cannot predict semantic indirect conflicts, especially those that end up in run-time problems. Our approach is based on a certain kind of awareness information that we call Spheres of Influence. The Spheres of Influence show developers which source code entities are influenced by their changes. This information is dynamically broadcasted to developers as it emerges while they make changes.

We implemented the visualization of the Spheres of Influence in our tool CASI. This tool is different from traditional approaches in its proactive nature. It is designed to warn developers of the risk of indirect conflicts at an early moment as opposed to the approach of other analysis tools, which analyze the changes to detect indirect conflicts after they have been introduced.

We tested CASI in some small examples. However, we still need to test it in more projects of different sizes and in real situations with real developers. In our examples, CASI was generally effective at showing whether the changed source code entities are involved in many dependencies. When the Spheres of Influence covered a large amount of source code entities, this was a warning that the chances of overlap had increased. With this, the risk of indirect conflict also increased.

However, there are still possibilities for simplifying CASI's visualization when the Spheres of Influence overlap. Developers would be subject to a tedious process of investigating the overlap of influenced source code entities if they tried to figure out which overlaps pose the highest risk. We discussed some improvements that might ameliorate this problem.

In the future, we plan to improve CASI for scaling to large systems and to simplify its visualization, maybe even by designing an entirely new one.

## 6. ACKNOWLEDGMENTS

This work was funded in part by Fundación Caja Madrid.

## 7. REFERENCES

[1] Apache Subversion. http://subversion.apache.org/

[2] I. Almeida Da Silva, P.H. Chen, C. van der Westhuizen, R.M. Ripley, and A. van der Hoek, "Lighthouse: coordination through emerging design," Proceedings of the

2006 OOPSLA workshop on eclipse technology eXchange, 2006, p. 15.

[3] S. Bajracharya, J. Ossher, and C. Lopes, “Sourcerer: An internet-scale software repository,” Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009, pp. 1–4.

[4] Li-Te Cheng, S. Ross, and J. Patterson, “Jazzing up Eclipse with collaborative tools,” Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange, 2003, pp. 45–49.

[5] CVS - Open Source Version Control. http://www.nongnu.org/cvs/

[6] P. Deitel and H. Deitel, Java™ how to program, 2006.

[7] Eclipse. http://www.eclipse.org

[8] R. Hegde and P. Dewan, “Connecting programming environments to support ad-hoc collaboration,” 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. ASE 2008, 2008, pp. 178–187.

[9] Java Primitive Data Types. http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html

[10] B. Magnusson and U. Asklund, “Fine grained version control of configurations in COOP/Orm,” Lecture Notes in Computer Science, 1996, pp. 31–48.

[11] A. Sarma, G. Bortis, and A. van der Hoek, “Towards supporting awareness of indirect conflicts across software configuration management workspaces,” Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007, pp. 94–103.

[12] T. Schümmer and J.M. Haake, “Supporting distributed software development by modes of collaboration,” Proceedings of the seventh conference on European Conference on Computer Supported Cooperative Work, 2001, p. 98.

[13] D. Shao, S. Khurshid, and D.E. Perry, “Evaluation of semantic interference detection in parallel changes: an exploratory experiment,” Compare, vol. 4, p. 5.

[14] M.A. Storey, D. Čubranić, and D.M. German, “On the use of visualization to support awareness of human activities in software development: a survey and a framework,” Proceedings of the 2005 ACM symposium on Software visualization, 2005, pp. 193–202.

[15] C. van der Westhuizen, P.H. Chen, and A. van der Hoek, “Emerging design: new roles and uses for abstraction,” Proceedings of the 2006 international workshop on Role of abstraction in software engineering, 2006, p. 28.

[16] M. Weiser, “Program slicing,” Proceedings of the 5th international conference on Software engineering, 1981, pp. 439–449.