Towards IDE Support for Abstract Thinking

Oren Mishali
Department of Computer Science
Technion, Haifa Israel
32000
omishali @ cs.technion.ac.il

Yael Dubinsky
IBM Haifa Research Lab
Haifa, Israel
31905
dubinsky @ il.ibm.com

Itay Maman
Department of Computer Science
Technion, Haifa Israel
32000
imaman @ cs.technion.ac.il

ABSTRACT
Abstract thinking is considered to be a high level cognitive skill that enables a comprehensive understanding of a specific concept or a problem using different levels of detailing. Based on a lab activity we conducted on the matter of abstraction, we present guidelines for enabling an Integrated Development Environment (IDE) to promote abstract thinking. The guidelines are defined in the context of an Aspect-Oriented Process Support (AOPS) framework that aims at customizing IDEs to automatically support various software development practices. Specifically, we suggest two kinds of guidelines. The first is concerned with a positive feedback from the IDE in cases where abstraction is used. The second kind is concerned with cases in which the developer is encouraged to move to a different level of detailing, that is, promoted to use abstract thinking.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments – *Integrated environments;* K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software development, Software process;* K.3.1 [**Computers and Education**]: Computer Uses in Education – *Computer-assisted instruction (CAI)*

General Terms
Management, Performance, Design, Human Factors.

Keywords
Abstract thinking, process support, aspects

1. INTRODUCTION
Abstract thinking is defined as the ability to generalize, or focus on the significant details while ignoring the less significant ones [9]. It is considered to be a key skill for software engineers as reported in some preliminary educational initiatives [1,3]. In this paper, we suggest using concrete IDE support for abstract thinking and present guidelines for the definition of rules that can be integrated into the IDE. The goal is to provide the developer with notifications and alerts during the course of development.

The definition and deployment of the rules is facilitated by the Aspect-Oriented Process Support (AOPS) framework [6], which aims at providing automatic guidance for following the best practices of software development. Test-driven development (TDD) is one of the practices already supported by AOPS over the Eclipse platform. User evaluation experiments showed that accompanying the development with notifications and alerts in general, and TDD support in particular, is indeed effective [5].

Defining support for abstract thinking is challenging. Unlike TDD, abstract thinking is an informal practice, which cannot be easily captured by a set of well-defined rules. In order to understand the nature of the support needed, we conducted a three-part lab activity. In the first part, thirty-three participants studied the notion of abstraction, namely abstract thinking. Then they worked on real-life development tasks, where their development steps as well as their visible thinking processes were logged by observers. Finally, a reflection on the activity [7,8] was collected.

Analyzing the data from the logs and reflection, we conclude that concrete IDE support for abstract thinking is practical and suggest two kinds of such support. The first is concerned with a positive feedback from the IDE in cases where abstraction is used, and the second with cases where the developer is encouraged to use abstract thinking. We believe that such support may increase the awareness of abstraction and encourage further utilization.

In the next section we give an overview of the AOPS framework and in Section 3 we describe the lab activity and its findings. Finally, in Section 4, we provide a discussion as well as conclusion.

2. AOPS FRAMEWORK
The AOPS framework facilitates the definition and deployment of support for a large variety of software processes in the form of rules. The framework uses aspect-oriented technology [2] to expose significant development events (e.g., creation of a Java class, a refactoring operation) from the underlying development environment. These events are called *key-events* and serve as a basis for the definition of the rules. Each AOPS rule consists of a condition part and an action part. The condition part specifies when the rule is activated and contains a set of key-events and a Boolean condition. During the development, a rule is activated when one of its key-events occurs and the Boolean condition holds. The action part of the rule specifies what happens when a rule is activated and usually provides the developer with a message of a certain type (e.g., Error, Warning).

Internally, the framework translates the set of defined rules into automatically generated code in the form of classes and aspects. This code is then automatically integrated into the target development environment (currently, Eclipse) thereby modifying the environment to support the practices described by these rules.

ROA'08, May 11, 2008, Leipzig, Germany.

In Figure 1, we see an example of an AOPS message given to the developer due to the activation of a TDD rule. The key event occurs when the developer attempts to code while not having a failing test; this is a violation of the TDD practice. As seen in the figure, the feedback is both textual and visual. The feedback may also be presented within Eclipse wizards. In Figure 2, for example, the class creation wizard prohibits the developer from creating a new class when a corresponding failing test does not exist.

We use a metaphor to guide the definition of AOPS rules. A software development process is viewed as a trail shaped by recommended methods and practices, and the developers are hikers who are supposed to follow the trail. Consequently, an AOPS rule can be either an on-track rule or a deviation rule. An on-track rule is triggered when the trail is followed and it provides the developer with positive feedback. Similarly, if the developer deviates from the trail, a deviation rule is triggered and a negative feedback is provided. As presented in the next Section, we analyzed the activity logs in light of this metaphor. More precisely, we searched for cases where abstract thinking was used (on-track), and for cases where abstract thinking was needed but not exercised (deviation).

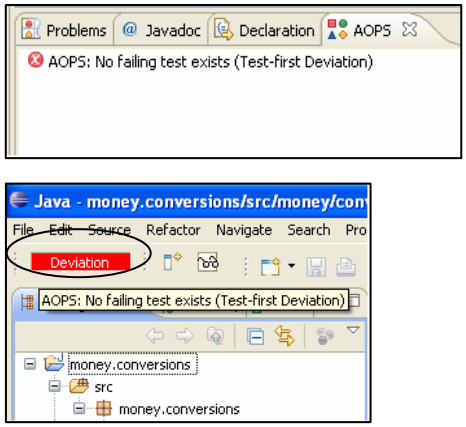

Figure 1. AOPS message presented in the AOPS View and the AOPS Bar

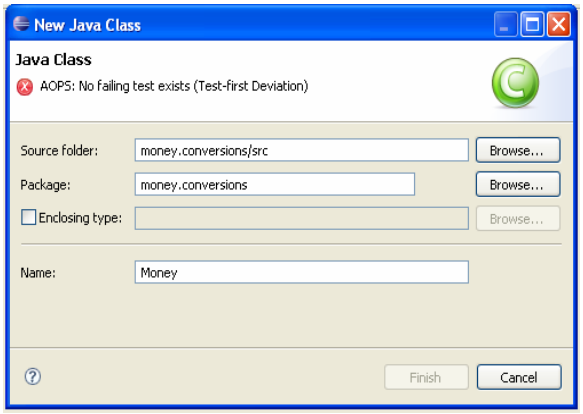

Figure 2. AOPS message presented within Java class creation wizard

3. LAB ACTIVITY

We conducted a lab activity on the matter of abstraction in software development. The participants were thirty three fourth-year students who were computer science majors in an advanced Software Engineering project course. The activity had three parts:

- Lecture – in which participants learned and discussed the notion of abstraction and how it is related to software engineering.
- Development activity – in which participants heard about the pair programming practice and were distributed into groups of two or three developers in order to perform a development task in solo or pairs, respectively, where the additional person in each group served as an observer. This part was performed in the lab where the course was taught.
- Reflection – in which participants reflected on the complete activity using a web-based feedback mechanism familiar to the students. The feedback was given during the week after the activity.

As part of the work in the course, about two months before the activity we asked the participants to fill in a questionnaire about their level of familiarity and experience with programming concepts and tools. Twenty seven participants answered. The results showed that participants felt knowledgeable with Java programming and object-oriented design, less knowledgeable with Eclipse IDE and unit testing, and as novices with respect to JUnit and TDD. Regarding the development process, participants felt they were less experienced with measuring the development process and product, but felt knowledgeable and even expert with working in pairs.

In what follows we elaborate on the development activity itself and the findings.

3.1 Development Activity

Participants were distributed into thirteen groups: seven groups of three participants (a pair of developers and an observer) and six groups of two participants (a developer and an observer). All developers were asked to pick a development task from their on-going project and to develop it during this phase of the activity. Solo developers were asked to think aloud – i.e., talk while working – about whatever they were doing. The observers were asked to be non-participatory observers, i.e., not to talk with the solo/pair or interfere in their work. Their task was to document the development activity. Table 1 presents the kind of pages observers were asked to fill during the activity.

Table 1. Observer page

Time stamp	Conversation	Activity	Comment

Once the lab activity started, all developers began working on their tasks and observers began to record their observations. This lasted for exactly one hour, after which we stopped the participants asking them to upload the code and test files that were changed to the course's web site. The observers' pages were collected and an open discussion was guided in order to receive an initial feedback.

3.2 Findings

Overall we collected thirteen activity logs each spanning two to five observation pages; Observers of solo developers filled a total of fourteen pages (2,2,2,2,3,3 – average of 2.3) where observers of pairs filled a total of twenty-five pages (2,2,2,4,5,5,5 – average of 3.6).

We analyzed the logs and found cases of *abstraction usage* (i.e., cases in which abstract thinking is used) and cases of *abstraction necessity* (i.e., cases in which abstract thinking can help).

3.2.1 Abstraction Usage

We found four categories of abstraction usage that are characterized by the use of generalization or by the process of ignoring details in order to advance the development work. In what follows we describe each category and illustrate it using the data from the activity log.

Searching. To perform a search operation, one needs to select keywords that adequately represent the problem domain. This selection of searching keywords is an abstraction of type removing details. Here, in the process of solving a problem, participants selected keywords from the problem domain and searched for these in Google or in the API documentation in order to find a solution. Tables 2 and 3 present such a case as reported by observers of a solo and a pair, respectively. When no data appears in the comment column it is not presented.

Table 2. Searching in solo programming

Time stamp	Conversation	Activity
11:04	Open meta-inf in order to add extension point	Looks in Google, studies how to add the required action
11:05	Use popupMenu as extension -> view contribution – add a unique ..	Continues to move between Eclipse and Google in order to build the extension.

Table 3. Searching in pair programming

Time stamp	Conversation	Activity
11:12	Discussion about how to implement the server	
11:16		Adding Jars to the client project
11:20		Searching Google for permission in RMI
11:24		First trial to solve the permissions problem
11:27		Second search in Google for a solution
11:28	One suggests solving the problem using a policy file. Also suggests refining the search according to the Eclipse error.	

We can see that in both cases there is a move between coding and searching activities. In the pair observation there is also a conversation on which keywords to use and about refining these keywords.

Naming. When choosing a proper name for an underlying coding element (e.g., classes, methods), one captures the essence of the element in a short yet meaningful term. The process of choosing a name involves abstract thinking of type removing details. In Tables 4 and 5 we show naming-related excerpts from two different pairs: naming of a class and renaming class members.

Table 4. Naming a class

Time stamp	Conversation	Activity
11:00	How to name the class?	Creating LoginDialog Class

Table 5. Renaming class members

Time stamp	Conversation	Activity
11:35	Performing refactoring to member names that are more correct in IOAutoX	Update of gui_integration_objects. IOAutoX

Testing. We claim that writing a test for a specific unit requires a different level of abstraction than writing the code of the unit itself. Instead of focusing on the implementation details of the unit, writing the test triggers thinking about how to check the unit functionality. Table 6 presents such a case.

Table 6. Testing a class

Time stamp	Conversation	Activity
11:14	A slight argument on the right location of classes under the different packages. Talk about testing the AutoXlog class using JUnit.	Move AutoXlog to Model_Experiments package. Adding a test for AutoXlog named AutoXlogTest.
11:20	Making a decision on the correct protocol between the GUI and the model.	Add GenerateAutoXStatistics Event in the Event Enum.

In this case the test code was written only after the application class was written. Once the testing class was finished, the developers moved back to the application code to develop a different functionality.

In JUnit, the setUp() method holds initialization code common to all test methods. Therefore, the time when setUp() is introduced (see Table 7) is an example of abstraction of type generalization since common initialization code for all existing test methods should be identified.

Table 7. Implementing setUp( )

Time stamp	Conversation	Activity
11:31	Consulting with each other about the way of testing	Implement setUp(), add a new test, write test function named testGetExperimentID()

Using diagrams and views. The phrase "a picture is worth a thousand words" hints at why moving from writing lines of code

to looking at a diagram one can actually change one's level of abstraction and gain meaningful insight while doing so. The diagram can be a UML-like representation of a class (Table 8) or an Eclipse view (Table 9) depicting the packages in a program.

Table 8. Using a diagram

Time stamp	Conversation	Activity
10:54	A conversation among the pair developers with respect to the requirements. What one object needs from the model and vice-versa. The conversation involved sketching and designing an additional class to represent the AutoX object	
11:01	Talking about code that is automatically generated as a result of an inheritance … Talking about performing refactoring to part of the code as a result of moving from iteration 1. Using a previous diagram in order to define the class members.	Adding classes to the code …

Table 9. Using an Eclipse view

Time stamp	Conversation	Activity
11:03	One refers to some packages with errors and discusses with the other how to handle them, i.e., define new tasks.	

Table 8 (10:54) denotes a conversation that led to the drawing of a class diagram and to a subsequent design discussion. This diagram was then used (Table 8, 11:01) by the developers when they coded in the members of that class. In Table 9 the developers readily evaluate their project's status, from Eclipse's Package-Explorer view, in terms of amount of pending errors. This evaluation leads to the definition of a yet another development task.

3.2.2 Abstraction Necessity

We identified two categories of abstraction necessity in the activity logs. A more elaborate discussion of AOPS support for abstraction necessity is presented in Section 4.

Searching. We identified events in which the developers searched for types in their code in order to perform their task (see Table 10).

Table 10. Searching for types

Time stamp	Conversation	Activity	Comment
11:02	A conversation on where to add the solution in the code	Erasing existing code and a discussion on the solution implementation	Search for the solution location
11:06	How to implement the solution and using which existing classes in the project or in Java	Begin to write the new code and search for a specific method	Begin implement ing and search classes that can help

Searching for a type can be performed using several methods. It can be done by navigating through the project's file tree or by using higher-level search facilities such as the Search-Type-Dialog. We therefore suggest that a *lengthy* browsing through the project's file-tree should serve as an AOPS key-event that will open a dialog recommending higher-level searching aids.

Testing. A testing-related necessity is characterized by the creation of several test methods in a test class. We suggest that this key-event trigger a recommendation dialog suggesting the creation of setUp() and tearDown() methods to enable a common initialization and finalization of the test fixture, thus encouraging abstract thinking of type generalization.

Another necessity stems from the fact that we have activity logs where not even a single testing-related activity was recorded. This obviously implies that the developers are not looking at their program from a test-centric standpoint. Such situations can be identified by a timer-based AOPS key-event that is associated with a testing-oriented recommendation dialog.

3.3 Reflection

In this phase, participants were asked to reflect on the activity. Specifically, they were asked what they had learned, how they planned to use abstraction, and to give an example of a pair conversation that includes abstraction. Twenty six such reflections were examined.

The following are some answers by participants with respect to how they planned to use abstraction.

- "When I explain the project goal to my friends I use a level of abstraction that can give them a concept without delving into the details. At work, when we plan a project we go over all abstraction levels in the UML and do them all."
- "I will use abstraction in the following case: there is a need to implement classes for different kinds of messages that are transferred in the network. All messages need the functionality of converting the message from stream class to bytes and vice versa. In this case I will create an abstract class that includes the abovementioned functionality and common definitions where every message will inherit it and implement the functionality according to its needs."
- "In the design phase of iteration 2 we used abstraction of the code for a static diagram (classes) of the software in order to be able to talk about what will be in the code from a high level perspective."
- "Working in a large team, in order to reach the biggest common denominator I will use abstraction to explain the project essence and its tasks."

Most participants answered this question by referring to high-level design activities. In the last example, there is a use of abstraction for simplifying communication. In reply to this question, only four out of twenty six participants mentioned IDE-level activities, such as the ones mentioned in the second example above.

Eighteen participants out of twenty six answered the question to describe a pair protocol that includes the use of abstraction. Among them fourteen indicated coding activities. We present two specific answers in which participants used a metaphor as the abstraction usage in the conversation. Metaphor [4] can be viewed as an abstraction kind since it uses one set of terms in order to better explain concepts expressed in another set of terms. The following are parts of the answers of participants.

- Developer A: We need to develop software to implement the communication between two stations.
  Developer B: Let's think as if we have here two people that talk one with each other. How the conversation begins?
  Developer A: One says hello and the other answers.
- Person A: I would like to implement a communication protocol between server and clients.
  Person B: What do you mean?
  Person A: I would like to develop a program that implements a connection that is similar to a father who distributes roles among his sons; he can give a role only to one son simultaneously and cannot answer questions of more than one son simultaneously.

4. DISCUSSION AND CONCLUSION

In this section we reflect on our findings and describe how these findings can be used to define concrete AOPS rules for abstract thinking.

We believe that abstract thinking is an important cognitive tool that is required in the field of software engineering. In order to use this tool, a developer must be aware of its existence and of its applicability to the task at hand. Examining the issue of awareness we see that the majority of participants in our activity associated abstract thinking with activities that are external to the use of the IDE (see Section 3.3). The answers – in the reflection part of the activity – primarily mentioned design activities and sentences regarding the nature of the project or the initial definition of large modules thereof. Despite the fact that our activity logs show many events where abstract thinking was used during coding-time, the participants largely ignored events of this kind in the reflection.

This suggests that our participants' awareness regarding the role of abstract thinking during actual coding activities is low. Therefore, in what follows we suggest AOPS rules that will enable the IDE to improve the developer's awareness to coding-time abstract thinking.

- **AOPS on-track rules.** Such rules identify certain events where the developer moves between different levels of abstraction and provide a positive feedback that may encourage further usage of abstract thinking. Moreover, bringing into awareness the use of abstraction may lead to a more professional usage of this cognitive tool.

  As an example, an AOPS rule can detect cases where after a long editing session the developer consults a related design diagram and then go back to coding. We see this as a good use of abstract thinking since the developer decided to move from a detailed view of the program to a less detailed one and vice versa.

- **AOPS deviation rules.** Such rules relate to abstraction necessity. These rules can monitor the developer's activity and can provide a recommendation whenever a different level of detailing may be helpful.

  For example, we can define a rule that detects cases where several tests have failed together. As a result, the AOPS may pop up a dialog box that shows the following recommendation: "Try to find what these tests have in common". Further assistance can be provided by showing the intersection of the code coverage of these failing tests. Naturally, the developer does not have to follow this recommendation. However, if he/she chooses to accept the recommendation he/she will be able to think about the task from a different point of view (with a different level of detailing).

We conclude with one subtle point that is related to the interplay of pair programming and abstract thinking. Observers of pairs recorded significantly more observations than observers of a solo developer. This is clearly indicated by the average number of pages filled by the observers (2.3 vs. 3.6). We attribute this to the fact that the pair-programming practice promotes verbal communication, which in turn promotes abstract thinking. The developers had to *explain* their intentions instead of simply *writing them in code*. This process of explanation requires abstraction, as well as self reflection, and is expected to enhance the pair's quality of work. Implementing support for verbal communication within the IDE is left as food for thought.

Acknowledgements. We would like to thank Orit Hazzan who contributed to our activity by teaching and discussing the notion of abstraction with the students and by suggesting reflection questions.

5. REFERENCES

[1] Hazzan, O., Kramer, J. 2007 Abstraction in Computer Science & Software Engineering: A pedagogical perspective. Featured Frontier Columnist, System Design Frontier - Exclusive Frontier Coverage on System Designs, 4(1) 6-14.

[2] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. 1997 Aspect Oriented Programming, in European Conference on Object-Oriented Programming, Springer-Verlag, pp. 220-242 .

[3] Kramer, J. 2007 Is Abstraction the Key to Computing? Communication of the ACM 50(4) 37-42.

[4] Lakoff G, Johnson M. 1980 Metaphors We Live By. The University of Chicago Press.

[5] Mishali, O., Dubinsky, Y. and Katz, S. 2008 (submitted) The TDD-Guide Training and Guidance Tool for Test-Driven Development, The International Conference on Agile Processes and eXtreme Programming in Software Engineering (XP), Limerick, Ireland, June 10-14, 2008.

[6] Mishali, O., and Katz, S. 2006 Using aspects to support the software process: XP over Eclipse, in International Conference on Aspect-Oriented Software Development, ACM, Bonn, Germany, pp. 169-179.

[7] Schön, D. A. 1983 The Reflective Practitioner, BasicBooks.

[8] Schön, D. A. 1987 Educating the Reflective Practitioner: Towards a New Design for Teaching and Learning in The Profession. Jossey-Bass, San Francisco.

[9] The Free Dictionary 2008 Farlex Inc.,. http://www.-thefreedictionary.com.