VisUML: A Live UML Visualization to Help Developers in Their Programming Task

Mickaël Duruisseau[1,2](✉), Jean-Claude Tarby[2], Xavier Le Pallec[2], and Sébastien Gérard[1]

[1] CEA LIST, Boîte 94, 91191 Gif sur Yvette, France
mickael.duruisseau@gmail.com, sebastien.gerard@cea.fr
[2] Univ. Lille, UMR 9189 - CRIStAL, 59000 Lille, France
{jean-claude.tarby,xavier.le-pallec}@univ-lille1.fr

Abstract. Developers produce a lot of code and most of them have to merge it to what already exists. The required time to perform this programming task is thus dependent on the access speed to information about existing code. Classic IDEs allow displaying textual representation of information through features like navigation, word searching or code completion. This kind of representation is not effective to represent links between code fragments. Current graphical code representation modules in IDE are suited to apprehend the system from a global point of view. However, the cognitive integration cost of those diagrams is disproportionate related to the elementary coding task.

Our approach considers graphical representation but only with code elements that are parts of the developer's mental model during his programming task. The corresponding cognitive integration of our graphical representation is then less costly. We use UML for this representation because it is a widespread and well-known formalism. We want to show that dynamic diagrams, whose content is modified and adapted in real-time by monitoring developer's actions can be of great benefit as their contents are perfectly suited to the developer current task. With our live diagrams, we provide to developers an efficient way to navigate through textual and graphical representation.

Keywords: Human-Computer Interaction
Model Driven Engineering · Software engineering
Unified modeling language · Human-centered design

1 Introduction

Human-Computer Interaction (HCI) has significantly evolved in recent years with the appearance of mobile and tactile devices, voice and gesture recognition, augmented and virtual reality, etc. Nowadays, most of the smartphone users know how to interact with a map, using simple interactions like touch,

© Springer International Publishing AG, part of Springer Nature 2018
S. Yamamoto and H. Mori (Eds.): HIMI 2018, LNCS 10904, pp. 3–22, 2018.
https://doi.org/10.1007/978-3-319-92043-6_1

but also some more complex, like swipe or pinch. In the meantime, software practitioners still develop applications only with a keyboard and a mouse. Furthermore, "development tools are showing mainly text with (so much) obstinacy" [1] despite some improvements concerning HCI in their IDE, like syntax coloration and auto-completion. We may consider software visualization tools as an improvement of the HCI, but their place in IDE and their use remain anecdotal. Visualization tools generally help developers to understand the global architecture of the application they are working on or the impact of what they are changing. Development consists mainly in producing code but not dealing with considerations of macroscopic nature. We argue these visualization tools are not focused on the most important and elementary task: programming. We claim that a graphical representation of elements that are currently knitted by a developer may be more easily accepted. The first reason is it can quickly provide information that is less visually explicit in textual code and still relevant for coding. More specifically it may highlight the different relations between elements (structural relations or specific execution flow). The second reason is that graphical representations are more suited to mobile and tactile devices (like tablets) than textual code and so, by taking advantage of them, they can provide HCI improvements of IDE.

In this paper, we present VisUML[1,2] a tool which uses a "*live diagramming*" approach and implements this point of view of software visualization. We defined *live diagrams* as being diagrams (UML or not) that display information according to the current task of the developer. We assume that currently opened elements in an IDE refer to this task and are therefore part of developers' mental models. These diagrams are updated instantly each time the code changes. Our approach consists in reducing the number of displayed elements but also to ease the navigation between code and diagrams.

To present this approach, we first describe the scientific background on which VisUML is based, as well as our design guidelines. Second, we review works that are related to our idea. Then we explain how VisUML works, with a focus on user interactions. Finally we highlight the contributions of this tool and we discuss its features and its evolutions. At last, we conclude with a summary and perspectives.

2 Scientific Background and Design Guidelines

The psychological mechanisms related to programming received much attention during the 90s [2–4]. Notable among these was the fact that developers work in little "spurts" [5] (sprint). Green [5] mentions the notion of spurts to emphasize that programming is a series of small steps where each one refers to a mental chunk or scheme. Therefore it is logical that the developer's main concern consists in connecting the spurt result with what has been produced so far. Indeed,

[1] VisUML website: http://these.mickaelduruisseau.fr/VisUML/.
[2] VisUML demonstration video: https://youtu.be/buyGojmbUpQ.

developers often read and analyze what has been done in order to properly "knit" (link) what they do with the rest of code.

The development environment should therefore optimize the "reading/production" cycle. It must be adapted to the current spurt and simultaneously provides quick access to information that will help developers to link their code to the existing one [6].

Thus, it is no wonder that most of current code editors propose shortcuts to go quickly to the definition of the selected element or to list all the invocations of the selected method. However, navigation is not the only way to find relevant information. Changing visual properties of code elements is another way to highlight what can interest the developers in their knitting task; for instance, the indentation clearly shows the different control structures in which the current line of code is nested, background color variation is sometimes used to identify the different places where a variable is used, etc. Visual changes can go further with concrete transformations of shapes and concerned elements. That is the speciality of what the software community called visualization tools. These ones are of great help in order to understand the existing code; this is particularly mentioned in the SoftVis/VISSOFT conferences cycle. [1] insists on the necessity of the requisite ubiquity of visualization in development environments, even if it means rethinking them entirely. A majority of the work on software visualization results in tools that allow finding or obtaining macroscopic information.

At the opposite, our approach provides microscopic information by focusing on more specific and activity related data. We do not aim to provide information about possible impacts of each modification, nor knowing which application's parts have to be rewritten. Instead, our goal is to allow developers to have a visual support of their code. This will enable them to quickly find information about entities and their relations and will also add an easy navigation mean. In addition, this visual support could be shared with peoples that have different needs in terms of visualization. For example, a product manager will not use the same kind of representations than a developer, but the displayed entities remain identical. In the same idea, developers can have a fully detailed class diagram, whereas project leader may want to see a class diagram without attributes or operations, with a color code that indicates the code quality of each class or their modification date, number of commits, etc.

Developers constantly execute code reading operations to find information in order to allow them to modify their code. We wish to shorten these reading operations by giving a quick access to elements often used or viewed by the developers. These elements are mostly represented as entities connected by links. The textual support is not very effective to represent a system with interconnected elements; however the diagrammatic representations are much more efficient in this task [7]. Furthermore, works on the psychology of senior developers show that they have mainly problems understanding the control flow rather than the basic bricks of a language (e.g. variables names) [8]. Thus, in addition to the two aspects to be displayed (entities and links), we can add the control flow between entities that are associated to the active coding task. To ensure that the access

of information through the graphical representation is as fast as possible, the reading/decoding of this representation must take as little time as possible. The cognitive fit is a main concern for our tool, and the time spent when switching between a representation to another, or when switching between tasks in general (e.g. code review vs. diagram creation, class dependencies search vs. debugging), is thus very important. This concern is the heart of the cognitive dimensions [9] and can be found as a rule *cognitive integration* in the physics of notations [10]. In our case, when switching from a textual code editor to a graphical representation, it is clearly necessary that developers keep their references. For example it is important that developers recognize the entities they have just been manipulating. Therefore the graphical representation has to be close to their mental model and display information about:

- Entities linked to the active coding task: whether they are open or not in the IDE and which element is currently active
- Neighbourhood data: accessible via variables of a method, attributes of an object, inheritance...
- Control flow: the content of a currently consulted method, or at the origin of a search or navigation.

We use the UML language for the graphical representation because it remains a language known and mastered by developers, even if according to different surveys it is not enough used in firms. This selection was made according to the principle of cognitive integration [9]: adapt to the knowledge of developers.

Forward and Dzidek [11,12] attest that two of the three most widely used UML diagrams are the class and the sequence ones. We consequently chose them in VisUML. The **class diagram** is important because developers can recognize the entities they manipulate, as well as consult other related entities thanks to the different types of links.

In order to display the control flow, especially for the body of a method, we opted for the **sequence diagram**. We assume that this diagram is a complementary visual support for developers that are working on the implementation of a method. They will be able to see all the classes that are involved in this method, and especially the different exchanges (and thus links) between them. In addition, the temporality of these exchanges is emphasized because it is represented on the *y-axis*. This correspondence between *y-axis* and *temporality* reduces subjectivity in the layout, and is therefore less subject to interpretation than communication diagram, where the placement at x and y is arbitrary. Activity diagrams and states machines can be used for the implementation of a method but their point of view (activity or state) adds a semantic gap which is likely to increase the time required for decoding (without taking into account the ordering of instructions).

The navigation between the code and these two types of representation (class and sequence diagrams) is explained in the next section. Finally, we chose to display our diagrams in web pages (of a web browser) so that we can easily connect our module into any IDE. This aspect may seem purely technical but it is not: one of the cognitive dimensions is the visibility in which the juxtaposition of two points of view is a way to easily switch from one to the other. If the IDE

does not allow two large windows to be displayed next to each other (each one on a screen or both on the same screen), it is natively possible with our approach, even when using a tablet or any display device with an OS containing a web browser. However, in addition to these web pages, we also made modeling tools plugins (see Sect. 4.4) that enable live diagrams on them.

3 Related Works

In this part, we first describe works about visualization tools for code, their UML diagrams features as well as the possible interactions and navigation. Then we talk about reverse-engineering tools, especially how they generate diagrams and in what way the generated elements and source code are connected. Finally we conclude with a summary of these reviews and an opening to the presentation of our tool.

Code visualization tools usually allow to have graphical and exhaustive views of projects. These views can use 2D or 3D (for example [13] uses 2D diagrams connected in 3D and 2D diagrams overloaded with information in 3D like a city map with buildings), even in virtual reality. However, they require significant cognitive efforts (understanding the graphic representation, which is often unusual for the developers), as well as large screens or even specific equipment for VR. The advantage of this type of representation is the “macroscopic” view of projects (dependencies between packages or classes, code versioning...). The major drawback of these representations is the “off-line” aspect since they do not allow to reflect in real time the project in its current state. On the contrary, and on a “microscopic” aspect, [14] can see in real time the code of a project in a simplified way and thus making it easier to navigate in it. Unfortunately, this representation is only textual and requires time to adapt to the developer.

Since class diagrams, sequence diagrams and code, share elements, IDE and modeling tools propose more and more often a “find usages” or “find in diagram” command. These commands may be triggered in two ways. The first one is by using the top toolbar menus, but it requires to select which element must be looked for. The second is via contextual menus; in that case, the element is already selected and the menu shows only information and commands about it. One of the easiest way currently implemented in MagicDraw[3] is by adding an item in the elements’ contextual menu. Thus right-clicking an element will bring up the menu and browsing it will allow users to open a related diagram. However this menu is yet too much complicated since it does not just show a list of diagrams but displays the full (UML) path to a related element. Figure 1 shows this menu.

This navigation is indeed based on the UML relations between elements while we have a user-centered approach. This is especially true for tools using EMF[4], such as Papyrus, since the relations are even more complicated and less direct for the user; for instance a class inside a sequence diagram is the type of the property

[3] MagicDraw: http://www.nomagic.com/products/magicdraw.html.

[4] Eclipse Modeling Framework: https://eclipse.org/modeling/emf/.

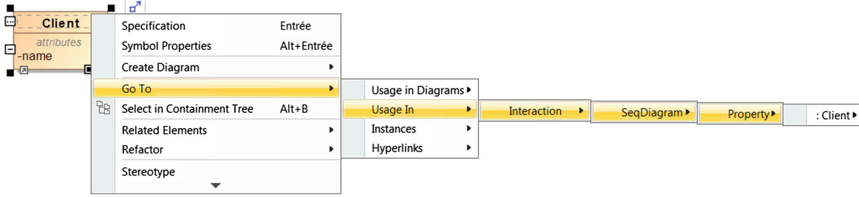


Fig. 1. MagicDraw - class diagram: "Usage In" feature requires 8 clicks

that the lifeline represents (e.g. In Fig. 1, the lifeline property has "*Client*" for type). This navigation action requires at least 8 clicks to switch from a class to any associated sequence diagram. At the opposite, in VisUML, a simple click on a method in the class diagram updates immediately the sequence diagram to display the correct method.

IntelliJ meanwhile implements a similar feature: when displaying a class diagram, it is possible for users to "Jump to Source" or "Find Usage" of any sub-elements (attributes and methods). The advantages of IntelliJ over MagicDraw is that it is directly linked to the code representation. As a result, it is possible to quickly switch from a diagram view to the code. Two visualization modes are available. The first one is classic: the diagram is displayed on a new tab (which can be shown near another tab). The second is interesting as it creates a popup window with the diagram inside. In the popup mode, nothing can be modified, but this allows a quick preview of a diagram. Despite these two modes of visualization, the work context is broken because developers must do several actions in order to generate and see any diagrams.

IntelliJ only supports class diagrams, it is therefore not possible to create the sequence diagram of a selected method. In addition, there is no navigation between diagrams (whether class diagram to class diagram, or class diagram to sequence diagram).

Figure 2 is an example of how IntelliJ shows usages of a class inside the project: an unordered list of all the occurrences of this class, without filtering options. In this example we wanted to find the class that extends *Entity*, the useful results are bordered in green, only 4 of 18 lines are relevant.

Reverse engineering is now widespread among IDE and modeling tools. It allows developers to create a graphical representation of their code, or part of code. All the tools proposing reverse engineering allow developers to produce class diagrams. However, sequence diagrams remain yet less common.

ObjectAid[5] is one of the tools (together with MaintainJ[6], VisualParadigm[7] and MagicDraw) that handle sequence diagrams. Depending on the tools, there are several ways to generate a sequence diagram. All of them (except MaintainJ) use a common menu that allows developers to choose which elements they want

[5] ObjectAid: http://www.objectaid.com/sequence-diagram.
[6] MaintainJ: http://maintainj.com/.
[7] VisualParadigm: https://www.visual-paradigm.com/.

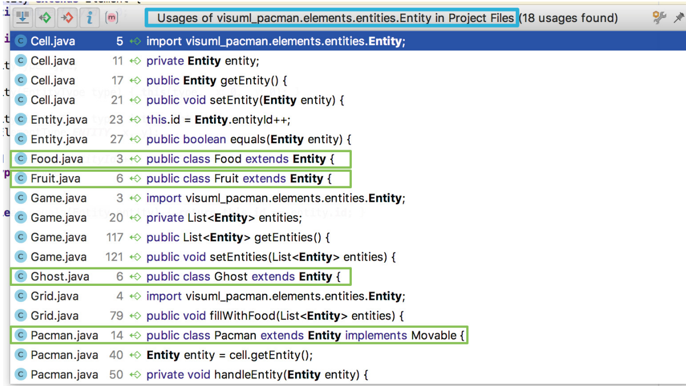


Fig. 2. IntelliJ - find usages of “Entity” (only 4 of 18 lines are relevant)

to reverse-engineer in a sequence diagram. ObjectAid and IntelliJ (only for class diagrams) also add drag & drop support, from any location in the IDE. This means that developers can add elements on diagrams easily, without having to browse the entire project. However, it is still up to them to choose which elements should appear or not. In VisualParadigm, developers have to navigate through four windows, when they want to “instant” reverse a method. For this, they must (1) select the source code folder and (2) find the correct class. Once they’ve found it, they must (3) select the method they want to reverse. This process is complicated as it takes at least twenty clicks.

At last, some tools are specialized in visualization after code execution, such as MaintainJ, which only works at runtime, and ObjectAid which can analyze Java stack-traces. These tools can therefore generate sequence diagrams that reflect the execution of a particular method, but not its complete representation (e.g. “*alt*” or “*loop*” fragments are missing). Although they can generate a lot of sequence diagrams which are interconnected, it is up to the developers to choose which one to display, and then navigate through them, using basic interactions, i.e. right click on a specific invocation to see its own sequence diagram.

Finally, among all the tools we have analyzed, none allows developers to have live diagrams that fit their current task. Some presented solutions propose to display diagrams at runtime, with information and values extracted from stack traces or execution, but this does not necessarily correspond to the active coding task. Moreover, interactions in these diagrams remain basic. Most of the navigation actions must be triggered with contextual menus and clicks on elements, and they simply allow developers to switch from one diagram to one of its sub-diagrams. In addition, some interactions can link the diagram with the code, but most of them use the “search” function. Overall, each navigation interaction forces users to choose from a list of elements, rather than automatically display elements that are relevant to their task.

4 VisUML Presentation

VisUML is a tool composed of two parts: an IDE plugin, presented in Sect. 4.3, and multiple visualization tools[8] of two types of UML diagrams (class and sequence diagrams). These two parts are connected through our communication bus which is named WSE. This bus allows applications to send and receive information in JSON messages. Those messages can contain any kind of information, whether it comes from the IDE or from a diagram.

As previously described, we aim to help developers in their coding task. To this end, all the information displayed on the UML diagrams refers to elements currently opened in the developer's IDE. This tool does not aim to do a full synchronization between the code and the models, but focuses on the active coding task of the developers. As a result, we use a light mechanism of synchronization, using WSE as way of communication.

Figure 3 gives an overview of the interactions between the IDE and both diagrams. In this section, we present the different parts of VisUML: the class diagram view, the sequence diagram view, the IDE plugins and to conclude, a Papyrus class diagram plugin.

The first two sub-sections are split into two parts: displayed elements and interactions, in which we present the elements that our views display, as well as the interactions that are triggered from these views. The IDE plugins part presents the interactions triggered by the IDE, as well as the messages passing through WSE. Finally, the Papyrus plugin sub-section shows that VisUML visualizations also work on modeling tools.

4.1 VisUML Class Diagram View

The UML class diagram of our tool only shows the most important part of a class diagram: the classifiers (class, interface...), their attributes and operations, as well as the links (generalization, association...) between them, and the packages. According to empirical studies on UML in industry [11,15,16], the class diagram is often used in a simple way and informally. The displayed elements and the possibilities offered by our template are thus enough for our use.

Displayed Elements. Once activated, VisUML class diagram displays all the Java elements that are currently opened in the IDE. These elements can be classes, enumerations, interfaces, ... and they may or may not be related to any other element.

In addition to these elements, VisUML also displays unopened elements that have at least one relation with an opened element. However, those elements are differentiated by their opacity, as they appear more transparent than the others.

In order to highlight the element that is active in the IDE, we change the color of the associated graphical representation to green. Moreover, any links

[8] As of today, VisUML visualization tools are available on web pages, Papyrus and GenMyModel plugins.

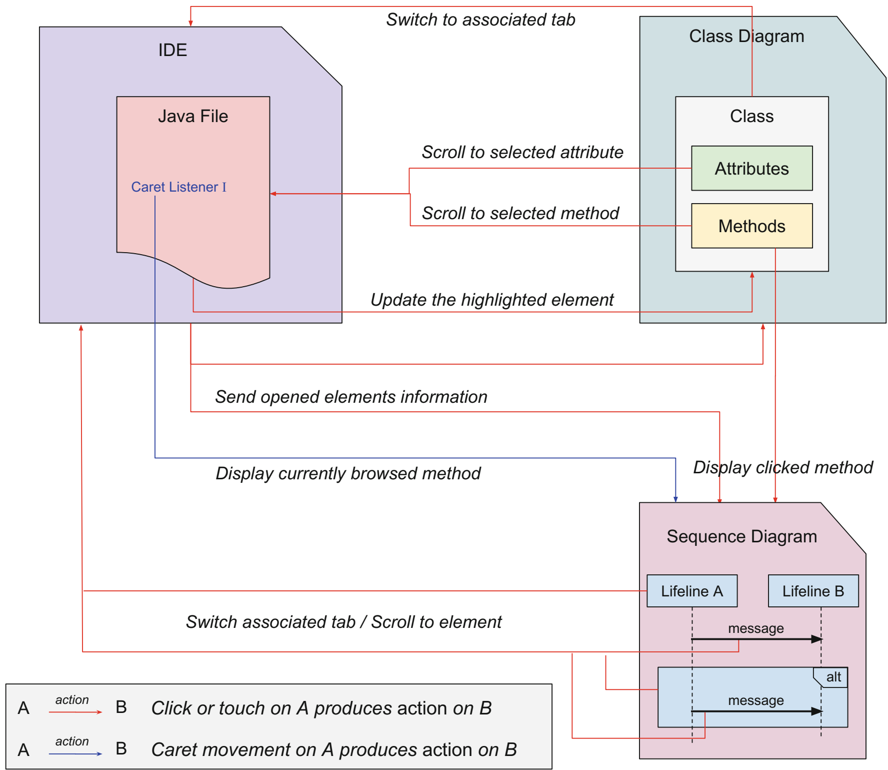


Fig. 3. Navigation interactions and resulting actions (Color figure online)

(i.e. relations) that are connected to this representation appear in bold and red. This allows users to easily detect related elements.

Figure 4 shows a class diagram with the active tab in a different color, as well as unopened related elements.

Filters. In order to allow users to limit the number of displayed elements, we added several filters to our class diagram view.

First, we added 4 visualization profiles (see part 1 of Fig. 5). Each profile has a different configuration, showing or not part of the diagram.

- **Packages only**: Display only packages, without any classifier inside
- **Classes**: Display classes, without attributes or operations
- **Public & Protected elements**: Display classes, with only public and protected attributes or methods. Private fields are hidden.
- **All details**: Default profile. Display everything.

Moreover, a simple button toggles the visibility of unopened related elements. This function aims to quickly switch between a diagram that matches exactly

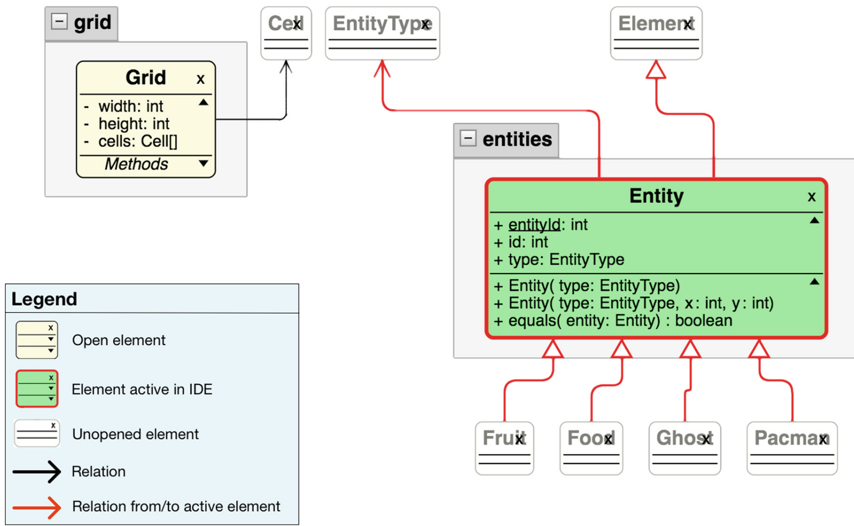


Fig. 4. Class diagram with unopened related elements and active tab highlighted (Color figure online)

opened tabs in the IDE, and a diagram that also shows relations associated to these elements. In the same idea, another button allows to show or hide getters and setters operations in every classes. These buttons are in part 2 of Fig. 5.

VisUML displays two lists of elements (that are currently loaded in the tool). These lists show respectively classifiers (class, enum, interface ...) and packages. They are displayed in part 3 of Fig. 5. On each element of either list, a checkbox allows to change the visibility of the related element. So, one can simply check or uncheck any element, which allows to quickly filter elements according to their name or package.

Finally, we chose to hide some elements that do not provide useful information to developers. For example, we hide the *java.lang.Object* element because it is inherited by all other Java classes. In the same idea, primitives types are also hidden. In the same idea, we can hide elements according to a specific framework or language. For example, when working on an Android project, we hide some Android specific elements such as *android.app.Activity*. There is currently no graphical interface to add or remove elements in this list, but this can be done easily by modifying the source code of the class diagram.

Interactions. In order to make the class diagram visualization interactive, we added interactions on each displayed element. These interactions send messages on our communication bus, and those messages are then received by any connected tools. Interactions in the class diagram view (also visible on Fig. 3) are:

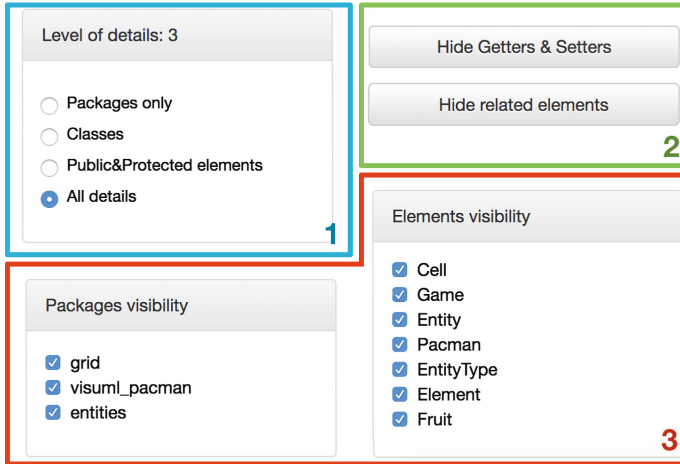


Fig. 5. VisUML filters

- A click on a classifier (besides its attributes/methods) will update the IDE by putting forward this element (changing the active tab, or opening the corresponding file).
- A click on an attribute or a method will switch the active tab to the associated file if needed, scroll the IDE to the definition of the chosen element and highlight or select it. Moreover, in the case of a method, the sequence diagram view will automatically be updated in order to display the selected method body.
- Finally, a click on the cross (X) on the top right corner of an element will delete its graphical representation and close the associated tab on the IDE.

Handling these events allows the IDE to remain entirely synchronized with our graphical representations ("*live diagram*") when the developer navigates inside a diagram or interact with it. It is therefore possible to switch between the two representations (code and model) without losing the context of work (since navigation inside the diagram or the IDE also updates the other). For instance, the selected class (in the diagram) will always be the active tab in the IDE. With this visual aid, the developer does not need to look for the active element. Moreover, a clicked/selected attribute or method in the diagram will always be visible (i.e. the IDE's editor will scroll if needed) and highlighted in the code. Finally, the currently browsed method (in the code, according to the caret location) is emphasized in bold and blue.

4.2 VisUML Sequence Diagram View

In addition to the class diagram, we chose to implement a sequence diagram view, as explained in Sect. 2. This diagram is often used to represent control flow, here the body of a method.

Displayed Elements. UML is often used in an informal way [11,15,16] but we chose to create a representation as close to the code as possible. However our sequence diagram differs from the UML standard in several ways as explained below.

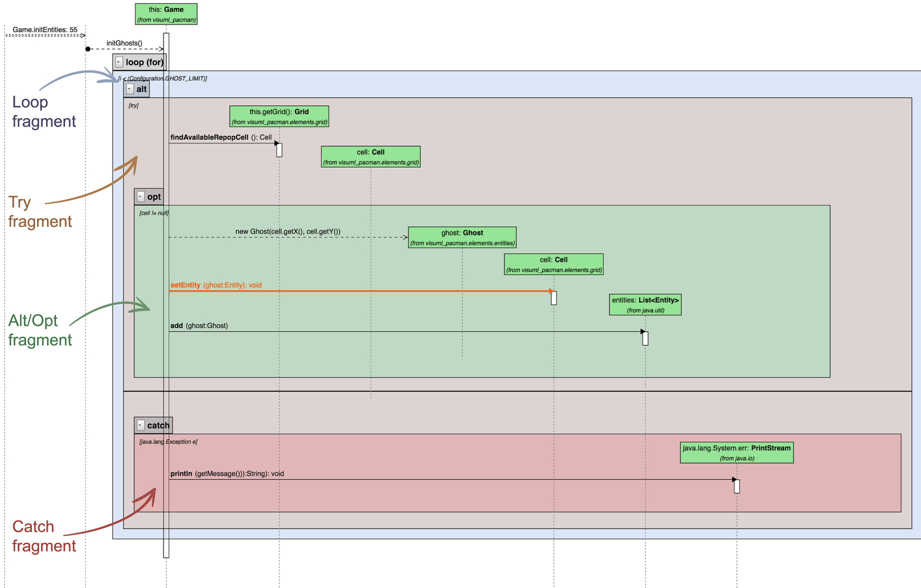


Fig. 6. Sequence diagram with colored fragments and highlighted message (Color figure online)

First, our sequence diagram acts like an UML one, but with more code specific information. For example, we create an “alt” fragment for each “try” and “catch” block and we display invocation details such as parameters (types and values). Moreover, when VisUML users let their mouse on an element, a tooltip appears with all code comments associated to this element. Finally, we added a specific color code on fragments which allows developers to easily recognize the type of a specific fragment, as well as their nesting level. We chose to be closer to the code in order to reduce the cognitive integration of the model, since our goal is to help developers in their current task but not to abstract their code. Figure 6 shows an example of colored fragments.

Interactions. As explained in the previous section, visualizations allow users to interact with the displayed elements. Possible interactions on this diagram are similar to the ones implemented on the class diagram:

– A click on a lifeline (which can be created at any time during the method) will scroll the IDE directly to the associated class, or variable assignment.

– A click on a message (link between two lifelines) or an activity (block on a lifeline), which is linked to a method invocation, will scroll the IDE to the corresponding code line.
– A click on a group, or fragment (alt, loop, ...), will highlight this group on the IDE (full selection of all the lines of this group).

In addition to these interactions, we added a caret listener in the IDE. As a result, when the developer moves the caret, we check if it is in the body of a method and send a message containing information about this method, as well as the line where the caret is located. This message is then received by the sequence diagram view, which automatically updates its display to show the graphical representation of this method. Moreover, since the message contains the location of the caret (the line in the code file), the sequence diagram knows what UML message (represented by links between Lifelines) is concerned and we highlight it by changing its color. Figure 6 shows an example of highlighted message (message *setEntity* is orange and bold).

Since the communication between the IDE and the visualizations is very fast (less than 100 ms between sending and receiving a message in normal conditions), it is possible for the users to see, in real-time, the sequence diagram lights up (highlight the related message) when the cursor moves in the code. This is especially convenient when browsing methods inside a file.

4.3 VisUML IDE Plugins

The current VisUML IDE implementation refers to an IntelliJ[9] plugin and an Eclipse[10] one. IntelliJ is a Java IDE developed by JetBrains. Android Studio[11] is based on this IDE, with specific functions for Android developers. IntelliJ uses a plugin system, allowing us to add functionality to any IntelliJ based IDE.

As mentioned, IntelliJ (as well as Eclipse) offers an API that entirely manages the interactive part of the IDE. Therefore it is possible to add listeners on any kind of event. In our case, we are mainly listening to five events:

1. A file has been opened
2. A file has been saved (or its content has been modified)
3. The user has changed the active tab
4. A tab has been closed
5. The caret has been moved

Each handled event contains parameters, primarily the name of the concerned file, which allows us to make a link between the code element inside it and the file name, as well as create a representation of that element (in which we store every needed information, such as its flags, attributes, methods, relations, etc.). Once the event has been received and the element identified, the plugin sends a specific message via WSE, to give an order to the connected applications.

[9] IntelliJ: https://www.jetbrains.com/idea/.
[10] Eclipse: https://eclipse.org/.
[11] Android Studio: https://developer.android.com/studio/index.html.

For events 1, 2 and 3, a *"createOrUpdateUML"* message is generated with all the information of the class (or several messages, if there are intern or anonymous classes in addition to the main class). These messages are then sent to WSE. Once received the UML class diagram (whether on the web page or on Papyrus) will create or update the UML graphical representations associated to this element.

The event 3 also sends a message of type *"highlightClass"*, containing the main class of the file in foreground on the IDE; it allows to put forward its graphical representation. In this way, the active tab of the IDE will always be highlighted on the diagrams, whether via a different background coloration, a flashing border or a zooming effect. Figure 4 shows an example of highlighted element.

The event 4 creates a *"remove"* message (or several) with the Fully Qualified Name (FQN)[12] of the element that has been closed by the user as parameter.

Finally, the event 5 sends a *"highlightMethod"* message, which is filled with the active class information, the currently browsed method and the line on which the caret is. This event is triggered by following the position of the user's caret in the code.

These five events allow the IDE and the diagram views to be synchronized at any time.

4.4 Plugins for Modeling Tools

In addition to the visualization into web pages, we also made a Papyrus plugin that listens to the same messages as these web pages, in order to create UML class or sequence diagrams. Papyrus[13] is an UML modeling tool based on Eclipse. It is developed by the Laboratory of Model Driven Engineering for Embedded Systems (LISE) which is a part of the French Alternative Energies and Atomic Energy Commission (CEA-List)[14]. Papyrus can either be used as a standalone tool or as an Eclipse plugin.

Our VisUML Papyrus plugin is still an early prototype. When connected to an IDE, the current Papyrus model shows a live representation of the opened tabs of the IDE. Several interactions have also been implemented, such as the click on a class (*switchToClass* message) and on an attribute or method (*highlightAttribute/highlightMethod*).

The advantages of Papyrus compared to a web page is that it implements a lot of useful features to UML diagrams, such as formal validation, exportation in various formats, easy refactoring, code generation, etc. Even if in our web pages the goal is to have live and transient diagrams, Papyrus allows users to store these diagrams and do UML operations on them.

In the same idea, we are developing a GenMyModel[15] plugin that works the same way. GenMyModel is an online UML editor that provides collaboration to

[12] A FQN is an unambiguous name that specifies an object (e.g. *com.myapp.model.Client*).

[13] Papyrus: https://eclipse.org/papyrus/.

[14] CEA LIST: http://www-list.cea.fr/.

[15] GenMyModel: https://www.genmymodel.com/.

its users, as well as a history mechanism. With this history, each modification is saved and can be *replayed*, helping users to understand the evolution of the UML model.

To sum up, VisUML is independent of IDE and modeling tools or UML visualizations. Any modeling tool with an open API and connectable to WSE (i.e. able to send and receive HTTP requests) can be connected to VisUML.

5 VisUML Contributions and Discussion

In this section we will describe what are the contributions of VisUML. At first, we will show that according to its technical implementation, VisUML is a flexible tool. Then we will point out that it is developer-centered, which is an important concept for such a tool, since we aim to ease the work of developers. With all these points, we partly answer to Chaudron's vision [17] (e.g. "mixing formal notations with informal notations", "higher level of integration of tools"). Indeed, we use both formal and informal representations of UML models. More, we make sure that our tools are focused on one specific aspect while being connected to each other. Finally, even if there is not yet any particular gestural interaction, our tools work on tablets and can manage touch and gestural events.

5.1 Distributed and Linked Applications

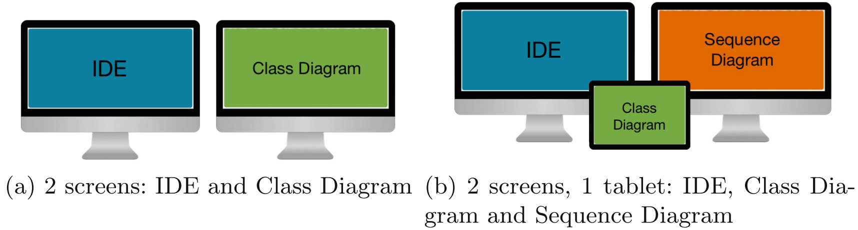
VisUML is composed of several applications that are independent and simply connected through WSE. It is therefore easy to add an application at anytime, or even replace one by another (e.g. UML visualization can be done by the GoJS[16] web pages as well as with the Papyrus plugin. With VisUML, a developer can switch at anytime between those two visualizations). Figure 7 shows how VisUML can be used in different work contexts.

In addition, because of their technologies and implementations, these applications are able to run on any platform and OS (Java works on Windows, OS X and Linux). The web pages can even work on Android and iOS, in any web browser app. Moreover, the UML class diagram view automatically switches to a specific template when it is displayed on a smartphone (with a "small" screen). This template shows only the name of the classes, but all the links between them. A simple click (or touch) on a class display all its details.

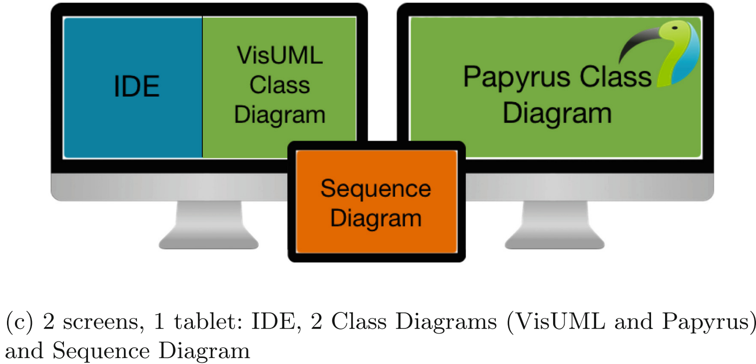
Finally, the applications are able to run on different devices (PC, tablet and smartphone) at the same time, allowing developers to use this tool in their environment without changing their habits.

VisUML is then entirely multi-platform; developers can develop on a Mac or PC, visualize their live diagrams on the same machine with a different screen, or on their tablet or another machine. They can also have a PC with their code, a laptop with Papyrus showing their UML class diagram, and a tablet with the sequence diagram on it. Thanks to WSE, each view is synchronized and interactive.

[16] GoJS: https://gojs.net/.



(a) 2 screens: IDE and Class Diagram (b) 2 screens, 1 tablet: IDE, Class Diagram and Sequence Diagram



(c) 2 screens, 1 tablet: IDE, 2 Class Diagrams (VisUML and Papyrus) and Sequence Diagram

Fig. 7. Examples of VisUML contexts of work

5.2 Displayed and Highlighted Information

In addition to this liberty of devices and displays, we aim to ensure that VisUML is current-activity centred. Rather than reverse-engineering all the project to create a gigantic and unreadable class diagram, we chose to only show the opened tabs of the developer's IDE. Although this solution works and allows the user to switch easily between the two representations (code and model, since they are exactly the same), a simple UML class diagram generation from opened tabs has shown that some information were missing. Indeed, in the code, a developer can rapidly know if a class extends another class or not. However, if a diagram only shows opened Java classes, this kind of information will not be displayed.

In order to fix this lack of information, in addition to every opened elements, VisUML displays all their related elements (an element can be in relation with another if one extends or implements the other, or if there is an association between them), even if they are not opened. This allows the developer to easily navigate in the graphical representation without losing their context.

5.3 Simplify the Use of Diagrams

Another important aspect in VisUML is to reduce the number of required actions for the developers to see and navigate in their diagrams. In most of the current

IDE or modeling tools, developers have to select which classes or elements they want to reverse-engineer or model, by navigation through the list of all the possible elements of the project and checking which one they want to process.

This results in a loss of the work context of the developers. They have to stop their work, take time to select what to show, and then they can resume their task. This is a big waste of time and concentration, because they changed their context and active task (cognitive overload of short-term memory), then they will have to do some cognitive work to restart their task. Moreover, if afterwards they realize that they forgot some elements, they will have to redo all the process, which will result in another waste of time, as well as a discouragement to build UML diagrams.

With VisUML however, we want to reduce to a minimum this waste of time and the consequences. Once developers started the plugin, everything is done automatically, without any actions from them. Obviously, developers must do some actions in their code (or IDE), for example open, edit or close a file, but nothing specific to VisUML. They just have to work as usual with new live diagrams views of the code on the desired screens and devices. If they want to see an element, they just have to open it. If they open an element A that extends another element B, then B will be shown (with lower opacity), and a simple click on B will open the corresponding code in the IDE. Therefore, in addition to easily navigate between UML diagrams and the corresponding code, VisUML adds a new and efficient way of navigation inside UML diagrams; indeed, a simple click on a method in the class diagram will update the sequence diagram to display the representation of this method.

6 Discussion

In this section, we discuss some of the side effects of VisUML, as well as improvements that can be made to this tool in order to makes it even more efficient.

6.1 Malleable Environment

Our first goal with VisUML is to adapt to the current work environment of developers. Instead of making a new IDE or modeling application, we created plugins that work with commonly used IDEs. This reduce the changes needed to adopt the tool and so ease its use.

In addition, we aim to add flexibility to the environment by separating features in multiple applications that share data to each others. With VisUML, most of the data come from the code and are transmitted through our communication bus (WSE). Any compatible application can connected to WSE to listen data and send actions if needed. This On-The-Go principle allows users to configure their work environment as they wish.

Moreover, there is no limit on how many devices or applications can connect to WSE. It is possible to be in any kind of configuration. For example, a user could have VisUML views and code, another Papyrus and VisUML sequence

diagram view. A third could use only its IDE while sending data to others. Then a fourth could use a tablet with GenMyModel.

On another hand, UML diagrams are not only used by developers. They are commonly used in firms by multiple peoples, all of them doing specific actions. We previously describe “profiles” in the class diagram view. The profiles we defined are developers oriented, but we thought about job oriented profiles. According to its job, a user could see only a part of a diagram, add extra information such as code quality, or even implementation of specific view, like Android layouts or SQLite databases.

As explained in Sect. 4.4, in addition to web pages, VisUML views are also available as a plugin for Papyrus (and soon GenMyModel). This allows developers to benefit from the modeler features (e.g. save, UML validation, XMI export, code generation...) without having to create the model and diagrams, as they are generated by VisUML. This aspect validates one of our goals, which was to adapt to the developers environments by implementing live diagrams on modeling tools.

6.2 Evolutions

Because our goal is to display information about the active task and not to ensure a complete synchronization, we did not add model to code transformation even if it would be easy to do it. This would allow a bidirectional modification between the model and the code, which would increase the efficiency of VisUML by reducing the number of switches between code and model.

In the same idea, many refactoring interactions are considered (some of them currently tested in a prototype) in both the class and sequence diagrams. For example, messages (links) in a sequence diagram could be moved, which implies a reorganization of the invocations order (code refactoring), as well as their belonging to a fragment or not. In the same idea, a simple deletion of the graphical representation of a link or a group could delete their corresponding element in the code. Likewise in the class diagram, developers could be able to move elements in or out packages.

Finally we also thought about adding new listeners on the IDE. For example, listeners on the code execution would enable information of debugging (breakpoints, execution errors, ...) to be displayed on diagrams. In the same idea, listeners on the syntax errors or code errors (shown in the IDE) could also be displayed on a sequence diagram (e.g. an incorrect type or a private method called in a wrong context would be highlighted).

7 Conclusion

In this paper we presented VisUML, a tool that helps developers in their coding and debugging tasks. We explained the current limitations of visualizations tools and the contributions provided by VisUML.

Whether it is to understand a project or to find interesting classes, developers spend their time looking for files, classes or methods. These subtasks disrupt developers concentration, as they force them to switch from one activity (e.g. write code) to another (e.g. find an element) and thus it breaks their context of work and overload their short-term memory. Indeed, most of the time, developers look at the code editor part of their IDE. However in order to find files they must look at menus, sub-menus and other windows, which is not efficient and results in a waste of time.

UML could help them to quickly understand a project architecture and its elements relations, since graphical representations of the code are more efficient for developers to understand all the existing relationships than text. However, modeling tools and UML itself are not generally used, or used in an informal way. Nowadays tools allow developers to reverse-engineer the entire project, which results for instance in a unreadable UML class diagram. It is also possible to create a class diagram using a subset of the project, but this requires a lot of interactions and time for developers, as they have to select which files should be considered.

Finally, we show that with our tool, developers could be able to access a view displaying live diagrams of their projects. These diagrams are designed to be easy to read as they only use information provided by the developers IDE.

We focused on two UML diagrams for the moment, but we do consider extending this set to other UML diagrams, such as Activity diagram (work in progress) or Object diagram. In the same idea, since our tool is not a modeling tool, we decided to use UML diagrams informally, and add extra information on them. For example, we thought about Android activities representations, using the XML layout, as well as the *intents* to build links between those activities. This would really be helpful for Android developers to be able to see their views and the links between them in a simple interface, without having to navigate through different files (activities and layouts files).

On the other hand, improving the visual representation of our diagrams would be interesting, but these changes are slightly outside the main domain of our work, which are interaction and navigation for the developers. Indeed, another thesis in the team [18], and in collaboration with CEA LIST, works on UML representations and semiology of graphics, which is complementary to this work.

References

1. Girba, T., Chis, A.: Pervasive software visualizations (keynote). In: Proceedings of 2015 IEEE 3rd Working Conference on Software Visualization, VISSOFT 2015, pp. 1–5, September 2015
2. Brooks, R.: Towards a theory of the cognitive processes in computer programming. Int. J. Hum.-Comput. Stud. **51**(2), 197–211 (1999)
3. Davies, S.P.: Skill levels and strategic differences in plan comprehension and implementation in programming. In: Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V, pp. 487–502. Cambridge University Press, New York (1989)

4. Détienne, F.: Expert programming knowledge: a schema-based approach. In: Hoc, J.-M., Green, T.R.G., Samurcay, R., Gilmore, D. (eds.) Psychology of Programming. People and Computer Series, pp. 205–222. Academic Press (1990)
5. Olson, G.M., Sheppard, S., Soloway, E. (eds.) Empirical Studies of Programmers: Second Workshop, p. 263. Ablex Publishing Corporation, Norwood (1987)
6. Davies, S.P.: Externalising information during coding activities: effects of expertise, environment and task. In: Empirical Studies of Programmers: Fifth Workshop, pp. 42–61 (1993)
7. Larkin, J., Simon, H.A.: Why a diagram is (sometimes) worth ten thousand words. Cogn. Sci. **11**(1), 65–99 (1987)
8. Church, L., Marasoiu, M.: A fox not a hedgehog: what does PPIG know? In: 27th Annual Workshop on PPIG 2016, pp. 17–31 (2016)
9. Green, T., Petre, M.: Usability analysis of visual programming environments: a 'cognitive dimensions' framework. J. Vis. Lang. Comput. **7**(2), 131–174 (1996)
10. Moody, D.L.: The "physics" of notations: toward a scientific basis for constructingvisual notations in software engineering. IEEE Trans. Softw. Eng. **35**(6), 756–779 (2009)
11. Lethbridge, T.C., Ave, K.E.: Perceptions of software modeling: a survey of software practitioners table of contents. In: 5th Workshop From Code Centric to Model Centric: Evaluating the Effectiveness of MDD (C2M: EEMDD), pp. 1–102 (2008)
12. Dzidek, W.J., Arisholm, E., Briand, L.C.: A realistic empirical evaluation of the costs and benefits of UML in software maintenance. IEEE Trans. Softw. Eng. **34**(3), 407–432 (2008)
13. Gregorovic, L., Polasek, I.: Analysis and design of object-oriented software using multidimensional UML. In: Proceedings of the 15th International Conference on Knowledge Technologies and Data-Driven Business, pp. 47:1–47:4 (2015)
14. De Line, R., Czerwinski, M., Meyers, B., Venolia, G., Drucker, S., Robertson, G.: Code thumbnails: using spatial memory to navigate source code. In: Proceedings - IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2006, pp. 11–18 (2006)
15. Chaudron, M.R., Heijstek, W., Nugroho, A.: How effective is UML modeling?: an empirical perspective on costs and benefits. Softw. Syst. Model. **11**(4), 571–580 (2012)
16. Petre, M.: UML in practice. In: Proceedings - International Conference on Software Engineering, pp. 722–731 (2013)
17. Chaudron, M.R.V., Jolak, R.: A vision on a new generation of software design environments. In: HuFaMo@ MoDELS, pp. 11–16 (2015)
18. El Ahmar, Y., Gerard, S., Dumoulin, C., Le Pallec, X.: Enhancing the communication value of UML models with graphical layers. In: Proceedings of 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, MODELS 2015, pp. 64–69, September 2015