Reproducible moldable interactions

Master Thesis

Mario Kaufmann
from
Thun BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

April 2018

Prof. Dr. Oscar Nierstrasz
Dr. Andrei Chiș

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Abstract

Object inspectors are tools that allow developers to explore the state of run-time objects. This exploration creates many interaction events between the developer and the inspector. Recording, saving and using those interactions directly in an inspector opens opportunities to reduce the amount of repetitive actions developers need to do during development and debugging.

To make this possible we propose an inspector model that records developer interactions as first-class entities and uses them to reduce repetition. This is enabled through a model that uses a tree to keep track of an inspection session, and a recording infrastructure that allows each widget to decide how user interactions should be recorded.

To validate this model, we identify several types of problems that can arise in object inspectors and show how they can be addressed if developer interactions are recorded by the inspector. For example, the new model allows developers to replay inspection sessions, restore partial navigation and generate code from an inspection session.

Contents

1 Introduction

Software developers interact with their software systems exclusively through the use of development tools. This interaction between developers and tools generates many types of events, like mouse clicks, method selections, code edits, *etc.* The recording and analysis of these types of interaction data can provide improvement opportunities for development tools [13]. Nonetheless, many times development tools are designed to only record and store this information with the goal of having it available for later offline analyses. However, this does not allow development tools to take live advantage of the recorded events and adapt their behavior at run-time to improve the development experience of developers.

To investigate how to address this, in this thesis we take a look at a specific category of tools, *i.e.,* object inspectors, and propose a model for an object inspector designed with the goal of recording user interactions and using them to improve the inspection process.

1.1 Object inspectors

Object inspectors are development tools that give developers access to the state of run-time objects. They are often integrated in debuggers, however, can also exist as standalone tools within IDEs. Object inspectors allow developers to see the values of attributes and dive into the structure of objects to explore how they are composed. They can further provide different ways of visualizing objects and allow developers to create their own custom views for domain objects [4].

When interacting with object inspectors, developers often use them to explore many interconnected objects [12]. These interactions form an inspection session that captures the history of the inspected objects together with the actions performed to explore those objects. Hence, these interactions capture the intention of what the developer was doing together with how she did that.

Figure 1.1 shows an example of an inspection session where the developer is exploring the content of an object representing an XML file. Initially the developer starts by inspecting the object directly. Then the developer uses a snippet of code to parse the file using an XML parser into an object representing an XML document, and inspects the result (step 2). Objects representing an XML document have a custom presentation showing a tree representation of the

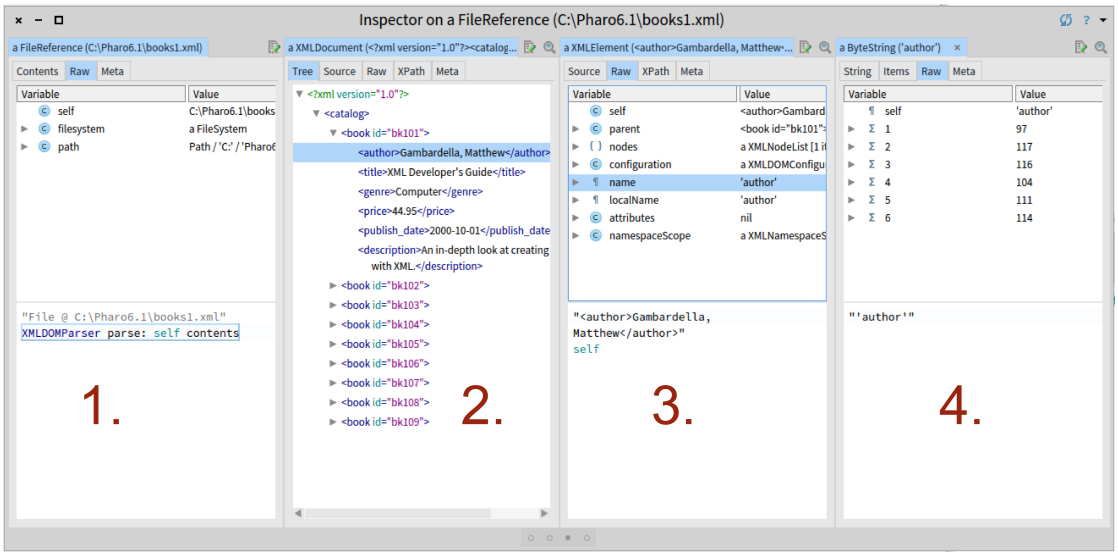

Figure 1.1: An example of an inspection session in an object inspector consisting of exploring an XML file: *(1)* the content of the file is is parsed using a code snippet; *(2)* using a custom view for XML documents a node in the document tree is selected; *(3)* an instance variable is selected and the resulting object is inspected *(4)*

document's contents. In that presentation the developer selects a certain interesting node. Last but not least, the developer ends this inspection session by inspecting one instance variable of that XML node object. Now in Figure 1.1 the developer has an inspection session of four steps that captures how she interacted with an object representing an XML file.

Not recording, saving and using saved inspection sessions directly in an inspector at run-time introduces several limitations during development that increase repetitive actions. The following are four types of problems caused by this:

Losing inspection sessions: if inspection sessions are not saved when closing the inspector, the inspection session is lost. If this is done by mistake, the developer will have to manually redo the exploration to get to the same point;

Manual repetitive explorations: after performing an inspection session on an object a developer could need to do it again on the same object at a later stage of the execution, or perform the same inspection on another object of the same type. If the object inspector does not offer the possibility of replaying previous sessions, this has to be done by hand;

Manual coding: depending on her task, a developer could need to create a code snippet that accesses the final object of the inspection session starting from the initial one. If the inspector does not offer the possibility to generate code from an inspection session she needs to write this snippet by hand;

Manual sharing: Inspection sessions can be helpful to explain to other developers how to extract information from an object graph to answer specific questions. Not having a mechanism to share sessions forces developers to explain how to redo the session manually or to use screenshots.

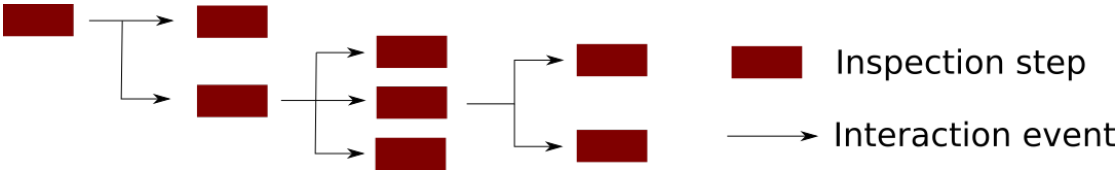

Figure 1.2: Modeling the inspection session as a tree of inspection steps rather than a list of steps. Every step can have any number of child steps. The interaction between steps is captured as interaction events.

The above problems could be solved by object inspectors using various dedicated mechanisms for each one. In this thesis we start from the premise that they can all be addressed in an uniform way by recording developer interactions within the inspector and using those interactions as first-class entities. We then show how to design an inspector model based on this idea and how to apply it to solve the four aforementioned problems.

1.2 Extending the moldable inspector model

In this thesis we build on top of the moldable inspector model [4]. This is an inspector model that allows every object to have multiple custom presentations, and groups inspected objects together into an inspector session. While the model provides the notion of an inspection session it does not provide any mechanism for recording and restoring inspection sessions.

First, to enable recording of inspection sessions, we extend the model with the notion of an *interaction event*. An interaction event models an action performed within the inspector, is stored directly within the model and knows how to reproduce itself. Every navigation between objects in the inspector is done through an interaction event. An inspection session can then be replayed by starting from an initial object and applying all interaction events that correspond to a navigation between objects from the session.

Second, to keep all interactions that happen within the inspector, we extend the model to store the steps in an inspection session using a tree rather than just a simple list, as is currently the case with the moldable inspector model. With this approach, every inspection step from the model points to a list of other inspection steps. This structure is illustrated in Figure 1.2. The most recent path through the inspection session is obtained by moving from an inspection step to its most recent child.

1.3 Moldable interactions

A challenge with relying on interaction events is in recording relevant interaction events given the fact that the moldable inspector model allows developers to create many different types of custom views. Hence, every view could have its own custom widgets for displaying information, navigating to the next object or visually highlighting the selected object. The mechanism for recording and replaying inspection sessions should however work for all views.

To ensure that we can record events for all widgets, but also that we can record high-level interaction events when possible, we propose a recording mechanism where the interaction with any widget (*i.e.,* any graphical element) is recorded by asking the widget to specify how the interaction should be recorded. By default every widget can respond to this request and return a generic interaction event. Custom widgets from domain-specific views can return different types

of events that capture more domain-specific information about the interaction. Domain-specific views can further customize standard widgets to return custom interaction events.

Replaying a set of events in order to replay an inspection session is achieved in a similar way by dispatching through the widget: every widget is passed an event that it previously created and asked to simulate that event. Every widget knows how to simulate a generic event, and custom widgets know how to simulate their own events.

Outline

The remainder of this thesis is structured as follows:

Chapter 2 discusses related work in the area of tools for recording and analyzing developer interactions and object inspectors;

Chapter 3 describes the problem being addressed in detail;

Chapter 4 presents a new moldable inspector model based on moldable interactions;

Chapter 5 describes implementation details about the proposed solution ;

Chapter 6 summarizes and concludes the thesis.

2 Related Work

In this chapter we explore several works related to this thesis. First we briefly look at several object inspectors and how they handle interaction events. Second, we describe research about the interaction of developers with their tools. Then, we look at existing tools that provide capabilities of recording and reproducing interactions. Finally, we introduce in more detail the technology stack which the implementation of our approach is based on.

2.1 Object inspectors

Traditional IDEs typically provide views that show object attributes and a customizable textual representation. Furthermore, they allow developers to execute code in the context of an object. For example, the Eclipse IDE[1] provides an object inspector that uses a tree view to visualize object state. Through *detail formatters* and *logical structures* developers can customize the representation of objects. A detail formatter is a snippet of code that constructs a custom string which can be used to represent instances of that class anywhere in the inspector. Furthermore, each class can have a logical structure that can override the default key-value pairs that are displayed for an object in the inspector. Netbeans[2] offers the notion of *variable formatters* through which custom views for an object can be defined. IntelliJ also supports custom views for objects using *data type renderers*. Still, neither Eclipse, Netbeans nor IntelliJ provide a way of recording and managing an exploration session directly at the level of the object inspector. Sessions can only be captured by using generic event recorders that work at the level of the entire IDE and record events like mouse clicks or keyboard strokes.

Move advanced object inspectors improve the way in which developers interact with objects. jGRASP [5], for example, allows objects to have visual representations not limited to text. It automatically constructs custom views for objects based on their internal structure: for an object representing a tree, a tree view would be automatically used. LIVE [3] can create visualizations for data structures automatically from ASTs. As soon as the developer enters a program, LIVE

[1]Eclipse IDE - `http://www.eclipse.org/`
[2]Netbeans IDE - `https://netbeans.org/`

parses it and the resulting AST is then used to create an animated visualization showing the evaluation of the data structure. The developer can change the visualization and the changes are reflected back to the code that created the visualization. These inspectors, however, also do not have dedicated recording mechanisms.

Debugger Canvas [7] is a tool that extends the navigation mechanism of traditional object inspectors by displaying each object in a bubble and linking objects in an exploration session. The bubbles stem from the CodeBubbles paradigm [2]. Through this developers can always reason about how they reached an object. Debugger Canvas offers the possibility of exporting and loading interaction sessions. However, the saved sessions are only used for sharing information between developers and not for live improving the inspector. Like the solution proposed in this thesis, Debugger Canvas also has a tree model for representing an inspection session.

2.2 Interaction between developers and tools

Developers generally spend a lot of time on program understanding. Minelli *et al.* performed an analysis of how much time developers spend on what activity during development [14]. Their findings suggest that developers spend about 70% of their time performing program comprehension. 14% of their time is spent fiddling with the UI and only 9% of the time can be accounted to navigating and editing source code.

In addition to spending time on interacting with source code, developers interact with their integrated development environment (IDE) in many different ways. Minelli *et al.* show that IDEs can often be seen in a state described as *chaotic*. They use the term chaotic to describe an environment in which a developer is forced to open many UI components at the same time to reveal the relationships between code entities for the task at hand. Trying to quantify the level of chaos of an IDE showed that developers spend more than 30% of their time in a chaotic environment [17].

Interaction data can potentially be a valuable source of information that can be leveraged to improve support for the developer through the IDE. However, this data is largely unused by modern IDEs. *Interaction-Aware Development Environments* [13] are IDEs that collect, mine and leverage the interactions of developers to support and simplify their work-flow. For example, they could monitor how developers navigate source code and the IDE could suggest the program entities that might be relevant for a particular task. With Interaction-Aware Development Environments three issues need to be tackled to make it possible: first of all the interaction data has to be modeled and recorded since it is not persisted by default. After that, an interpretation of the data is necessary. From the raw data itself it is often not possible to conclude how it could be useful to improve the IDE and as a consequence be useful for the developer. Finally, the design and implementation of the IDE have to be updated using insights gathered from the collected data.

Another possible approach for supporting the developer through in-IDE events was proposed by Sebastian Proksch *et al.* [20]. They combine the change history of source code from version-control systems with an analysis of in-IDE events into so-called enriched event streams. These event streams not only capture developer activities in the IDE but also specialized context information. This could for example be source-code snapshots for change events. To be able to store such code snapshots they introduced a new intermediate representation called *Simplified Syntax Trees (SSTs)* and built *CARET*, a platform that offers reusable components to conveniently work with enriched event streams.

Gail C. Murphy *et al.* introduced a usage monitoring approach to allow tool builders to sample how developers are using their tools in the wild [18]. This data can be used to prevent

Tool	low-level	file	method	navigation	layout	VCS	high-level (other)
Mylyn					x	x	
Mylyn + PLOG		x	x		x	x	
Mylyn + IDE++	x	x			x	x	
CodingSpec./Trac.	x	x				x	x
Fluorite	x	x					x
Damevski *et al.*	x			x			x
FeedBags		x		x			x
DFlow			x	x			x
Selenium	x						

Table 2.1: Summary of the recording capabilities of some development tools. On the left the respective tool is listed. The individual columns specify the features that the tools provide.

feature bloat and to evolve the environments according to user needs. Furthermore, they state that information about how developers work in a development environment can also provide a baseline for assessing new software development tools.

Given the importance of program comprehension and tool navigation in the development work-flow it makes sense to attempt to streamline the process for the developer as much as possible. Collecting in-IDE events typically is the first step in that process. These events have to be analyzed and potentially enriched with additional information from external sources such as version control systems. The resulting data can then influence the design of development tools.

2.3 Recording and reproducing interaction

In order for tools to leverage developer interactions they first have to collect them. There are many different approaches to this problem. Some tools record low-level events such as mouse clicks or keyboard shortcuts. Some others use more high-level information, for instance source code modifications or currently opened files. To better understand the current state of the art, we investigated several tools that record developer interactions and looked at the mechanism for doing the recording. Table 2.1 summarizes the different type of recording that we encountered in those tools. The columns can be interpreted as follows:

- low-level: tools that record low-level events such as mouse clicks or keystrokes
- file: tools that record which files are modified
- method: tools that record which methods are modified (more granular than file modifications)
- navigation: tools that record navigation actions (*e.g.,* clicking a button to open a toolbar) which lead to a certain state of the user interface
- layout: tools that record the state of the user interface itself (*e.g.,* opened files and windows)
- VCS: tools that record events concerning a version control system (*e.g.,* a commit)
- high-level (other): tools that record other high-level IDE events (*e.g.,* refactoring, building, executing a test)

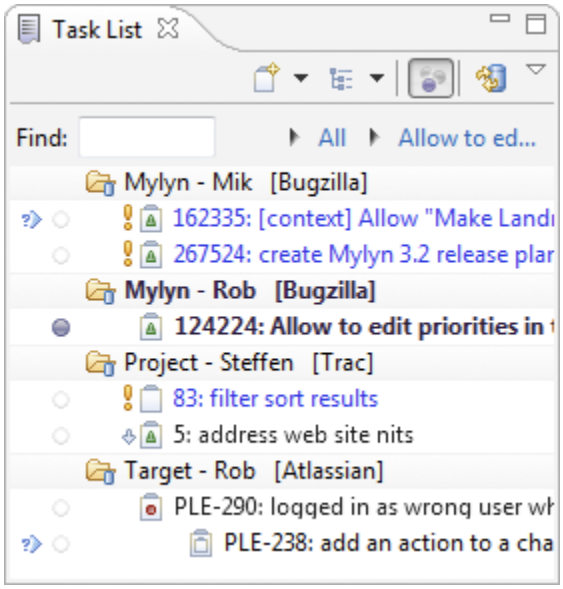

Figure 2.1: The task list window of Eclipse Mylyn. Developers can switch between the tasks. Each task has an associated context describing open files, windows and tools.

Eclipse Mylyn [19] is the task and application life-cycle for the Eclipse IDE. It is based on a degree-of interest model for IDEs [10] and was originally called Mylar. Mylyn assigns elements of the IDE a degree of interest. Elements with a low degree of interest are hidden from the developer while elements with a high degree of interest are kept visible and easily accessible to the developer. The degree of interest is determined through the interaction of the developer with the IDE. For example if a user closes a certain source file Mylyn assumes that it bears little to no significance for the current context.

Eclipse Mylyn allows the developer to record their activities while they are working and persist them to a task. They can decide to start a new task at any time and switch back and forth between the old task and the new task without losing the context they have built. Figure 2.1 shows a task list window in Eclipse. It contains multiple open tasks between which the user can switch back and forth.

Fritz *et al.* refined the degree of interest model of Mylyn with a degree of knowledge (DOK) model [8]. Instead of just considering how much a developer interacts with a certain source file the model also takes code authorship into account. In the model, knowledge about code is not just obtained by authoring it but also by using it. A DOK value is specific to a developer. Different developers can have different values for the same source code element. Similar to DOI, Mylyn can use the DOK value to determine which source code elements to show to the developer.

Kobayashi *et al.* developed PLOG as an extension of Mylyn [11]. It captures interaction histories inside the Eclipse IDE. PLOG records source code interactions both at file and method level. During recording it distinguishes between actions that modify the source and and actions that only reference it. In addition to the standard capabilities of Mylyn, PLOG tries to predict file- and method-level changes using recorded interaction histories.

A plug-in extending Mylyn is IDE++ [9]. It was introduced by Gu *et al.* In comparison to Mylyn it captures more fine-grained interaction data. Instead of predefining what kinds of interaction can be captured IDE++ records event at the lowest level possible, down to a single

keystroke. This allows users of IDE++ to realize developer-aware applications that the authors did not anticipate by defining their own events to be recorded.

Another set of tools that provide data collection for Eclipse are CodingSpectator and CodingTracker [22]. They modify Eclipse to collect more data in a minimally intrusive way. They were developed with the focus on refactoring operations. Whenever a developer initiates a refactoring operation CodingSpectator records whether the operation was actually performed or canceled, what kind of refactoring was performed (renaming a variable, extracting a method, *etc.*), which source code element was selected, how the user configured the refactoring and other details about the operation. CodingTracker tracks fine-grained source code modifications down to the insertion or deletion of a single character. Furthermore it also integrates with version control systems to have access to code commits.

Another plug-in for the Eclipse IDE is Fluorite created by Yoon and Myers [23]. It logs low-level events in the code editor. Three types of events are recorded: commands (*e.g.,* copy-pasting, typing text or moving the cursor), document changes (*i.e.,* changing the active file) and annotations.

Most research about recording and reproducing interaction focuses on the Eclipse IDE and Mylyn [13]. There also exist tools not related to the Eclipse IDE.

Damevski *et al.* recorded and studied interaction data inside the Visual Studio IDE[3] [6]. The tool is implemented as an extension of the IDE. It records all the user events provided by the IDE to plug-ins. These interactions encompass IDE commands (*e.g.,* building or executing a test), views (*e.g.,* clicking on a specific window), or events (*e.g.,* build finished). The tool also records low-level events such as keystrokes or a moved cursor and aggregates those to reduce data size.

Ammann *et al.* implemented another plug-in for the Visual Studio IDE to record developer interaction [1]. The plug-in is called FeedBag. The plug-in captures more high-level interaction events, namely build events (*e.g.,* starting and finishing a build), commands (*e.g.,* saving a file), documents (*e.g.,* opening, saving and editing documents) and windows (*e.g.,* switching, opening or closing windows).

To be able to track the workflow of developers Minelli and Lanza implemented an extension of the Pharo IDE called *DFlow* [16]. DFlow does not collect low-level events or fine-grained source code modifications but concrete high-level developer actions. Such actions could for example be system navigation, object inspection or source editing [15]. It can then be used to create software visualizations to understand the data recorded during the development sessions. One example for such a view they implemented is a *System Complexity View* containing all entities touched during a development session. This allows them to get an overview of how developer activities are spread across the system. These visualizations can then be used to compare and classify different developer sessions according to the activities performed. Such comparisons could be useful for detecting issues in current IDEs that reduce the productivity and efficiency of developers. DFlow can provide developers with suggestions or ease some of their recurring activities in the IDE.

Selenium [21] is a software-testing framework for web applications. It provides tools for recording and playing back user interaction on a web application. A user can use the Selenium IDE[4] to create Selenium scripts. It is implemented as a browser extension. With it tests can be created and debugged. Tests can both be created by recording them on the user interface and by editing the test script directly. Once a test script is complete it can be replayed on the user interface automatically. Assertions can be included in the script to perform checks on user interface elements. Through this it is possible to create entire testing suites for the user interface of a web application.

To summarize, there are many different approaches to recording developer interaction in use.

[3]Visual Studio IDE - `https://www.visualstudio.com/`

[4]Selenium IDE - `https://www.seleniumhq.org/projects/ide/`

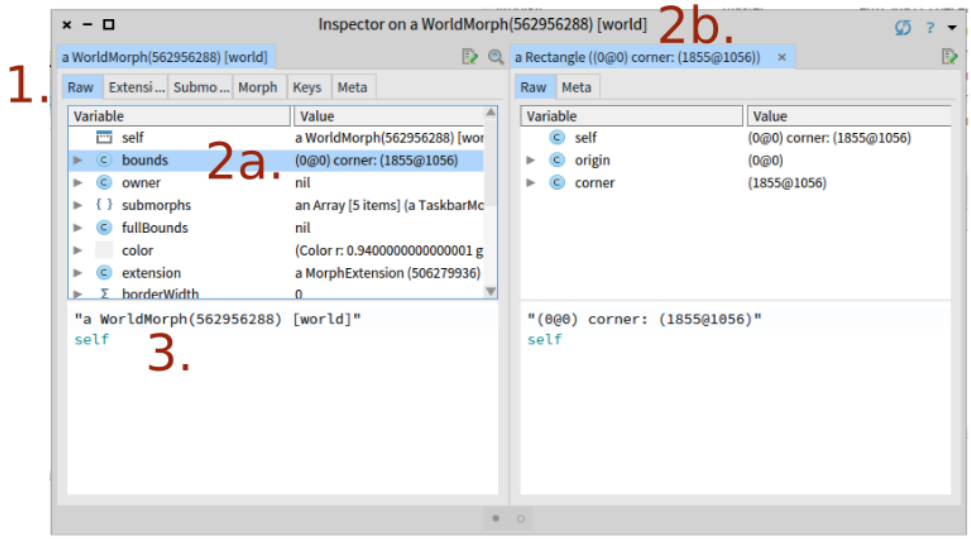

Figure 2.2: The GTInspector open on the World object in Pharo. 1. The raw view of an object listing its properties. 2a. The selected property of the inspection object. 2b. The inspection pane for the object value of the selected property. 3. Area to execute arbitrary code on the inspection object

Some tools prefer to only record low-level events and make assumptions based on that, some others also record low-level events but have an extensible framework for more specific actions. Some tools record high-level events such as opening and closing a document, and some others record all events, low-level and high-level that are provided by the IDE. Other tools are capable of not only recording the interaction but also reproducing it at a later time. However, most of those tools record and collect the interaction for post processing. An exception is Mylyn which updates the list of methods and classes to highlight those relevant for the task. Our proposed approach follows this direction but aims to support even more run-time adaptations based on the recorded interactions.

2.4 GTInspector in Pharo

In this thesis we build on the moldable inspector model and on the *GTInspector*, an object inspector that implements this model. In this section we provide more details about the GTInspector.

2.4.1 Pharo

The GTInspector is an object inspector for the Pharo IDE. Pharo[5] is an open source project that consists of the Pharo language itself and a dynamic programming environment around it. Pharo is Smalltalk based language, where the integrated development environment (IDE), the source code of the system and the execution of user-developed applications are all located in the

[5]Pharo programming environment - `https://pharo.org/`

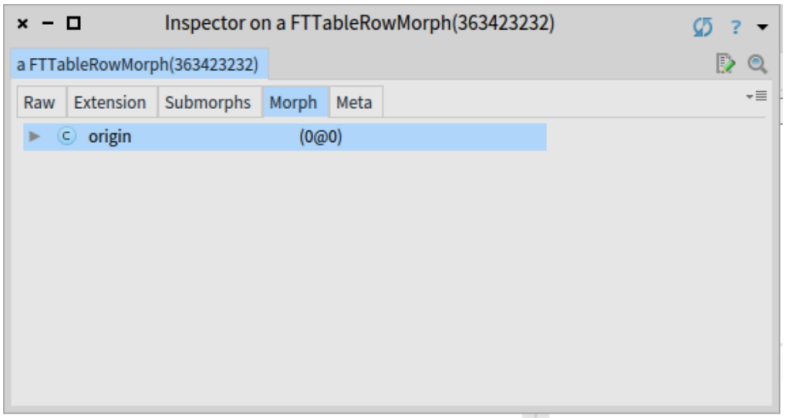

Figure 2.3: The GTInspector open on a graphical element (a row in the inspector). A preview of the element is available directly in the inspector without the surrounding user interface

same system. Through this, Pharo allows direct manipulation of both the source code of a user application and of the IDE and the language runtime.

2.4.2 GTInspector

The GTInspector is the current object inspector from the Pharo IDE. At its core it allows every object to have multiple domain-specific presentations. By default, every object can be inspected in a *Raw* view (see figure 2.2). This view resembles how objects can be inspected in classical debuggers outside of Pharo. It lists all properties of an object and their respective values. By selecting one of those properties the value of that property is inspected in a new pane to the right of the current object. Through this, one can essentially navigate through an object's structure. Furthermore, it is possible to explore which messages an object can respond to and to execute arbitrary code on the object under inspection.

If the Raw view is not very descriptive or inconvenient for looking at an object, a developer may choose to implement custom representations of an object, so called *presentations*. Through this mechanism, the interaction with an object can be made more efficient and more convenient. For example, for a *Color* object it is helpful to actually display the corresponding shade of color and not just display the color values as numbers. For a graphical element a preview of the actual element greatly facilitates the interaction with that object (see Figure 2.3).

2.5 Summary

Given the useful information interaction data can bring, development tools employ a wide range of mechanisms to collect these data, from recording mouse clicks to user-defined events. When looking at object inspectors however they do not directly record and employ interaction data. Instead most of the time they rely on the infrastructure provided by the IDE containing them. Even when they provide support for sharing a session they do not use the recorded sessions to adapt their behaviour at run-time.

3
Problem description

Inspection sessions capture the history of how developers have explored the object graph in order to extract information from the running application. Not explicitly recording inspection sessions in an inspector and using them to improve developer activity can increase the amount of repetitive work developers need to perform in the inspector. In this chapter we illustrate four categories of problems that can occur if object inspectors do not record and leverage inspection sessions. In Chapter 4 we then propose an inspector model that addresses these problems by leveraging recorded interactions.

3.1 Losing inspection sessions

When selecting a stack frame in the debugger, or closing the inspector window, if an inspector does not take explicit steps to save the current inspection session, that session will be lost. If the developer has closed for example the inspector by accident she will have to remember the exact steps from her inspection session and manually apply them to reach the same point as before. The same would happen if she needed to perform the same navigation on the same object, or on an object of a different type.

Figure 3.1 illustrates the problem for an inspector embedded inside a debugger. If the developer explores an object inside the debugger and subsequently selects a different stack frame the state of the inspector will change and reflect the context of the new frame. However, if the developer decides to go back to the original stack frame the inspector will be back in its default state and not show the exploration session the developer had performed previously.

3.2 Manual repetitive explorations

Even if the inspector addresses the aforementioned problem and save the steps of an inspection session, developers still need support for automatically applying those steps on new objects, otherwise they will have to do that manually.

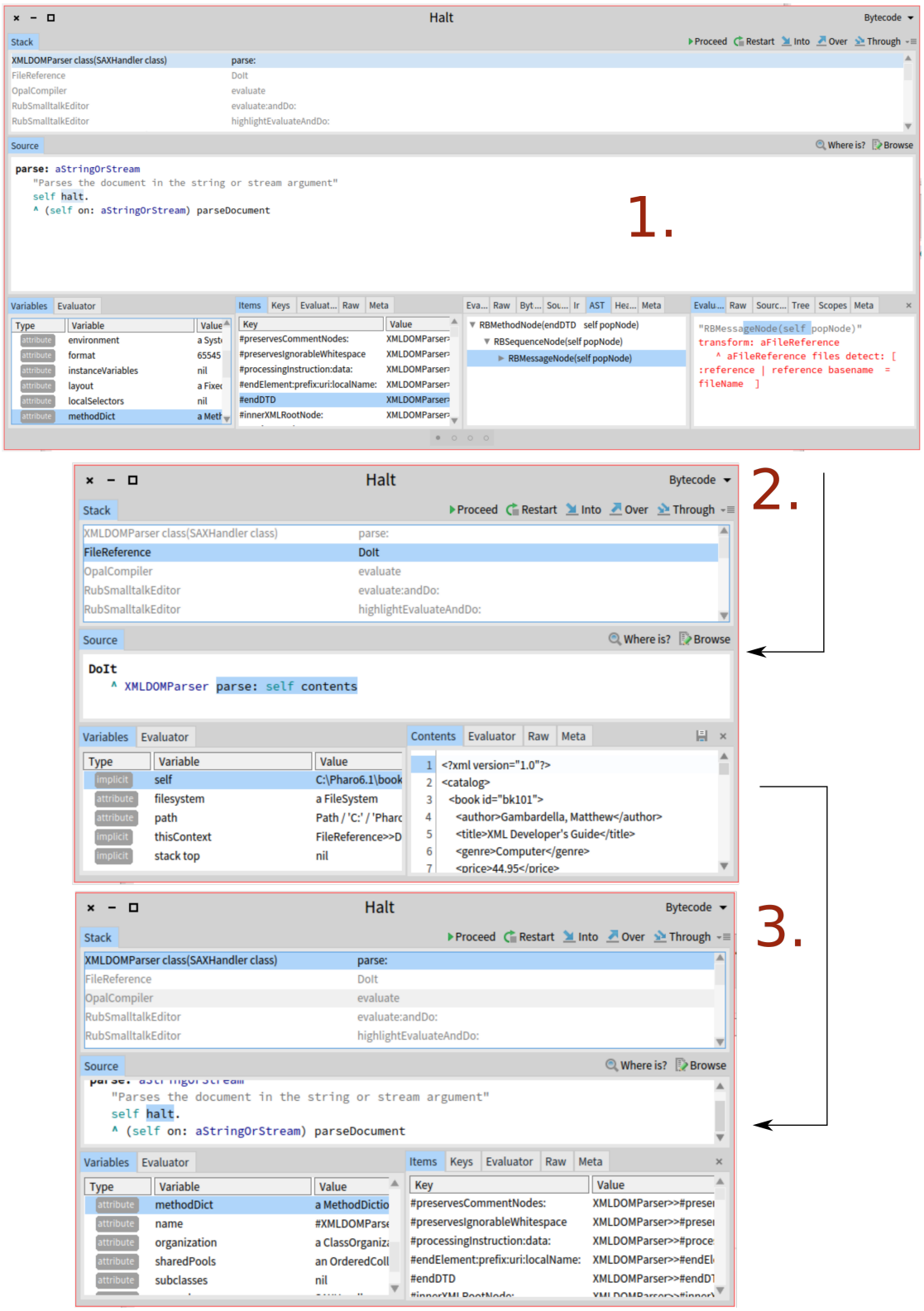

Figure 3.1: Losing inspection sessions when navigating between stack frames. In the first step the developer explores an object tied to the current stack frame *(1)*. The developer then decides to switch to the next stack frame and the inspector will update to show the context of the new frame *(2)*. If the developer then goes back to the original stack frame the inspector will be in its default state *(3)* and not show the previously created inspection session.

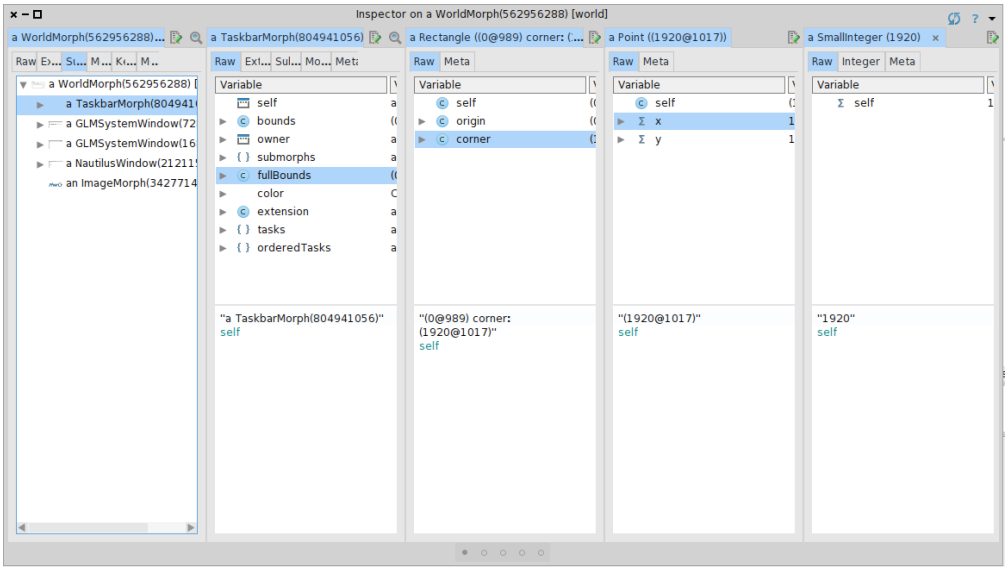

Figure 3.2: The GTInspector open on an object. The user has created multiple inspection steps. In the very first step the user has selected one item out of a list.

The number of repetitive actions can increase significantly especially in the case of manually redoing partial exploration sessions. This happens as an exploration session can have multiple steps. Each step opens a new step with a result computed based on the object selected in the previous step. If the user makes a change in one step (for example selects a different value) all the steps to the right will be removed and the new value will be opened in a new step to the right. Now if the developer needs to reach the same point as before on the new value, she will have to manually redo those steps.

Figure 3.2 shows a longer inspection session starting from a presentation showing a tree of morphs (*i.e.,* graphical components). By diving into the structure of a selected morph in the first view, the developer has created multiple inspection steps. Given that all objects in the initial view are morphs, the user could select another object, and need to redo the same inspection session. However, by clicking on another element of the list, only the second step is updated and the others are removed (Figure 3.3). Likely, this is not the expected behavior for the user, as they have to manually repeat the remaining steps to arrive at the same state as before.

3.3 Manual coding

When using the inspector the developer specifies interactively what operations to perform on the inspected objects in order to reach the last object. However, these operations are embedded in the tool itself. To create a code snippet that reaches the same object, the developer has to manually map the operations from each step.

Figure 3.4 shows an inspection session with a few steps already created. The developer took a graphical object, retrieved its class by executing the code `self class` and selected the property `subclasses` in the *Raw* view. From the resulting list of subclasses the developer selected the first item which is the final result of the inspection session, namely the first subclass of the class of the graphical object.

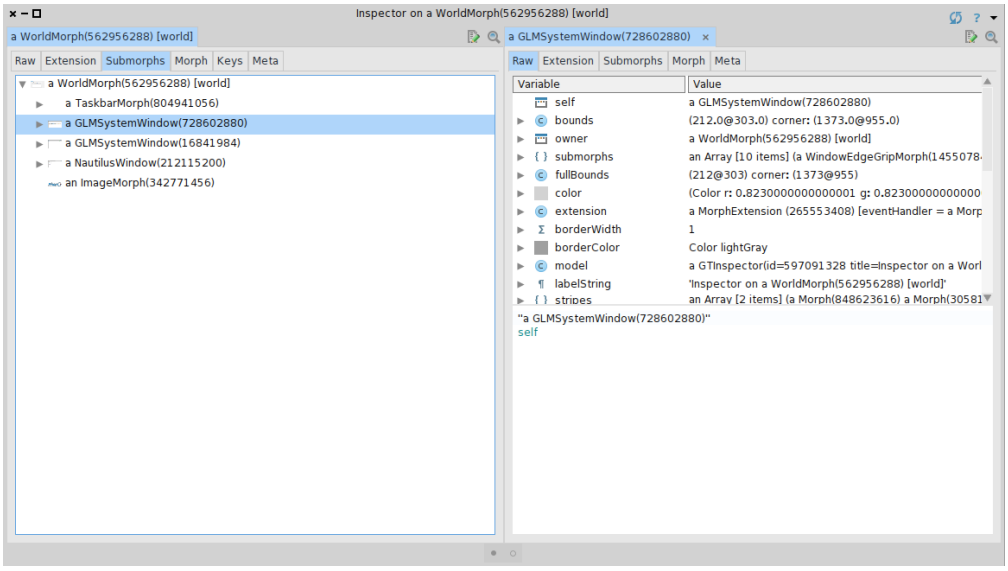

Figure 3.3: The state of the inspector after the user has selected a different item from the initial list (in comparison to figure 3.2). The next pane is updated but the three remaining steps are removed and cannot be recreated.

If a developer wants to recreate the operations from the inspection session in functionally equivalent code she has to do so by hand. She has to figure out which operations in the code correspond to the visual operations in the user interface. Listing 1 shows the code that the developer might write manually. It is very concise and some of the operations performed in the inspector are combined into one statement.

```
1 subclasses := World class subclasses.
2 subclasses at: 1
```

Code 1: The code functionally equivalent to the inspection session shown in figure 3.4 but written manually.

An alternative solution could be for the inspector to be able to generate code from an inspection session. For example, from the previous inspection session the inspector could generate code similar to the one in Listing 2. In that example, the operations are separated into single statements but the code is functionally equivalent. The developer could generate the code directly from the inspector and then refactor it manually.

```
1 object := World.
2 class := object class.
3 subclasses := class subclasses.
4 subclasses at: 1
```

Code 2: The code functionally equivalent to the inspection session shown in figure 3.4 but in the form it might be generated in.

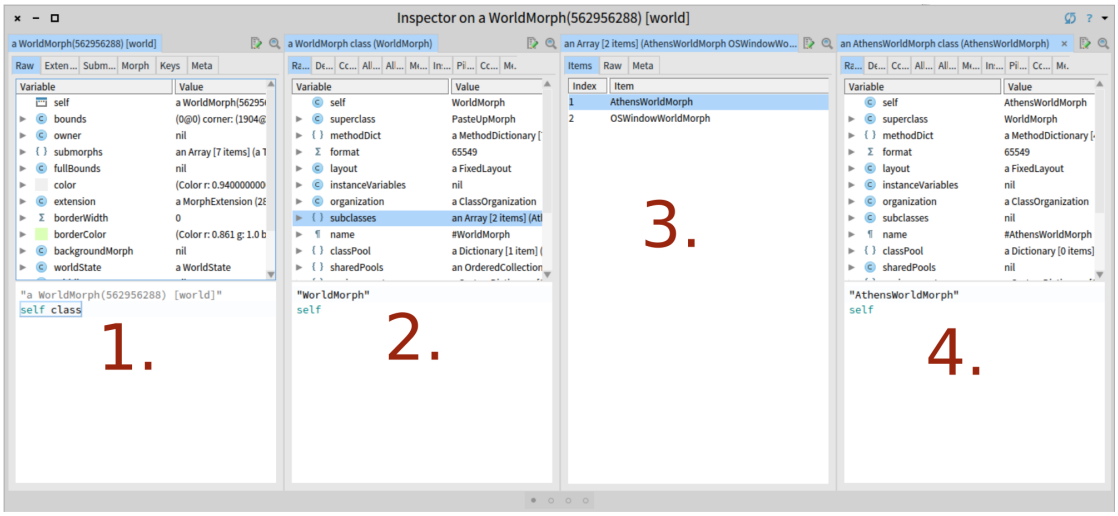

Figure 3.4: An inspection session with multiple steps. The developer is inspecting a graphical element. With the complete inspection session they retrieve the first subclass of the class of the graphical element.

3.4 Sharing an inspection session

Currently, there is no way to share an inspection session with someone else. With a detailed description of the performed interaction a developer has to manually repeat every step to reach the same point as someone else. Being able to share the session is the base for many use cases. For example one might to want save a session and have it replayed by someone else. Another developer might want to record a video of the inspection process or create a tutorial for the inspector with the session as base. In all of these cases it is necessary that an inspection session can be recorded.

3.5 Summary

In this section we looked at four problems that arise in object inspectors that allow developers to work with inspection sessions. These problems can increase the amount of repetitive work done by developers during development. These problems could be addressed in an uniform way by object inspectors that record the user interactions.

4
An improved inspector model

Object inspectors that rely on inspection sessions can have several limitations, as described in Chapter 3. In this chapter we look into how to address those problems through an inspector model that records and takes advantage of developer interactions.

4.1 Extending the moldable inspector model

The moldable inspector model provides an inspector model that allows objects to have multiple domain-specific presentations and groups inspected objects in a linear inspection session. An inspection session is a reflection of the inspected objects and the operations developers applied to these objects.

Every inspection session consists of a number of *inspection steps*. For every operation that the developer performs on an object (*e.g.,* selecting an item from a collection or an instance variable from an arbitrary object) an inspection step is created. An inspection step has an inspection object, which is the object that is under inspection in that specific step. In the inspector, one visual pane is created per each inspection step.

In the current inspector model, the inspection operations are always represented in a linear fashion. If a developer causes the inspector to create a new step, the steps to the right of the new step are lost. To remove this limitation we propose to update the way an inspection session is modelled to use a tree structure instead of just a list. To achieve this we extend the model to allow every inspection step to have as children a list of inspection steps (Figure 4.1). All steps that share a common parent are performed on the same object. The last child step always represents the latest operation that was executed. The inspector could then show a linear sequence of steps or the entire tree structure.

In the current model the action that is executed to create a new pane when the developer interacts with the current pane is hardcoded in the implementation of the graphical widgets. As we want to offer an easy way to record those actions, we extended the model to remove this limitation by adding to every inspection step a *transformation action*. A transformation action represents the operation that was performed within a step in order to create a new step. To capture the different ways in which developers can navigate between steps, there can be many

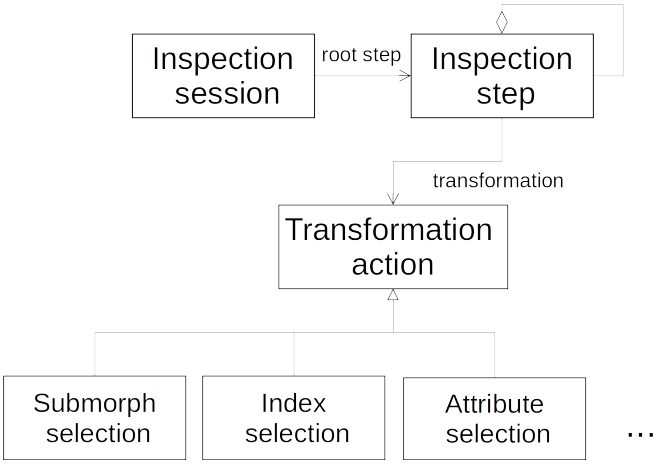

Figure 4.1: The inspector records an inspection session with a root inspection step. Each inspection step contains any number of child steps and a moldable transformation action.

types of transformation actions (Figure 4.1). For example, performing a navigation the *Raw* view would be captured though a `AttributeSelectionAction`; in a view showing a list of objects the navigation action would be an instance of `IndexSelectionAction`. Extension creators can further choose to add their own types of transformation actions to customize the navigation in the inspector.

By starting then with the first object on which the inspector was opened, and iterating through the inspection steps in the desired order while applying the respective transformation actions, the final object and all intermediate objects can be obtained.

4.1.1 Transformation actions

A *transformation action* plays the role of a function that takes one argument, the selected object, and returns one other value, the transformed object. The transformation encapsulates the operation that is performed on the selected object in the inspector to compute the object that should be inspected next.

For instance, if a developer clicks on an entry in a dictionary the corresponding transformation action could retrieve the value with the given key from the actual dictionary. Another example is an action that retrieves the value of an instance variable with a given name from an object. This can be used to model selection actions in the *Raw* view.

4.2 Moldable interactions

In order to address the issues described in Chapter 3 we need to ability to record inspection sessions. Given that the inspector is a tool designed to be extensible we need a recording mechanism that follows the same principle.

To support this we propose *moldable interactions*, an extensible approach for recording and replaying events. This is based on the idea of allowing graphical widgets to decide how a user interaction should be recorded. Always, when a user interacts with a widget that widget has to decide what event will be recorded. By default, the widget can return a generic action that identifies the widget on which the action was performed based on its location in the tree of

widgets. However, widgets can also decide to return custom events. In the case of the inspector, widgets displaying a presentation directly return the transformation action that is executed to generate the object to be inspected next. Using this approach, transformation actions are used both within the inspector model to capture user actions, and are recorded as interaction events.

4.2.1 Generically identifying widgets

An important part of this approach is the ability to have a generic event that identifies the widget with which the developer interacted. This is needed to allow the recording mechanism to be generic, and to not require developers to provide a transformation action for every widget. Developers should provide custom actions only for interesting widgets. For the others a generic event should be recorded.

To record an interaction with a widget the right widget has to be identified. Just identifying a widget by its pixel coordinates has many disadvantages. Whenever the user interface of a tool is rescaled or otherwise changed, the position of an element at the time of the recording can become invalid. An alternative is to rely on the fact that most graphics frameworks have a notion of *component hierarchy*. Typically, one widget is the root of that hierarchy and each widget has zero or more sub-widgets.

Hence, instead of identifying a widget by its visual position we identify it by its position within the component hierarchy. That is, we can identify a widget by collecting first its list of parents until we reach the root component of the graphical hierarchy. Then for each widget in this list that has a parent we record its position within the list of children of its parent (*i.e.,* similar to the *n-th child* selector in CSS). This gives a path that can be used to generically locate a widget within a graphical structure.

Figure 4.2 shows an example of a possible path using the position within the parent as an identification measure. The path consists of four path actions.

4.2.2 Customizing the path for locating widgets

Always locating a widget by its position within the list of children of its parent covers a lot of cases and can be used as a sensible default. However, in the context of the inspector this introduces a limitation. For example, for a user interface containing a tabbed widget it makes more sense to select the desired tab by name, not by its position. With that approach the correct tab can be selected even when the order or the number of tabs is changed. This is important for the inspector as the list of presentations attached to an object can evolve. Saving paths just using the index of the widget means that older sessions will not be restored correctly.

To address this, instead of always using the position within the list of children to identify a widget, the widget can decide how it should be identified by providing a custom *path action*. A path action, is an action that returns a serializable way to identify a sub-widget within a parent widget. By asking all ancestors of a widget for their path actions a customized path through the component hierarchy is retrieved and can be persisted. This path can then be used to identify widgets when replaying an action on the user interface.

By default, the position of a widget within the list of children is used. Widgets can then customize this. For example, Figure 4.3 shows an illustration of a possible path. It consists of four path actions. First, the pane at position 1 is selected. Since the figure shows a newly opened inspector only one pane is available. Afterwards, the tab named *Submorphs* is selected. Here the tab is selected by name instead of by position which is reproducible even if the order of the tabs changes. Third, the tab's first child is selected. This is the default path action if no custom action

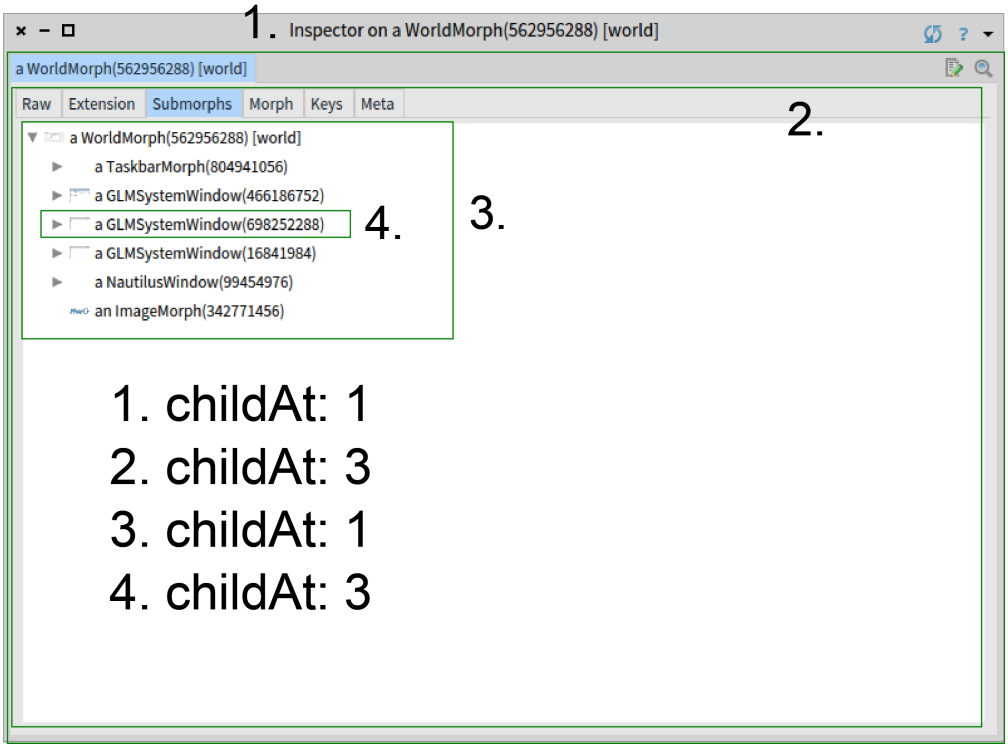

Figure 4.2: Illustration of a path identifying a widget. The path consists of 4 path actions. For every action a specific child of a widget is selected. The index of the child to be selected is determined by the index in the *childAt:* path action.

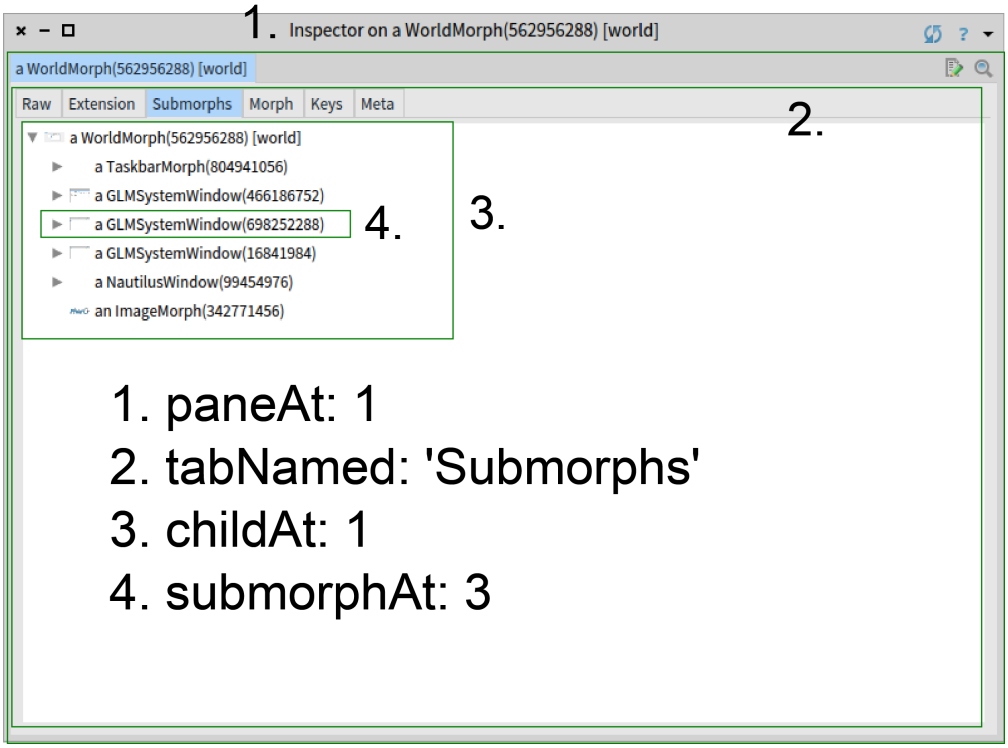

Figure 4.3: Illustration of a path identifying a widget. The path consists of 4 path actions. The second action is a custom action and selects a tab by name. Selecting a tab by name is more robust since changes in the ordering of tabs don't matter. The third action selects the first child. This is the default action used when no custom action is defined.

is defined. Finally, the submorph at position 3 is selected with a custom path action responsible for selecting a submorph by position within its parent's children.

4.3 Improving the inspection process

Next we look at how with this new inspector model and solution for recording developer interactions it is possible to address the four problems described in Chapter 3.

4.3.1 Losing inspection sessions

Currently in Pharo, if the inspector is closed its state cannot be restored. The user has to open a new instance on the same initial object and repeat all previous interaction.

With the model described in Section 4.1 it is possible to replay a saved session: an inspector can be opened on the initial object and the inspection session can be recreated by replaying for each inspection step the latest transformation action.

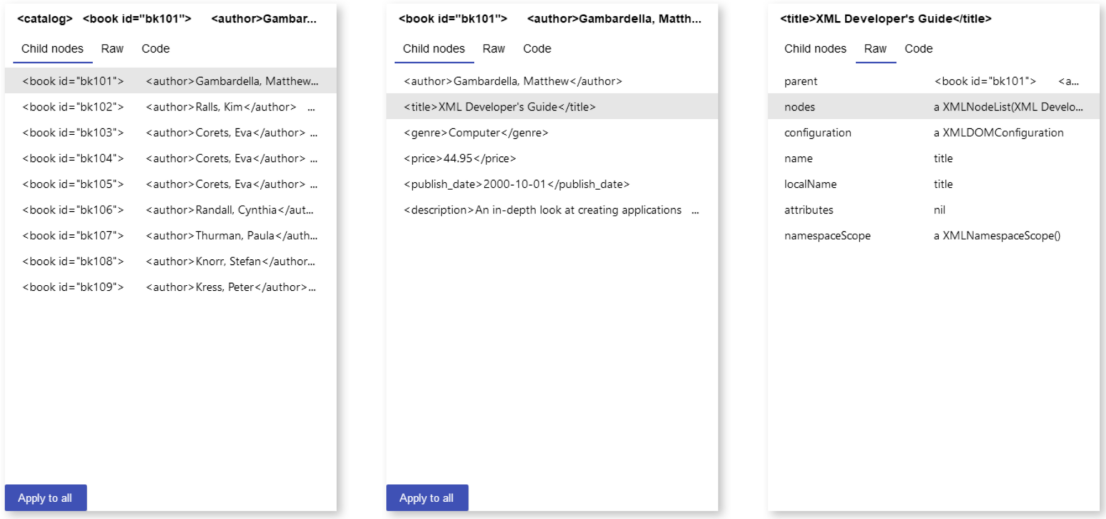

Figure 4.4: The inspector opened on an XML document containing meta data about books. The developer selected one of the books and then its title.

4.3.2 Manual repetitive explorations

As described in Section 3.2 partial sessions can be lost if the user decides to interact with the inspector not in the last inspection step but a previous one. In some cases it would be beneficial if the remaining steps after the changed inspection step could be reapplied automatically.

With the proposed inspector model this can be addressed as the model stores all the partial paths in the exploration using a tree. When the user makes a change in an inspection step that is not the last in the session, the steps following that step can be extracted from the session. These steps are then added as child steps to the new step created by the user. If the type of the new object is the same as the type of the previous object then the transformation actions of the previous steps can be automatically applied.

Figures 4.4, 4.5 and 4.6 illustrate an example for reproducing parts of an inspection session. In Figure 4.4 an XML document containing meta data about books was parsed and is being inspected. The developer has selected one of the books and then selected the title of that book. After that the developer switched to another book. The result is visible in Figure 4.5. The action of selecting the title from the book was reproduced by the inspector. The developer does not have to select the title manually anymore. Figure 4.6 shows the state of the inspector after the developer clicked the *Apply to all* button in the book list. The inspector applied the action of selecting the title from a book to all elements of the list. This results in a list of book titles. The developer can then choose to inspect a title as usual by selecting it from the list.

4.3.3 Manual coding

An inspection session contains all the necessary information for extracting code that is able to transform the inspection object in the same way as the user did. Currently, that code has to be extracted by hand. However, by using the inspection steps from the inspection session this code can be generated automatically as transformation actions capture the intent of the user.

To achieve this, the source code of the transformation functions in the transformation actions

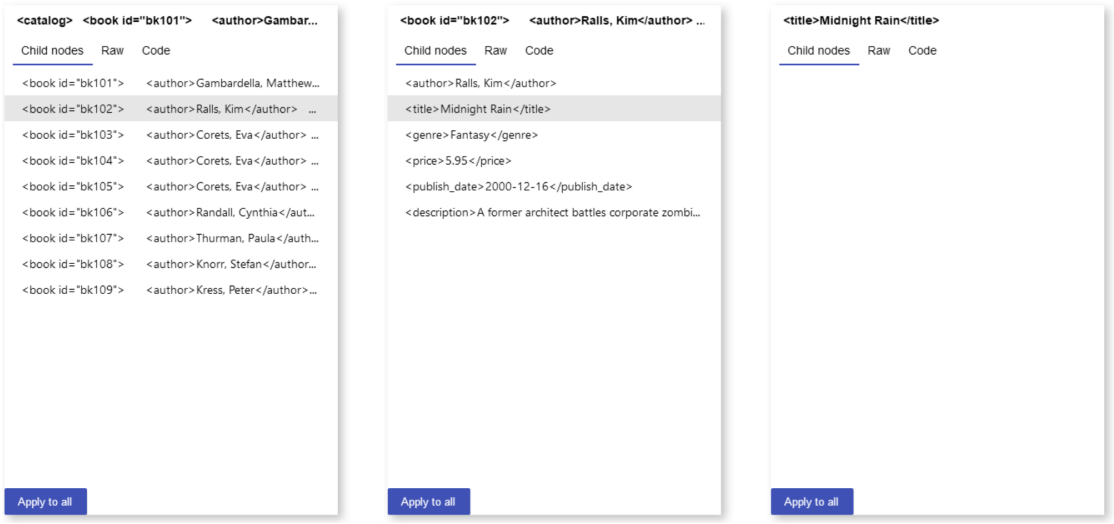

Figure 4.5: The state of the inspector after the developer selected a different book to the one from Figure 4.4. The inspector reproduced the action of selecting the title of the book.

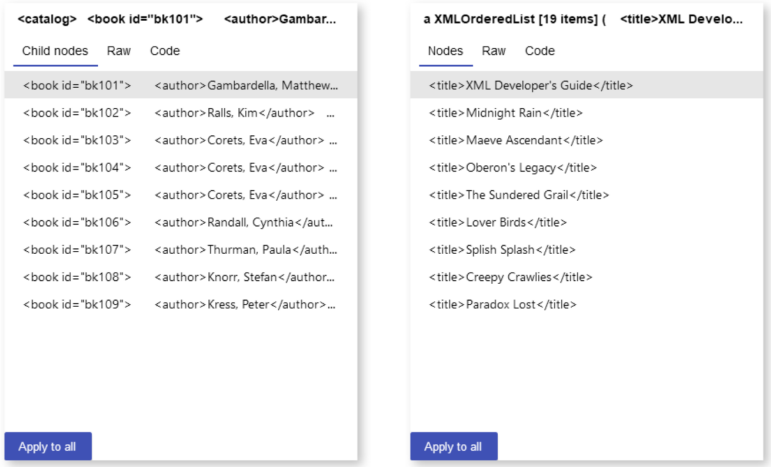

Figure 4.6: The state of the inspector after the developer applied the interaction to all books visible in Figure 4.4. This resulted in a list of book titles which can be inspected further.

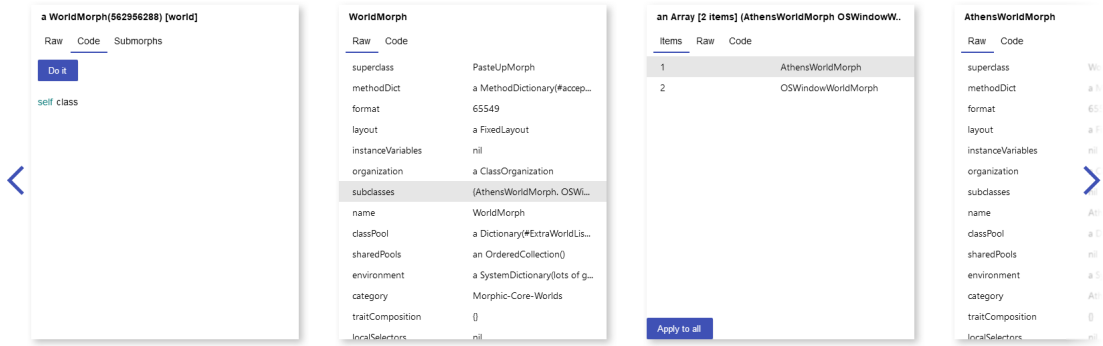

Figure 4.7: A screenshot of an inspection session with three inspection steps

can be combined in a snippet of code. This is done by first assigning the inspection object to a temporary variable. For every inspection step the transformation function is added to the source code.

The abstract syntax tree (AST) of the transformation functions is modified to reduce the number of manual edits the user will have to do on the code. The result of each function is stored in the temporary variable and the argument to the function is renamed to match the name of the temporary variable.

Section 3.3 illustrates an example of a session from which we can generate code. In this section the developer is inspecting a widget (in this case a `Morph`). In the first step the class of the morph is retrieved. After that the subclasses of the resulting class are retrieved by selecting the instance variable named `subclasses`. Finally, the first item of the resulting collection is selected. Figure 4.7 shows the inspection session in the inspector and Listing 3 shows the generated code.

```
1 tempVar := (tempVar class compiler
2    source: 'self class';
3    receiver: tempVar) evaluate.
4 tempVar := tempVar instVarNamed: #subclasses.
5 tempVar := tempVar at: 1.
6 tempVar
```

Code 3: The code generated from the inspection session shown in figure 4.7.

4.3.4 Sharing an inspection session

With the extended inspector model it is possible to persist and share an inspection session. For example, an inspection session could be serialized to a file and then sent to another developer. Reopening a shared inspection session is possible by deserializing it and then restoring the inspector's state by replaying the transformation actions from the session.

When replaying a session the inspector can further choose to replay the actions instantly or with a delay. By introducing a delay between individual steps in the inspection process they become visible. It could also be possible to only replay the next action once the user clicks a specific button. This makes a demonstration from within the inspector itself possible.

4.4 Summary

Recorded developer interactions can provide important information about the behavior of developer. In this chapter we looked at how they can be embedded directly in an inspector model and used to reduce repetitive actions during the inspection process.

5 Implementation

In this chapter we describe in more details several aspects about the solution presented in Chapter 4. The source code of the implementation can be found on github[1]. This implementation is based on *Bloc*[2], a low-level user interface framework for Pharo.

5.1 Recording actions

During the recording of an inspection session two types of actions are used, path actions and transformation actions.

5.1.1 Path actions

Path actions only provide the functionality to select a child widget within a parent. They are used to identify a graphical element within the component hierarchy. The most basic path action is the `BlChildSelectionAction`. It selects the n-th child of an element. It is only related to the graphical framework and not to the domain of the inspector. As mentioned in Section 4.2.1 custom path actions can be added to extend the functionality. The inspector implementation itself uses custom path actions for selecting inspector panes and presentation tabs.

5.1.2 Transformation actions

Transformation actions capture an explicit navigation from one object to another. They define a method `transform:` which corresponds to the transformation function described in Section 4.1.1. This method assumes that the action can be applied to the provided argument. Before applying the transformation function to an object the inspector checks whether the action is actually applicable. For that purpose the action provides a `canTransform:` method. This method can check whether the operations that are provided in `transform:` are valid for the provided argument.

[1] Implementation source code - `https://github.com/mariokaufmann/inspector-replay`
[2] Bloc - `https://github.com/pharo-graphics/Bloc`

Listing 4 shows an example of a transformation function. It is taken from the transformation action `IndexSelectionAction`. This action selects an element from a collection by index. Its `canTransform:` method (Listing 5) checks whether the object is sequenceable and then furthermore ensures that the desired index is within the bounds of the collection.

Another example is the `CodeAction`. It takes some arbitrary code and then executes that on the provided argument. Its transformation function (Listing 6) compiles the contained source code on the class of the given object and evaluates it on that object. Since the code can be applied to any object, the action's `canTransform:` method always returns true.

```
1 IndexSelectionAction>>#transform: anOrderedCollection
2   ^ anOrderedCollection at: index
```

Code 4: The transformation function for an action that selects an element from a collection by index. The index is contained in an instance variable in the action.

```
1 IndexSelectionAction>>#canTransform: anObject
2   (anObject respondsTo: #isSequenceable) and:
3     [ anObject isSequenceable and: [ ^ anObject size >= index ] ].
4   ^ false
```

Code 5: The `canTransform:` method for an action that selects an element from a collection by index. It checks whether the collection is sequenceable and ensures that the desired index is within the bounds of the collection.

```
1 CodeAction>>#transform: anObject
2   ^ (anObject class compiler source: code; receiver: anObject) evaluate
```

Code 6: The transformation function for an action that can execute arbitrary code on an object. The source code to be executed is contained in an instance variable in the action.

5.2 Custom path actions

As mentioned before, it is possible for widgets to provide a custom path action that will then be used to identify the element within its parent element. There are two mechanisms for providing custom path actions that a widget can use.

In the first solution, widgets (subclasses of `BlElement` in Bloc) have a `userData` dictionary in which users of the widget can save custom data. The inspector uses this to store a custom action, if the user desires to do that. For this the inspector extends `BlElement` with the method `BlElement>>#recordingAction:` with which the custom action can be set. Sending the message `recordingAction` to a `BlElement` will return the action. An element that has no custom action will return the default path action which identifies the element as the n-th child of its parent (`BlChildSelectionAction`).

The second solution consists in overriding the `BlElement>>#recordingAction` method. Instead of returning the custom action from the user data dictionary, the method can return any other

recording action. This is a more convenient approach for custom widgets that subclass `BlElement`. Listing 7 shows how the custom widget for displaying a tab overrides the method to model a tab selection.

```
1 GT2TabLabelElement>>#recordingAction
2    ^ GT2TabSelectionAction withTabName: self tabName
```

Code 7: Overriding the `recordingAction` method to provide a custom recording action for selecting a tab.

5.3 Code generation

With this inspector model it is possible to generate code from an inspection session. This is a process that consists of multiple steps. First, the code generator collects all inspection steps and gathers their transformation actions. Then the generator extracts the abstract syntax tree (AST) of the action's transformation functions; this is a straightforward process in Pharo as the AST can be obtained by sending the `ast` message directly to a `CompiledMethod` object. Before combining the collected ASTs into a single one, a transformation is applied to ensure that transformation functions can be chained together into a single snippet. The current implementation achieves this by assigning the result of a step to the variable `tempVar` and passing this variable as input to the next step. An example can be see in Listing 3. This is only a temporary solution to explore how code generation can be supported by the new inspector model.

5.3.1 Replacing parameter name

The code generator gathers the abstract syntax trees from all transformation actions. Another visitor, an instance of `GT2InspectionCodeGeneratorVisitor` is used to perform a second modification of the ASTs. The visitor is a subclass of `RBProgramNodeVisitor`.
The visitor visits the argument node of the method. The name of the parameter is changed to a temporary variable name provided by the code generator. Additionally, all usages of the parameter are also replaced with usages of the temporary variable. This facilitates passing the results from one transformation action to the next in the final generated code.

5.4 Replaying UI interaction

With the help of Bloc, UI actions can be simulated on the user interface. Bloc offers the functionality to simulate user interactions to which the user interface will react as if the user herself had performed the interaction. For example, with `BlSpace>>#simulateClickOn:` one can simulate a click on an arbitrary graphical element.

6 Conclusion and future work

In this chapter we conclude the thesis and explore possibilities for future work.

6.1 Conclusion

The goal of this work was to build an inspector model that can directly record and replay user interactions, and use those interactions to improve the development experience of developers.

Simple approaches, like storing pixel coordinates or serializing the inspector to a file, are not sufficiently flexible and robust to solve the problem. Instead we rely on explicitly modeling user actions and capturing them in the inspection session and show how this can address several issues that developers can encounter while using the inspector.

6.2 Future work

There are numerous open points where future work can begin.

6.2.1 Full inspector implementation

To support the inspector model presented in this thesis a new user interface was created for the inspector from scratch. This implementation is still a prototype and does not support all the features provided by the current inspector from Pharo. For example, only several types of custom presentations are present, there are no context menus and toolbars, presentations cannot be filtered, *etc.* To make the new inspector a viable replacement for the current one it has to be brought up to par feature-wise with the current inspector.

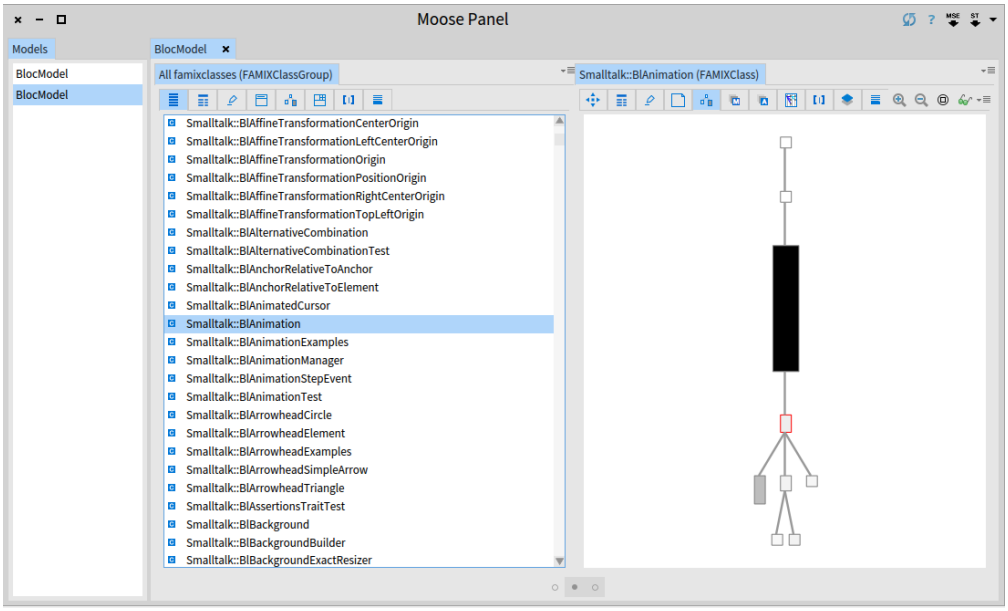

Figure 6.1: A screenshot of an inspection session using Moose. Custom presentations visualize the source code that is under inspection

6.2.2 Updating custom presentations

One powerful aspect of the current inspector are the custom presentations that have been created for all kinds of projects. Figure 6.1 shows an example for presentations used by Moose[1]. Recreating the presentations poses two main challenges.

First, the new inspector implementation should provide the same API for creating custom presentations. However, the new inspector is implemented in a new graphical framework (namely Bloc). Currently for moving a few types of presentations, the API could be preserved. However, this might not be the case for all existing types of presentations. This could require changes in the code defining the presentations.

In addition, presentations have to define the transformation actions that will be used during navigation. These transformation actions are then used to build the inspection session. If custom presentations don't specify any transformation actions it will not be possible to properly use them in recording and reproducing interaction. This is a significant difference to how the custom presentations are currently defined.

6.2.3 Create custom path actions for UI recording

To enable the inspector to record UI actions all user interface elements have to provide path actions. The inspector defines a default action for `BlElement`. However, to make use of the moldable recording infrastructure custom actions should be added. Only then can a tool be equipped with proper UI recording. More effort is needed to properly create path actions for other widgets

[1]Moose - platform for software and data analysis `http://www.moosetechnology.org/`

6.2.4 Improve code generation

The extraction of code from an inspection session can be significantly improved. Currently, code generation works best if the transformation functions of the inspection steps are short and don't rely on other methods. In future work code generation for transformation functions sending a message to another object can be implemented. Furthermore, there is currently no way to avoid that temporary variable declarations of multiple transformation functions clash if they have the same names. At the moment, an assignment statement is generated for each transformation action. This is also the case even for an action that is extremely short. It could be possible to automatically pull together multiple transformation functions into one statement in the final code.

6.2.5 UI recording infrastructure

The inspection session model with its inspection steps is specific to the inspector. However, recording infrastructure is not, and is built directly into the UI framework. Any tool using the same UI framework could potentially use it. Future work could explore how the recording infrastructure can be transferred to other tools and how it would need to be extended.

Bibliography

[1] S. Amann, S. Proksch, S. Nadi, and M. Mezini. A study of Visual Studio usage in practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 124–134, March 2016.

[2] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles: A working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2503–2512, New York, NY, USA, 2010. ACM.

[3] Alistair E. R. Campbell, Geoffrey L. Catto, and Eric E. Hansen. Language-independent interactive data visualization. *SIGCSE Bull.*, 35(1):215–219, January 2003.

[4] Andrei Chiş, Tudor Gîrba, Oscar Nierstrasz, and Aliaksei Syrel. The Moldable Inspector. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2015, pages 44–60, New York, NY, USA, 2015. ACM.

[5] James H. Cross, II, T. Dean Hendrix, David A. Umphress, Larry A. Barowski, Jhilmil Jain, and Lacey N. Montgomery. Robust generation of dynamic data structure visualizations with multiple interaction approaches. *Trans. Comput. Educ.*, 9(2):13:1–13:32, June 2009.

[6] Kostadin Damevski, David C. Shepherd, Johannes Schneider, and Lori Pollock. Mining sequences of developer interactions in Visual Studio for usage smells. *IEEE Trans. Softw. Eng.*, 43(4):359–371, April 2017.

[7] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. Debugger Canvas: Industrial experience with the Code Bubbles paradigm. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1064–1073, June 2012.

[8] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 385–394, New York, NY, USA, 2010. ACM.

[9] Zhongxian Gu, Drew Schleck, Earl T. Barr, and Zhendong Su. Capturing and exploiting IDE interactions. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 83–94, New York, NY, USA, 2014. ACM.

[10] Mik Kersten and Gail C. Murphy. Mylar: A degree-of-interest model for IDEs. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 159–168, New York, NY, USA, 2005. ACM.

[11] Takashi Kobayashi, Nozomu Kato, and Kiyoshi Agusa. Interaction histories mining for software change guide. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, RSSE '12, pages 73–77, Piscataway, NJ, USA, 2012. IEEE Press.

[12] R. Minelli and M. Lanza. Visualizing the workflow of developers. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4, September 2013.

[13] Roberto Minelli. *Interaction-Aware Development Environments*. PhD thesis, Università della Svizzera italiana (USI), 11 2017.

[14] Roberto Minelli, Andrea Mocci and, and Michele Lanza. I know what you did last summer: An investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 25–35, Piscataway, NJ, USA, 2015. IEEE Press.

[15] Roberto Minelli and Michele Lanza. DFlow - a platform to profile developers. `http://esug.org/data/ESUG2013/5-Fri/02-DFlow-ESUG2013.pdf`, 2013.

[16] Roberto Minelli and Michele Lanza. DFlow - towards the understanding of the workflow of developers. In *SATToSE 2013 (6^{th} Seminar Series on Advanced Techniques & Tools for Software Evolution)*, 2013.

[17] Roberto Minelli, Andrea Mocci, Romain Robbes, and Michele Lanza. Taming the IDE with fine-grained interaction data. In *Proceedings of ICPC 2016 (24^{th} International Conference on Program Comprehension)*, 2016.

[18] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, July 2006.

[19] Eclipse mylyn - a task-focused interface for Eclipse. `https://projects.eclipse.org/projects/mylyn`.

[20] Sebastian Proksch, Sarah Nadi, Sven Amann, and Mira Mezini. Enriching in-IDE process information with fine-grained source code history. In *International Conference on Software Analysis, Evolution, and Reengineering*, 2017.

[21] Selenium - web browser automation. `https://www.seleniumhq.org/`.

[22] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Roshanak Zilouchian Moghaddam, and Ralph E. Johnson. The need for richer refactoring usage data. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, pages 31–38, New York, NY, USA, 2011. ACM.

[23] YoungSeok Yoon and Brad A. Myers. Capturing and analyzing low-level events from the code editor. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, pages 25–30, New York, NY, USA, 2011. ACM.