MASH: a Tool For End-User Plug-In Composition

Leonaro Mariani, Fabrizio Pastore
Department of Informatics Systems and Communication
University of Milano - Bicocca
Milano, Italy
{*mariani,pastore*}*@disco.unimib.it*

***Abstract*—Most of the modern Integrated Development Environments are developed with plug-in based architectures that can be extended with additional functionalities and plug-ins, according to user needs. However, extending an IDE is still a possibility restricted to developers with deep knowledge about the specific development environment and its architecture.**

In this paper we present MASH, a tool that eases the programming of Integrated Development Environments. The tool supports the definition of workflows that can be quickly designed to integrate functionalities offered by multiple plug-ins, without the need of knowing anything about the internal architecture of the IDE. Workflows can be easily reshaped every time an analysis must be modified, without the need of producing Java code and deploying components in the IDE.

Early results suggest that this approach can effectively facilitate programming of IDEs.

***Keywords*-plug-in composition, end-user programming**

I. Introduction

Nowadays Integrated Development Environments (IDE) are not only tools that support development and programming activities, but are also environments that can be programmed and customized to satisfy at best the user needs. For example, the Eclipse[1] users who regularly need to run an analysis that is not natively available in the IDE can implement a new plug-in that provides the required analysis.

If modern IDEs empower users with the capability of designing, implementing and customizing analyses through plug-ins, they also require relevant expertise for being tailored and extended. In fact users can implement new plug-ins if they know the architecture of the IDE, the APIs of all the plug-ins they intend to use, and the deployment process supported by the IDE. In contrast with these technical requirements, it is worth noticing that many interesting and useful automated analyses can be obtained as an integration of existing plug-ins, with little programming effort. For example, statistical fault localization (e.g., the Tarantula technique [1]) can be implemented by integrating JUnit[2], a plug-in for unit testing, with Eclemma[3], a plug-in for code coverage, and computing some statistics about the executed lines of code. Unfortunately, due to the cost of the design of the new plug-ins, these opportunities are seldom exploited and developers do not augment their IDE with the analyses they would like to run.

We believe that end-user programming of IDEs should be strongly simplified [2]. In particular, end-users should be able to rapidly compose the functionalities available in an IDE and design the glue code that might be necessary to produce new features, without intensive programming effort.

This paper presents MASH, a tool that enables effective end-user programming of IDEs. The key principle in MASH is that end-users should be able to visually compose plug-ins and plug-in functionalities into workflows that can be flexibly modified, executed, and reused. On a technical viewpoint MASH relies on three key components: the MASH Framework, which enables the use of Task-Based plug-ins (TB-Plug-ins) as part of IDEs; the TB-Plug-ins which provide functionalities as tasks that can be integrated and executed within the workflows; and a Workflow engine, which executes the workflows designed by end-users. In addition to tasks natively provided by TB-plug-ins, users can manually generate new tasks, obtained as an integration of the functionalities offered by multiple plug-ins, through a recording function provided by MASH. The recording function captures user activities and automatically synthesizes the observed activity into a re-executable parametric task. In this way, users can produce tasks without the need of invoking any method defined in any API.

We implemented MASH as an Eclipse plug-in that augments Eclipse with plug-in composition capabilities, but the same components described in this paper can be developed for other IDEs. The MASH approach can be even replicated in applications different from IDEs. For instance, Rich Client Applications obtained from Eclipse can be augmented with MASH to provide the end-users with an easy way to visually customize and program the application. Our early experience with the MASH framework suggests that it can concretely facilitate the customization of IDEs [3].

The paper is organized as follow. Section II presents our MASH implementation for Eclipse. Section III presents a sample case. Section IV discusses related work. Section V provides final remarks.

[1]http://www.eclipse.org
[2]http://www.junit.org
[3]http://www.eclemma.org

II. MASH

The MASH framework supports rapid design of custom analyses that can be executed and maintained overtime. Custom analyses are obtained through visual workflows that combine the tasks provided by the installed TB-plug-ins. Figure 1 shows the architecture of the MASH framework. The top area of Figure 1 shows the structure of plug-ins and TB-plug-ins installed in the IDE. The bottom area of Figure 1 shows the components in the framework.

Regular plug-ins export their functionalities through API and widgets visualized in the GUI of the IDE. TB-plug-ins export their functionalities through tasks and workflows. A task is an executable unit of work with a name, inputs, outputs and a configuration. A workflow is a complex flow of operations that includes one or more tasks. Tasks are directly executed by plug-ins. Workflows are executed by a workflow engine. Figure 1 shows that regular plug-ins can be reified into TB-plug-in using proxy TB-plug-ins that export the API of the regular plug-ins as tasks, which forward calls to the original API when executed. Here we do not describe how to implement a TB-plug-in, but we focus on the main characteristics of the entire framework. The interested reader can find additional information about the design of TB-plug-ins in [3].

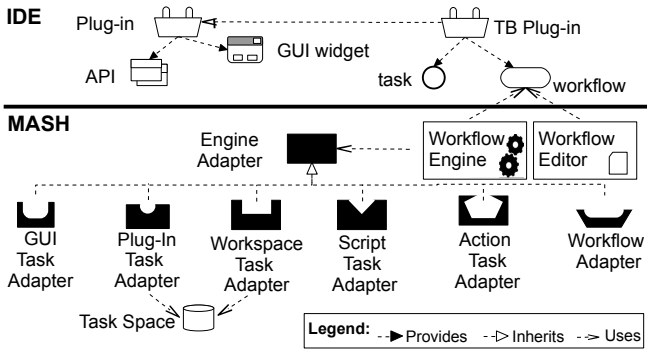

Figure 1. The MASH architecture

Tasks can have multiple inputs and outputs. Plug-in developers assign names to input and output variables. Variable names must be unique in the task where they are defined. IDE users can visually connect task outputs to inputs or assign constant values to task inputs. At runtime, MASH assigns actual values to task inputs according to the workflow specification.

Task behaviors may be tuned through configuration parameters. For instance, a task that exports code coverage information may require a specification of the output format, which could be provided as a configuration parameter. In MASH, each task has a default configuration that can be changed by IDE users. Developers of TB-plug-ins can define panels that allow IDE users to change the task configuration. MASH supports the visualization of these panels.

When a workflow includes multiple instances of the same task, a same task configuration can be shared among these instances. Sharing is obtained by linking the visual element that represents a configuration to multiple task instances.

Tasks often produce outputs that are not only useful to other tasks but are also meaningful for IDE users. Developers of TB-plug-ins can create ad-hoc editors for visualizing and modifying task outputs. IDE users can use these editors to edit task outputs and re-execute workflow segments to run "what if" analysis.

IDEs, like Eclipse, already provide a number of executable user actions, that is an implementation of the reaction executed by the system when a user performs an action on the GUI, such as clicking on a menu. MASH automatically recognizes the user actions implemented in the IDE and makes them available as tasks that can be used in workflows (we call these tasks *action tasks*).

In addition to executing the tasks and workflows distributed with TB-plug-ins, IDE users can design new workflows and tasks. IDE users design new workflows by using a workflow editor integrated in the IDE, while create new tasks by either capturing GUI actions or writing Java tasks.

Since the functionalities provided by regular plug-ins are accessible through GUI widgets, the easiest and fastest way to integrate the functionalities offered by multiple regular plug-ins is to manually produce a sample execution of the intended task, by using the widgets associated with the regular plug-ins, and let the framework to replay it. Our Eclipse-based implementation of MASH exploits the TPTP framework[4] to provide this feature. In particular, MASH can transform a sequence of actions executed by a user on the IDE into a task, we call these tasks *GUI tasks*. A GUI task can be loaded into a workflow and executed. Since a GUI task represents a specific case that often needs to be generalized, MASH automatically extracts the parameters entered in the executed sequence of actions and produces a parametric task, where these parameters can be changed from the workflow editor. For instance, if the sequence includes right-clicking on a file, the name of the selected file is automatically transformed into a parameter of the task generated by MASH.

When users require functionalities not provided by any tool, such as computing the diff between the coverage produced by the execution of two test cases, they can implement a task by writing a simple Java class, which is stored in the project workspace. We call these classes *workspace tasks*. Using workspace tasks, users can concentrate on the implementation of the algorithms, for instance the Tarantula algorithm, without worrying about the integration with Eclipse. MASH automatically populates the workflow editor with the available workspace tasks. To quickly produce simple glue code, users can also implement new tasks as

[4]http://www.eclipse.org/tptp/

scripts written directly in the workflow editor, we call these tasks *script tasks*.

The bottom area of Figure 1 shows the main components of the MASH framework that support the execution of tasks and workflows. The execution and design of workflows is supported by a workflow engine. Our Eclipse-based implementation of MASH uses JOpera [4], which is a workflow engine distributed as an Eclipse plug-in. Our MASH implementation automatically loads every task provided by the TB-plug-ins installed in Eclipse and makes them available within JOpera. In addition MASH manages the integration between the many kinds of tasks that can be executed and the specific workflow engine that is used by a hierarchy of adapters. At the present time, MASH supports six adapters. Every adapter extends the *Engine Adapter*, which implements an abstract adapter. The six specific adapters correspond to the kinds of tasks that can be executed in a workflow.

A task's entire lifecycle is handled by its task adapter, which loads the task implementation, creates a new task instance, executes the task and destroys the instance. Moreover the Plug-in Task Adapter and the Workspace Task Adapter decorate tasks with a *Task Space*, a persistent memory available to that task only. Tasks use the task space to produce persistent data without any risk of generating conflicts with other tasks, even of the same kind.

III. Building a Debugging Workflow with MASH

This case study shows how MASH can be used to obtain an implementation of the Tarantula algorithm [1] by visually integrating the JUnit, Eclemma, and PSA Eclipse plug-ins[5]. The integration towards a visual workflow rather than the implementation of a new plug-in has many benefits: if enough TB-plug-ins are installed, a visual workflow can be created in few minutes; workflows can be partially executed (e.g., run a different fault localization algorithm without rerunning test cases), changes do not require any deploy operation or any knowledge about the Eclipse architecture; workflows can be reused into other workflows to create complex hierarchical workflows; and MASH workflows can be easily extended with GUI tasks of arbitrary complexity obtained by recording a sequence of GUI actions.

Figure 2 shows a screen capture of the workflow we designed with MASH to implement the Tarantula fault localization algorithm. The left area of Figure 2 shows the Eclipse workspace. The *Debugging* project contains the MASH project. The *MyProgram* project contains the Java program analyzed using the workflow designed with MASH.

The workflow that implements the analysis is the *Debugging workflow* and its definition is shown at the center of Figure 2. The *Debugging workflow* executes two instances of the *CoverTestCases* workflow (the two instances are named *coverPassingTests* and *coverFailingTests*). The definition of the *CoverTestCases* workflow is shown on the right side. For both workflows we show the data-flow only. MASH supports editing of both data- and control-flows.

The *project workflow* receives the parameters *passingTestCases* and *failingTestCases* as input, which are a set of passing and failing test cases, respectively. The inputs of the workflow are used as input for the two instances of the *CoverTestCases* workflow. These workflows return the coverage data produced by the execution of the test cases passed as input parameter. The results returned by *coverPassingTests* and *coverFailingTests* are used as inputs for the task *diff*.

The *diff* task is implemented by the PSA Eclipse plug-in, a plug-in that implements several statistical debugging algorithms like Tarantula. In absence of PSA a user could define a workspace task implemented by a class that compares the lines covered by passed and failed executions to localize bugs according to the Tarantula algorithm. The diff task generates a list of lines of code in *MyProgram* ranked according to the probability that they contain a fault.

The *CoverTestCases* workflow, used twice by the *Debugging* workflow, contains three tasks: *coverTestCases*, *exportCoverage*, and *filterCoverage*. The *coverTestCases* task is a GUI task recorded with MASH. MASH automatically created the parameter *TREE_ITEM_3*, which identifies the test class that must be executed. By connecting *TREE_ITEM_3* with the workflow input *testCasesToExecute* we can run the same GUI task on different test cases. The parametric GUI task consists of the following steps: right click on the class indicated as input, press the menu button *"Coverage as.."*, and press the button *"JUnit Test"* to execute the test cases.

The *ExportCoverage* task is a plug-in task provided by the plug-in *Eclemma-Mash*. This task simply exports the coverage data stored by *Eclemma*. Because *Eclemma* is not a TB-plug-in, we implemented the *Eclemma-Mash* TB-plug-in that reifies *Eclemma* by providing some of its core functionalities as tasks ready to be used in workflows.

The *filterCoverage* task is a workspace task implemented by the *Filter.java* class defined in the workspace. This class opens the coverage file generated by *Eclemma-Mash* and removes coverage data about source code in test classes, which are useless in fault localization.

The MASH framework and the plug-ins used for this example can be downloaded from http://www.lta.disco.unimib.it/tools/mash/. The ICSE demo video that shows the creation of this debugging workflow is available at http://www.lta.disco.unimib.it/mashIcseDemo2012.html.

IV. Related work

There is an increasing need of empowering end-users with the capability of customizing and programming the environments and the applications they use [2]. As a consequence,

[5]downloadable from the Eclispe Update Site http://www.lta.disco.unimib.it/downloads/eclipseUpdates/

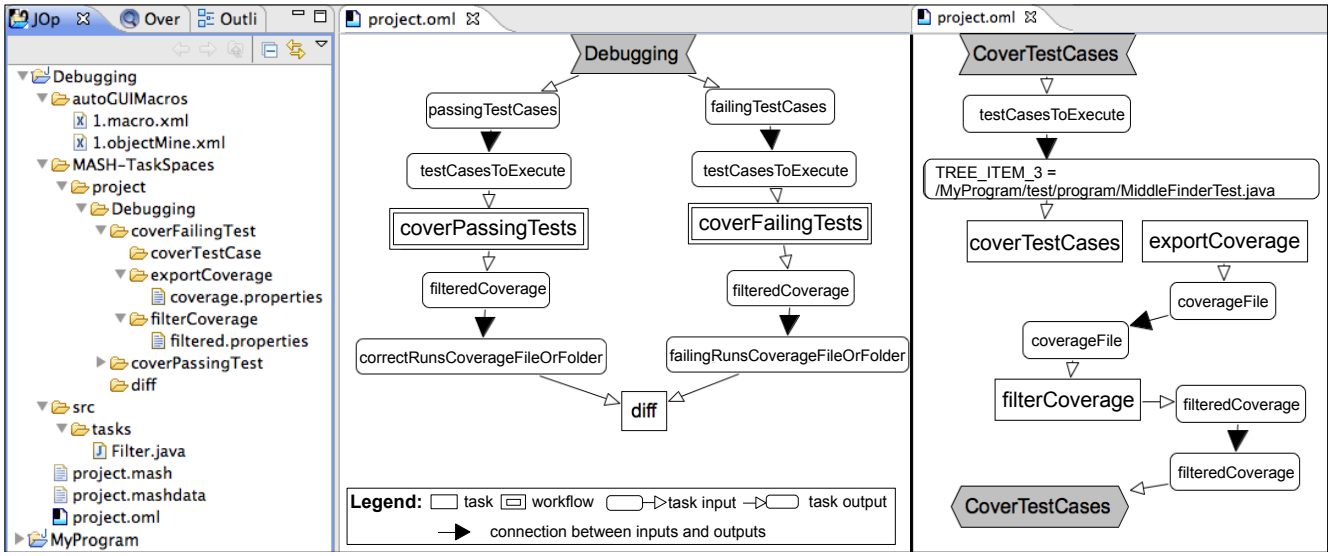

Figure 2. A screen capture of MASH.

the research community is devoting a lot of attention into the attempt of "incorporating software engineering activities into users' existing workflow, without requiring people to substantially change the nature of their work or their priorities" [2]. This need of empowering end-users without asking them to change their style of work is true not only for unskilled users, but also for professional developers, who would like to adapt their programming environment without devoting large effort to the adaptation tasks.

The MASH tool goes into this direction by enabling Eclipse users with the capability of designing and adapting analyses using a workflow language. Some of the ideas in MASH have been early explored in other domains. For instance, Web applications such as Yahoo Pipes[6] already shown the usefulness of frameworks that enable end-users with the capability to mash-up services provided by multiple parties. The recent success of commercial products such as Automator[7] and Automate[8], which can replay sequences of user actions and execute simple GUI scripts, also highlight an increasing need of automation in the end-user environment. In line with these results MASH integrates the flexibility of a plug-in architecture with the power and the simplicity of a visual language, thus allowing end-users to compose plug-in functionalities directly in the IDE.

V. Conclusions

Developers demand for programming environments that can be extended and customized according to project needs. Plug-in architectures provide a number of extension opportunities, however any form of integration or customization that was not foreseen by plug-in developers is still impossible to achieve, unless manually implementing new plug-ins. Even if developers are expert computer users, they seldom invest the time necessary to implement the plug-ins that provide the desired analyses, and thus they loose an opportunity to automate part of their work.

This paper presents MASH, an Eclipse plug-in that supports the composition of the functionalities provided by multipe plug-ins using a workflow visual language. MASH supports multiple kinds of tasks and workflows, and provides a useful front-end for the definition of custom analyses.

[6]http://pipes.yahoo.com
[7]http://www.automator.us
[8]http://www.networkautomation.com/automate/8/

Acknowledgment

This work is partially supported by the European Community under the call FP7-ICT-2009-5 – project PINCETTE 257647.

References

[1] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[2] J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The state of the art in end-user software engineering," *ACM Computing Surveys*, vol. 43, pp. 1–44, April 2011.

[3] L. Mariani and F. Pastore, "Supporting plug-in mashes to ease tool integration," in *proceedings of the First International Workshop on Developing Tools as Plug-ins*. IEEE, 2011.

[4] C. Pautasso and G. Alonso, "The JOpera visual composition language," *Journal of Visual Languages and Computing*, vol. 16, pp. 119–152, 2005.