Supporting Plug-in Mashes to Ease Tool Integration*

Leonardo Mariani
University of Milano - Bicocca
Viale Sarca, 336 - Milano, Italy
mariani@disco.unimib.it

Fabrizio Pastore
University of Milano - Bicocca
Viale Sarca, 336 - Milano, Italy
pastore@disco.unimib.it

ABSTRACT

The majority of IDEs implement a concept of *plug-in* that nicely supports the integration of tools within the IDEs. Plug-ins dramatically simplify the structural integration of multiple tools, but provide little support to the design of the dynamic of the integration, which must be entirely coded by programmers from plug-ins' API.

Manually integrating plug-ins is costly, complex and requires a deep understanding of the underlying environment. The implementation of tools as plug-ins and the integration of the results produced by different plug-ins are still difficult, expensive and error-prone activities.

This paper presents the concepts of Task Based Plug-in (TB-plug-in) and workflow of TB-plug-ins. In our vision, IDE users must be able to execute plug-ins and integrate their results by designing workflows that can be persisted, executed and re-used in other workflows.

We validated our idea by refactoring a set of Eclipse plug-ins for log-file analysis into TB-plug-ins, and designing several workflows that integrate plug-in tasks. We compared the effort necessary to implement these analyses from plug-ins with the effort necessary to design the workflows from TB-plug-ins. We discovered that workflows can be easily designed with little knowledge about the IDE and the plug-ins' API, save significant effort otherwise devoted to the implementation of additional plug-ins and glue-code, and produce analyses that can be quickly modified and reused.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*CASE*

General Terms

Design

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

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

Keywords

Workflows, Task-based plug-in, tool integration, IDE.

1. INTRODUCTION

Modern integrated development environments (IDEs) are characterized by plug-in architectures that ease the integration of tools within IDEs (e.g., Eclipse[1], Netbeans[2] and Microsoft Visual Studio[3]). The integration of a tool in a development environment has two major benefits. It facilitates the adoption of the tool among developers who do not have to waste their time in configuring and running external tools, and it facilitates integration with other tools.

Even if plug-in based architectures significantly evolved in the last years, the customization of plug-ins is still mostly the domain of programmers who implement new tools by reusing the features of existing tools, rather than to IDE users, who need to integrate in an effective and automated way the functionalities provided by different plug-ins. Two well known examples of research tools developed as Eclipse plug-ins are DDstate[4] and Soot[5]. The former can automatically identify the program values that make a program fail. The latter provides static data-flow analysis, including program slicing. DDState and Soot could be integrated to identify the faulty instructions that might be responsible for a failure, but the manual integration of these tools requires the expensive, tedious and error prone development of glue code encapsulated in new plug-ins. Because of this barrier, many technical opportunities that can strongly impact on tool adoption and users productivity remain unexploited.

We strongly believe that the concept of plug-in, especially when used to implement a tool, should provide a stronger support to integration.

In this paper we present *Mash*, a framework where the concept of plug-in is extended with the concept of Task Based Plugin-in (TB-plug-in). A task based plug-in is a plug-in that declares the functionalities that can be executed as tasks. In our view, a task is a (batch or interactive) software process that has inputs, configuration parameters, and outputs. To better accommodate the implementation of tools as plug-ins, each task is associated with a folder (the Task Space) where it can write temporary data or persist results. IDE users can create workflows that execute multiple

[1] `http://www.eclipse.org`
[2] `http://www.netbeans.org`
[3] `http://www.microsoft.com/visualstudio`
[4] http://www.st.cs.uni-saarland.de/eclipse/
[5] http://www.sable.mcgill.ca/soot/

tools and integrate tool results. Workflows can be edited within the IDE itself, can be persisted as part of projects and can be automatically executed.

The benefit of coordinating task executions with workflows is already clear in the service-oriented domain [2], but it is also beneficial to the integration of tools within IDEs:

- workflows are powerful abstractions that can be easily handled by IDE users without requiring advanced IDE programming skills;
- glue code can be implemented as small procedures that are reused across workflows (this strategy requires less effort than implementing glue-code as plug-ins);
- workflow languages are powerful and flexible abstractions that support the rapid and simple design of tool integration thank to their visual representation;
- workflows can be entirely or partially executed, natively supporting the idea that in several cases only part of an analysis process needs to be executed;
- workflows can be reused across different projects when the required TB-plug-ins are installed;
- thanks to the increasing attention to standards and common exchange formats, workflows can be designed without requiring the implementation of a large amount of glue code.

We implemented the *Mash* framework for Eclipse. In particular, we defined the core architectural elements of the framework to support both native TB-plug-ins and standard plug-ins decorated as TB-plug-ins. We integrated the *Mash* framework with the JOpera workflow engine [4], which is distributed as an Eclipse plugin. We validated our ideas by decorating the KLFA [3] and AVA tools [1], released as Eclipse plug-ins, with the interface of TB-plug-ins. We also designed a number of workflows that implement different kinds of analyses and compared the effort necessary to obtain these analyses with and without the support of *Mash*.

Early results show that workflows can be easily designed with little a-priori knowledge about the IDE and the plug-ins' API, save the effort otherwise devoted to the implementation of additional plug-ins and glue-code, and produce analyses that can be quickly modified and reused.

The paper is organized as follow. Section 2 presents the *Mash* architecture. Section 3 presents how we implemented *Mash* in Eclipse. Section 4 presents an early experience with *Mash*. Finally, Section 5 summarizes the work and outlines future research directions.

2. MASH ARCHITECTURE

We envision IDEs as environments where TB-plug-ins can be integrated and composed through pre-defined and user-defined task flows. The conceptual architecture that enables this vision is shown in Figure 1. The top area in Figure 1 shows the structure of TB-plug-ins. A TB-plug-in, in addition to implement standard extension mechanisms (e.g., Eclipse extension-points) exports its functionalities by declaring tasks. A task is an executable unit of work with a name, inputs, outputs and a configuration. The `TB plug-in1` in Figure 1 represents a sample TB-plug-in.

Standard plug-ins can be reified as TB-plug-ins by simply implementing proxy TB-plug-ins that declare tasks and reuse the functionalities in the original plug-ins. Plug-ins `Plug-in2` and `TB-plug-in2` in Figure 1 represent the case of a sample plug-in reified as a TB-plug-in.

TB-plug-ins may optionally export predefined workflows that can be used and executed by end-users similarly to tasks. The major difference between workflows and tasks exported by TB-plug-ins is that tasks are directly executed by the TB-plug-in, while workflows are executed by a workflow engine. Workflows declare only the names of the tasks they execute, while tasks implementation reside in different plug-ins. Workflows exported by plug-ins represent execution flows that are immediately available to IDE users. Plug-in developers can implement these workflows to represent the typical flows associated with a plug-in. End-users can instantiate a pre-defined workflow without the need of understanding the internal details of the plug-in. In addition to execute workflows exported by plug-ins, IDE users can create new workflows, optionally starting from the ones distributed with the plug-ins.

The bottom area in Figure 1 represents the *Mash* framework. *Mash* integrates a workflow engine for running workflows. In addition, *Mash* provides task and workflow adapters that decorate tasks and workflows with features that guarantee the proper execution of these tasks within the IDE. In particular, the task adapter decorates tasks with the capability to persist results in the Task Space (which is a persistent area reserved to the task), the support to task loading, and the handling of the task lifecycle. The workflow adapter similarly decorates workflows with the support to workflow loading and the handling of the workflow lifecycle. The strategy adopted by *Mash* to plug adapters into tasks and workflows depends on the specific IDE; Section 3 describes our Eclipse implementation.

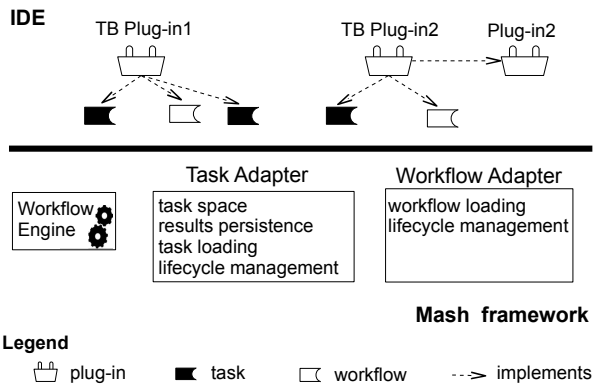

Figure 1: The *Mash* architecture

Figure 2 shows the entities that IDE users can create, execute and modify thank to the underlying *Mash* framework. These entities are typically stored in the project workspace. In particular users can:

- instantiate workflows defined in TB-plug-ins;
- define workflows as compositions of existing tasks and workflows;
- create hierarchical workflows;
- write scripts and use them as tasks to easily implement glue-code;
- define new tasks as classes in the workspace (note that a task differs from a script because it has an associated task-space and a configuration).

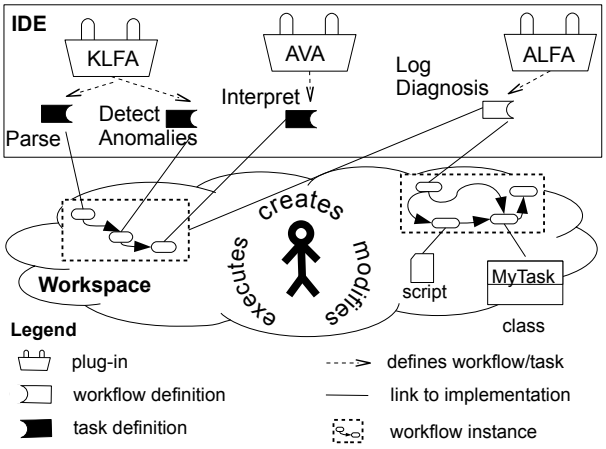

Figure 2: The *Mash* workspace

3. MASH FOR ECLIPSE

We implemented a prototype version of *Mash* for the Eclipse IDE. The prototype extends the notion of Eclipse plug-in to TB-plug-in, provides an implementation of the *Mash* framework, and integrates a workflow engine. Figure 3 shows how *Mash* is implemented in Eclipse.

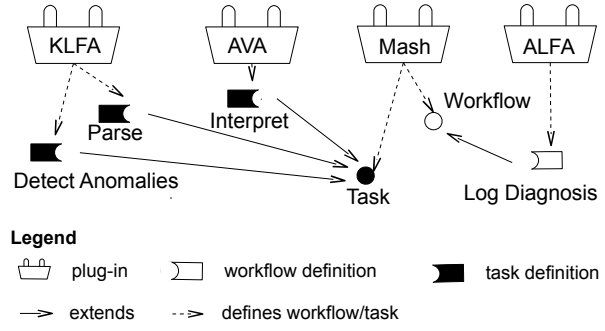

Figure 3: The *Mash* architecture in Eclipse

The entire *Mash* framework is implemented as an Eclipse plug-in. This solution allows the implementation of task and workflow adapters as extension points that are extended by TB-plug-ins. By reusing the Eclipse extension mechanism the TB-plug-ins automatically inherit the behavior in the adapters in a clean and elegant way.

The workflow engine (and editor) that we integrated in *Mash* is JOpera [4]. This engine supports a number of advanced features typical of workflow engines and extremely useful when designing tool integration: parallelism, loops, conditions, data-flow, etc. When a user intends to create a new workflow, *Mash* automatically populates the list of available tasks with the tasks that are accessible in the IDE. The Eclipse version of *Mash* already supports the creation, execution (both complete and partial) and persistence of workflows.

According to our Eclipse *Mash* implementation, plug-in developers can implement the tasks for their plug-ins by implementing the `ITask` interface and defining a `run` method that executes the task. The method `run` is automatically invoked by *Mash* during the execution of a workflow.

In general, tasks can have multiple inputs and outputs. For this reason the `run` method works with two maps: `ITaskInput` and `ITaskOutput`. The `ITaskInput` map is passed by *Mash* to the `run` method, and contains the input values of the task. The `ITaskOutput` map is populated and returned by the task as a result of the execution. *Mash* automatically persists task outputs (without requiring a dedicated support in the task) during the execution of workflows and makes these values available for future inspection.

Mash creates a dedicated folder (the Task Space) for each task. The Task Space extends the notion of workspace folder in Eclipse, and it simply consists of a relative path to a folder in the workspace of the current project. The Task Space is passed to a task as a parameter of the `run` method.

A task may need a configuration wider than the default one. For this reason *Mash* implements the `TaskConfigurationEditor` extension point that can be used to define a configuration panel for each `Task`. In this way developers can programmatically define an ad-hoc GUI that is displayed by *Mash* when users need to configure a `Task`. During workflow execution, *Mash* passes the `TaskConfiguration` to the task which can then behave accordingly.

Some outputs produced by some tools (plug-ins) may require the implementation of ad-hoc editors. *Mash* natively supports the creation of these associations through the extension-point `TaskOutputEditor`. This extension point can be extended to associate editors with the `Task` outputs. IDE users can manually edit task outputs and then re-execute part of the workflow to run "what if" analysis.

Due to lack of space, we do not show the described classes and associations in Figure 3.

Our *Mash* prototype is freely available at `http://www.lta.disco.unimib.it/tools/mash/`.

4. A LOG FILE ANALYSIS FRAMEWORK

We used our Eclipse implementation of *Mash* to create a flexible environment for log file analysis (we called this environment ALFA, which stands for Automatic Log File Analysis). In particular, we reified the KLFA [3] and AVA [1] tools as TB-plug-ins (they were standard Eclipse plug-ins before), and we used *Mash* to define multiple workflows that cover the most significant cases (of course at any time it is possible to add, modify and delete workflows). Note that the reification effort is payed once, and is not payed at all when plug-ins are directly developed as TB-plug-ins.

ALFA can be used to identify failure causes by comparing the log files recorded during a failed execution with logs recorded during successful executions, as shown by the workflow in Figure 4. ALFA first prepares for the analysis the logs recorded during successful executions (using tasks `CorrectLogsProvider`, `RegexBasedLogMessagesParser`, and `ComponentLevelSplitter`) and then produces models that generalize the legal behaviors observed in successful logs (using tasks `KLFAConfigurationBuilder` and `KLFAEngineTraining`). ALFA continues with the preparation of the logs recorded during the failed execution for the analysis (using tasks `LogsToAnalyzeProvider`, `RegexBasedLogMessagesParsing`, `ComponentLevelSplitter` and `SlctEventTypesDetector`) and then compares the log with the generated models to identify failure causes (using task `KLFAEngineChecking`). Failure causes are further processed to build failure cause interpretations that can be easily understood by testers (using task `AVA`).

The ALFA workflows integrate multiple tools (including

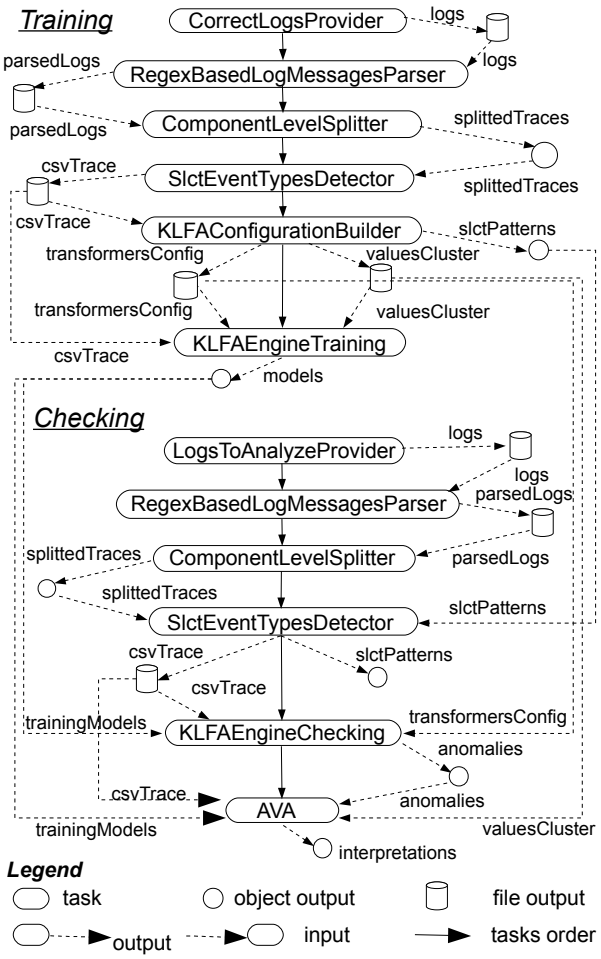

Figure 4: The ALFA workflows defined using *Mash* (to save space we report hand-drawn workflows rather than the JOpera workflows because JOpera represents the control- and data-flow with two different diagrams).

third-party tools, like SLCT [5]) and represent a good example of how the visual design of workflows permits to handle control- and data-flow information with simplicity and flexibility, although tasks accept multiple inputs and produce multiple outputs. Note that the definition of a Task Space makes the location of files that are generated by tasks unambiguous (in this example most tasks produce files). Due to lack of space we do not go into the details of these workflows.

To quantitatively evaluate the benefits of designing tool integration as workflows we report the effort (estimated as number of LOCs) required to develop the preliminary version of ALFA without using *Mash* and the effort (estimated as number of defined workflows and used tasks) required to develop the version of ALFA based on *Mash*. We do not directly report human effort because we do not have precise logs of programmers' activities. However, we believe that the measures we report provide good confidence about the advantage of an approach over the other.

The preliminary and the *Mash* based versions of ALFA use the same implementation of KLFA and AVA. Note that we only compare the effort necessary to develop the code that integrates tool outputs with tool inputs and handles user inputs. We do not consider the effort required to develop additional editors, which is also reduced by *Mash* through the use of the `TaskConfigurationEditor` extension.

The preliminary version of ALFA consists of 1248 lines of code, and has been completed after 63 commits for a total of 2216 committed lines. The *Mash* based version of ALFA required no development effort but only the design of 6 workflows with 6 tasks each on average, thus saving a large amount of development time.

The large development effort required to produce the preliminary version of ALFA is due to the implementation of the following features: saving of temporary files and permanent outputs without producing conflicts; cleaning persistent data when the analysis is rerun; implementing the proper flow of requests and responses to achieve the desired analysis; implementing a flexible flow that can be partially re-executed by reusing the results from previous runs; loading configurations from files. All these features are automatically provided by the Mash framework with the only exception of the design of the flow, which can be defined with the visual workflow editor. Thus *Mash* promisingly reduced the tool integration time. Moreover, the visual representation of the workflow allows IDE users to quickly change the flow of the analysis, while it would be more expensive, difficult and annoying to modify the plug-ins and re-deploy them.

5. CONCLUSION

This paper describes how the concept of plug-in can be extended to TB-plug-in to facilitate tool integration within IDEs. To support execution and integration of TB-plug-ins, we defined a core set of functionalities (represented as pluggable adapters) that decorate tasks and workflows and that are provided by the underlying framework, called *Mash*. We also presented an Eclipse based implementation of *Mash* and we reported early quantitative data that show the benefits of our framework for tool integration.

Future work includes refining the *Mash* framework and defining design patterns that ease the design of TB-plug-ins and Task based IDEs.

6. REFERENCES

[1] A. Babenko, L. Mariani, and F. Pastore. AVA: automated interpretation of dynamically detected anomalies. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[2] C. Lin, S. Lu, X. Fei, A. Chebotko, D. Pai, Z. Lai, F. Fotouhi, and J. Hua. A reference architecture for scientific workflow management systems and the VIEW SOA solution. *IEEE Transactions on Services Computing*, 2:79–92, 2009.

[3] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Proceedings of the International Symposium on Software Reliability Engineering*, 2008.

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

[5] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the Workshop on IP Operations and Management*, 2003.