SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2013; **43**:419–433
Published online 1 May 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2129

MASH: tool integration made easy[‡]

L. Mariani[*,†] and F. Pastore

University of Milano - Bicocca, Milan, Italy

SUMMARY

Integrating tools implemented as plug-ins is a complex and time-consuming activity that requires good programming skills and a deep understanding of the underlying plug-in architecture for successful completion. Even when tools are implemented as plug-ins for a same integrated development environment (IDE), users frequently prefer to manually perform the same inefficient operations every time they use the tools rather than implementing automated tool integration.

In our vision, IDE users must be able to flexibly execute plug-ins and easily integrate their results by designing workflows that can be persisted, automatically executed, and reused in other workflows.

This paper therefore presents MASH, a framework that extends IDEs with task-based plug-ins (TB-plug-ins) and workflows. A TB-plug-in is a plug-in that exposes its functionalities as executable tasks in a workflow. TB-plug-in workflows are processes that automatically execute multiple tools and integrate their results. IDE users can turn regular plug-ins into TB-plug-ins by writing simple scripts or using the GUI capturing feature that MASH offers.

We validated our idea with two case studies examining the design of two data-driven analyses as tool integration. We discovered that workflows can be easily designed by knowing little about the IDE or plug-ins API, saving significant effort otherwise devoted to implementing additional plug-ins and glue code, and they produce analyses that can be quickly modified and reused. Copyright © 2012 John Wiley & Sons, Ltd.

Received 21 September 2011; Accepted 4 April 2012

KEY WORDS: software engineering; tool integration; plug-in architecture; visual languages; workflow; MASH

1. INTRODUCTION

Modern integrated development environments (IDEs) such as Eclipse [1], Netbeans [2], or Microsoft Visual Studio [3] are based on plug-in architectures that allow users to extend the original IDE functionalities by installing new plug-ins. For instance, Eclipse users can install JUnit [4] to automate test case execution and Eclemma [5] to measure code coverage.

The availability of multiple tools in the same environment simplifies the definition of custom analyses that require tool integration. For instance, software developers can easily obtain an automated debugging solution by integrating JUnit and Eclemma: JUnit can execute test cases and Eclemma can identify lines of code covered by each test case. Likely fault locations can be thus identified by computing lines of code executed by the failed test cases but not the ones that passed (well-known automatic debugging techniques such as Tarantula [6] are built upon this intuitive idea). Even if this integration were possible, IDEs provide little support when designing and automating tool compositions unforeseen by plug-in developers. Any custom composition must be manually coded from the plug-in API. For example, comparing lines of code executed by failed and passed test cases

*Correspondence to: L. Mariani, Department of Informatics Systems and Communication, University of Milano - Bicocca, Viale Sarca 336, 20126, Milan, Italy.

†E-mail: mariani@disco.unimib.it

‡Supporting information may be found in the online version of this article.

requires implementing a new plug-in, which implies understanding both the specific plug-in architecture used (here, the Eclipse architecture) and the plug-ins' APIs to be composed (here, JUnit and Eclemma). Automated tool composition thus remains an activity reserved for experts, and users must repeatedly and manually execute operations that an IDE could automatically execute.

From our perspective, IDE users must be able to integrate plug-in functions and results by using simple visual formalisms with limited, or possibly without, coding effort. This paper presents MASH, a framework that supports the definition and execution of workflows that combine plug-in functionalities.

MASH augments IDEs with the concepts of task-based plug-ins (TB-plug-ins) and workflow of TB-plug-ins. A TB-plug-in declares functionalities that can be executed as tasks. A task is a (batch or interactive) software process that has inputs, configuration parameters, and outputs. To avoid conflicts in tool-generated persistent data, each task is associated with a folder (the task space) where it can write temporary data or persist results.

A TB-plug-in workflow is a process that can automatically execute multiple tools and integrate their results. Workflows can be edited within the IDE through graphical editors, can be persisted as part of projects, and can be automatically executed.

To help users compose the functionalities provided by the plug-ins available in the IDE, MASH allows users to define workspace tasks and scripts. Workspace tasks are tasks defined within the IDE workspace rather than within plug-ins. Scripts are code fragments directly specified within a workflow. Both workspace tasks and scripts can use the functions provided by zero or more plug-ins.

To further simplify defining tasks for regular plug-ins and workspace tasks that integrate multiple plug-ins, MASH captures the users' actions when interacting with the GUI of the IDE and saves the sequence of performed actions as a parameterized task (re)usable within workflows.

We implemented the MASH approach as an Eclipse plug-in that augments Eclipse with the notions of TB-plug-in and workflow of TB-plug-ins.

This paper extends the preliminary results described in [7] in the following ways: (1) by providing additional details about the MASH architecture and our Eclipse-based implementation of MASH; (2) by introducing the ability to generate parameterized tasks by capturing user actions, without requiring any programmatic effort from the user; and (3) by evaluating MASH with an additional case study.

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

The paper is organized as follows. Section 2 presents the MASH vision. Section 3 describes how we implemented MASH in Eclipse. Section 4 presents how IDE users can use MASH to create and execute a workflow. Section 5 reports two case studies. Section 6 presents empirical results. Section 7 discusses related work. And finally, Section 8 summarizes the work and outlines future research directions.

2. SUPPORTING END-USERS PLUG-IN COMPOSITION

We envision IDEs as environments where plug-ins can be integrated and composed through predefined and user-defined task workflows. Figure 1 shows the conceptual architecture that enables this vision. The top area of Figure 1 shows the plug-ins' structure and TB-plug-ins installed in the IDE. The bottom area of Figure 1 shows available components in the MASH framework.

Regular plug-ins export functionalities through standard extension mechanisms, including APIs that can be invoked programmatically and widgets that extend the GUI of the IDE. TB-plug-ins, however, export their functionalities as tasks and workflows. Standard plug-ins can be reified as TB-plug-ins by implementing proxy TB-plug-ins that declare tasks and reuse functionalities in the original plug-ins to execute the tasks (Figure 1 shows the possibility of reifying a regular plug-in, with the 'extends' arrow connecting the TB-plug-in to the plug-in).

A task exported by a TB-plug-in is an executable unit of work with a name, inputs, outputs and a configuration. A workflow exported by a plug-in is a complex flow of operations with one or more

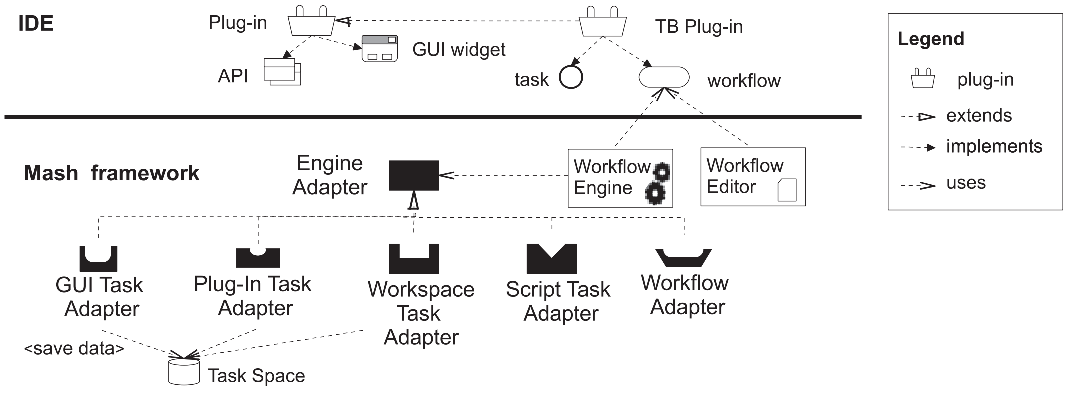


Figure 1. The MASH architecture.

tasks. The major difference between workflows and tasks is that the plug-in directly executes tasks, whereas a workflow engine executes workflows.

Workflows exported by plug-ins represent execution flows immediately available to IDE users. Plug-in developers can implement these workflows to represent operational flows typically associated with a plug-in. IDE users can instantiate a predefined workflow without understanding task details provided by the plug-in. In addition to executing workflows exported by plug-ins, IDE users can also create new workflows, optionally starting from those distributed with the plug-ins.

The bottom area of Figure 1 represents the MASH framework. MASH integrates a workflow editor to edit workflows, a workflow engine to run workflows, and a number of adapters to execute multiple types of tasks.

Every adapter extends the *engine adapter*, an adapter that the workflow engine can execute. The engine adapter is the integration point between the workflow engine and the different types of task. Different MASH implementations can use different workflow engines. Changing the workflow engine requires re-implementing the engine adapter and consequently re-implementing the specific adapters that extend the engine adapter. MASH provides five specific engine adapters that correspond to the kinds of tasks that can be executed in a workflow: the *GUI task adapter*, *plug-in task adapter*, *workspace task adapter*, *script task adapter*, and *workflow adapter*. MASH uses these adapters to bind a task/workflow instance appearing in a workflow definition to its implementation. Task implementation does not depend on the workflow engine used; changing the workflow engine does not require changing the task implementation.

The *GUI task adapter* loads and executes tasks obtained by recording a sequence of actions performed by an IDE user when interacting with the widgets displayed in the IDE. Section 3 describes how to obtain these tasks for the Eclipse IDE.

The *plug-in task adapter* loads and executes the tasks natively exported by TB-plug-ins.

The *workspace task adapter* loads and executes the tasks defined in the user workspace. These are tasks defined by the users implementing Java classes that extend a given task interface. These tasks are not associated with any plug-in, but they can use the functionalities offered by one or more plug-ins invoking plug-in API.

The *script task adapter* executes scripts directly written in the workflow by a workflow editor.

The *workflow adapter* loads and executes the workflows included within other workflows as tasks (hierarchical workflows).

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 GUI task adapter, the plug-in task adapter, and the workspace task adapter decorate tasks with the capability to persist results in the *task space*, which is a persistent area that the MASH framework reserves for tasks. [§]

[§]MASH assigns a dedicated task space to each task instance.

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 parameters according to the workflow specification.

A configuration may influence some task behaviors. For instance, a task that exports code coverage of test case execution may require 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. Plug-in developers can define ad hoc panels, allowing IDE users to change the task configuration. MASH supports visualizing these panels.

When a workflow includes multiple instances of the same task, the same configuration must be shared among these instances. As with input parameters, in MASH, configurations can be visually specified within workflows. To share a configuration, linking the visual element that correlates a task configuration to multiple task instances is sufficient.

Tasks often produce outputs that are not only useful to other tasks but are also meaningful for IDE users. MASH supports creating associations between editors and task outputs. Editors can open, visualize, and modify outputs. IDE users can employ these editors to edit task outputs and re-execute workflow segments to run a ‘what if’ analysis.

Figure 2 shows a possible MASH instance. The left part of Figure 2 shows three plug-ins installed in the IDE: `TB-plug-in1`, `TB-plug-in2`, and `Plug-in2`. `TB-plug-in1` exports a workflow and a task, `TB-plug-in2` both exports some functionalities as tasks and extends the plug-in `Plug-in2`. The right portion of Figure 2 shows some of the artifacts that the IDE user can produce using MASH:

- IDE users can instantiate and use the workflows defined in a TB-plug-in, such as the workflow exported by TB-Plug-in2, which is used in Workflow A;
- IDE users can define workflows as compositions of existing tasks and workflows, such as Workflow A;
- IDE users can write scripts and use them as tasks to easily implement glue code, such as the script used in Workflow A;
- IDE users can implement new tasks as task classes and use these tasks within workflows, such as the task class used in Workflow B (a task differs from a script because it has an associated task space and configuration);
- IDE users can capture sequences of GUI events and instantiate an action sequence as a GUI task, such as one of the tasks used in Workflow B.

3. MASH FOR ECLIPSE

We implemented a prototype version of MASH for the Eclipse IDE [1]. The prototype implements every feature described in Section 2, including the full support for TB-plug-ins, adapters, and a workflow engine.

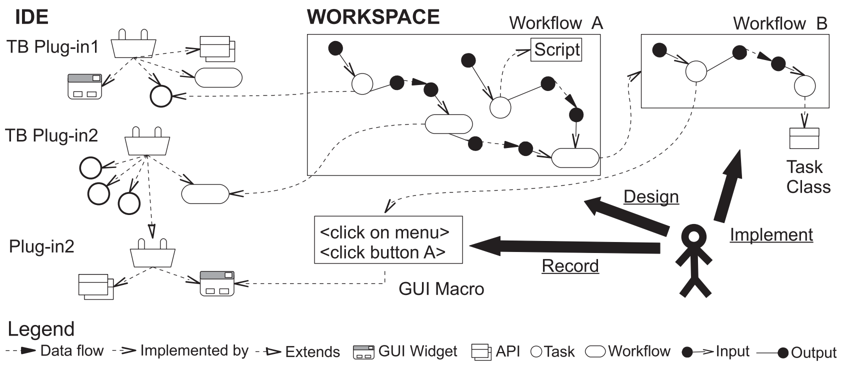


Figure 2. User activities in the MASH workspace.

Figure 3 shows the architecture of our MASH implementation for Eclipse. The entire MASH framework is implemented as an Eclipse plug-in. This solution enables reusing the Eclipse extension point mechanism to define tasks, workflows, editors, and configurations. The MASH plug-in provides four extension points: `task`, `workflow`, `taskOutputEditor`, and `taskConfigurationEditor`. Every new task, workflow, output editor, and configuration editor created by plug-in developers must extend the corresponding extension point to inherit the features provided by the framework.

MASH extends two extension points provided by Eclipse: `newWizard` and `popupMenus`. The `newWizard` extension point is extended by class `NewMashProjectWizard`, which creates a new MASH project under Eclipse (Figure 4 shows the wizard). The `popupMenus` extension point is extended by classes `CreateWorkspaceTask`, `OpenTaskOutputEditor`, `OpenTaskConfigurationEditor`, and `CreateGUITask`, which create various workspace controls (Figure 5 shows the workflow editor menu extended with MASH actions).

MASH integrates the JOpera workflow engine (and editor) [8], which is provided as an Eclipse plug-in. JOpera supports a number of advanced features typical of workflow engines and extremely useful when designing flows that integrate tools: parallelism, loops, conditions, data-flow, etc.

JOpera natively supports the concept of adapters, called `SubSystem`, to extend the number of entities executable by the engine. Our MASH implementation thus uses five specific adapter implementations: the `WorkflowSubSystem`, `ScriptSubSystem`, `PlugInTaskSubSystem`, `GUITaskSubSystem`, and `WorkspaceTaskSubSystem`. The `WorkflowSubSystem` and

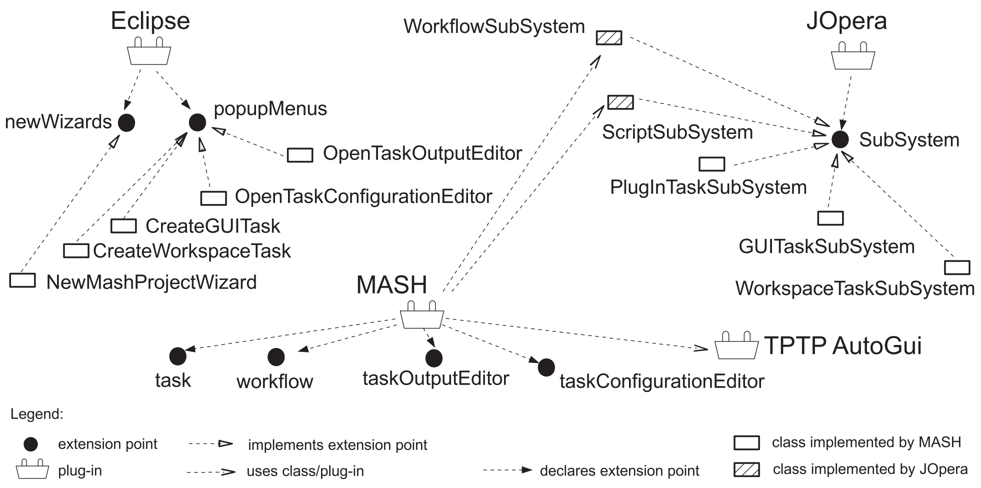
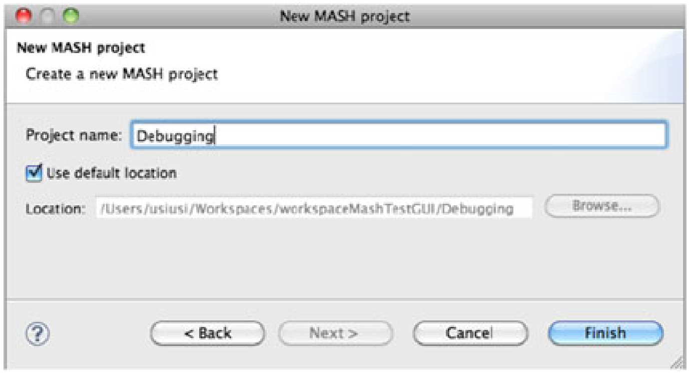


Figure 3. The MASH architecture in Eclipse.


New MASH project

New MASH project
Create a new MASH project

Project name: Debugging
[✓] Use default location
Location: /Users/usiusi/Workspaces/workspaceMashTestGUI/Debugging

Figure 4. The MASH new project wizard.

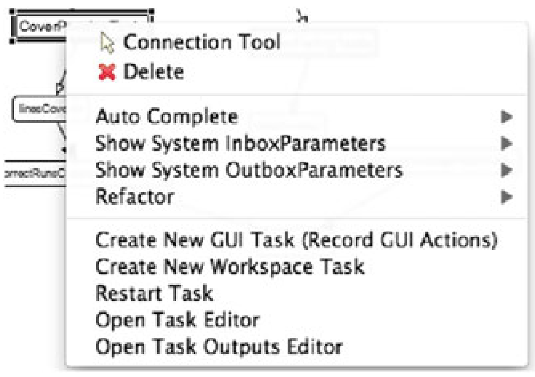

Connection Tool
Delete
Auto Complete
Show System InboxParameters
Show System OutboxParameters
Refactor
Create New GUI Task (Record GUI Actions)
Create New Workspace Task
Restart Task
Open Task Editor
Open Task Outputs Editor

Figure 5. MASH actions in the JOpera menu.

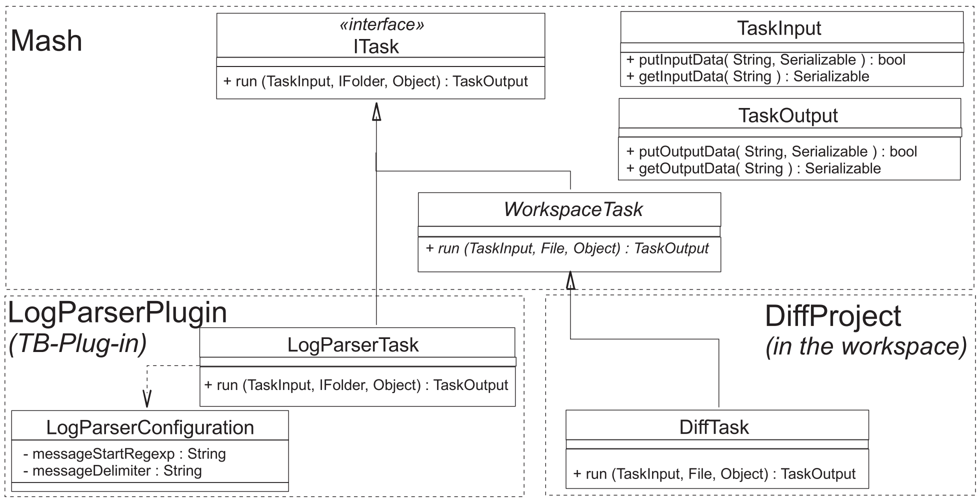

Mash
«interface»
ITask
+ run (TaskInput, IFolder, Object) : TaskOutput
TaskInput
+ putInputData(String, Serializable) : bool
+ getInputData(String) : Serializable
TaskOutput
+ putOutputData(String, Serializable) : bool
+ getOutputData(String) : Serializable
WorkspaceTask
+ run (TaskInput, File, Object) : TaskOutput
LogParserPlugin
(TB-Plug-in)
LogParserTask
+ run (TaskInput, IFolder, Object) : TaskOutput
LogParserConfiguration
- messageStartRegexp : String
- messageDelimiter : String
DiffProject
(in the workspace)
DiffTask
+ run (TaskInput, File, Object) : TaskOutput

Figure 6. Dependencies between two example tasks (`LogParserTask` and `DiffTask`) and the main classes of MASH API.

`ScriptSubSystem` are natively available in JOpera and can be used as they are, whereas the remaining SubSystems are implemented as part of MASH.

JOpera executes the TB-plug-in tasks by means of the following mechanism. JOpera workflows are saved as XML files, where each task instance is associated with a task ID. The JOpera Kernel must know the ID associated with each task implementation, so a task's implementation can be determined every time it must be executed. To satisfy this constraint, MASH automatically registers every task defined by every TB-plug-in at system startup in the JOpera Kernel. The name of every task provided by every TB-plug-in automatically appears in the JOpera editor.

The MASH plug-in uses the TPTPAutoGui plug-in to record and execute GUI tasks. The TPTPAutoGui plug-in implements the TPTP capture and replay technology [9]. This technology records and automatically replays the test cases manually executed by testers who test Eclipse plug-ins. MASH can transform TPTP tests into reusable tasks that can be executed as part of workflows.

MASH uses TPTP to automatically generate parameterized GUI tasks. MASH automatically detects every parameter and literal occurring in a TPTP test and exposes these parameters as task input parameters that IDE users can change when using the task. Even if the recorded test addresses a specific scenario, the corresponding task has a general nature and can thus be more flexibly reused.

Figure 6 shows how a TB-plug-in integrates with the MASH framework. The diagram is split into three areas: the Mash area shows the classes provided by MASH as integration point for TB-plug-ins; the LogParserPlugin area shows a sample task and its configuration implemented as part of a TB-plug-in; and finally, the DiffProject area shows a sample workspace task defined in the project workspace.

A plug-in task is obtained by implementing the `ITask` interface. The interface requires the implementation of a `run` method. The MASH framework executes this method to run the task.

A task's input and output are respectively represented by an instance of type `TaskInput`, which is passed as the first parameter of the `run` method, and `TaskOutput`, which is returned by the `run` method. `TaskInput` and `TaskOutput` encapsulate a map of input and output parameters, respectively. The getters and setters method implemented by these classes retrieve and store the parameters' values. MASH automatically persists task outputs and task inputs without requiring any support from the task. After executing a workflow, the persisted inputs and outputs are available to IDE users for inspection.

A task may need to create persistent data. To avoid conflicts between tasks, MASH assigns a space in the file system, the task space, to each tasks. The reference to this space is passed as the second argument in the `run` method.

Finally, tasks can be configured before execution. The configuration is passed as the third argument of the `run` method. The TB-plug-in must contain an implementation for the class used to store the configuration. In this example, the `LogParserConfiguration` is the configuration object used to configure the `LogParserTask`.

The IDE users can define tasks that stay in the workspace and are not associated with any plug-in. In this example, the `DiffTask` is a workspace task. IDE users must provide an implementation for the task. The class that implements the task must extend the `WorkspaceTask` class. Workspace and plug-in tasks work almost exactly the same with one exception: although plug-in tasks refer to the task space with an object of type `IFolder`, the type that refers folders from code executed in Eclipse plug-ins, the workspace tasks refer to folders by using an object of type `File`, which is defined in the JRE, preventing the generation of dependencies between the code in the workspace and IDE.

For simplicity, Figure 6 does not show the code necessary to define the MASH editors. However, editors are associated with TB-plug-ins much like the way tasks are associated with TB-plug-ins.

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

In the next section, we describe how the IDE user can interact with MASH to create and execute workflows.

4. CREATION AND EXECUTION OF A MASH WORKFLOW IN ECLIPSE

A MASH user typically starts by creating a new MASH project. The MASH project wizard guides the user through a few steps to comfortably create a MASH project. In the project workspace, users are free to create/modify/delete workflows by using JOpera. MASH automatically populates the menu provided by JOpera with the information coming from the installed TB-plug-ins, including the tasks exported by TB-plug-ins, the tasks defined within the workspace, and the GUI tasks recorded by users. For example, users can add a plug-in task to a workflow by selecting the task to be instantiated directly from a JOpera menu.

If the user needs to create a new task integrating multiple Eclipse plug-ins that have not been designed to cooperate, the user can generate a GUI task. The user simply presses the button 'Create New GUI Task' on the JOpera menu, performs the needed action sequence, and then presses the stop button to complete the recording. MASH automatically saves the captured action sequence as a task in the workspace.

Because users often need to customize actions recorded as a GUI task, MASH automatically identifies the parameters associated with every operation performed by the user and exports them as task inputs that can be modified in the workflow editor. For example, the sequence of actions *'right click on the class file Program.java in the project explore view'* and *'press on the run button'* can be customized by allowing users to select different classes in different workflows. The current implementation automatically recognizes text entered in text areas and items selected in a tree as customizable inputs. In this example, the `Program.java` file name automatically becomes a task parameter.

Users can also manually create tasks. To this end, users simply implement a new Java class that extends the abstract class `WorkspaceTask`. The button 'Create New Workspace Task' in the JOpera menu starts a wizard that asks the user for the new name of the class and generates the class code that the user fills with the task implementation. An alternative to implementing tasks as Java classes is writing scripts in workflows. The two approaches are interchangeable. Users should prefer workspace tasks for complex algorithms because they could use IDE facilities to implement and debug workspace tasks (e.g., the Java editor and debugger); however, they should prefer scripts to implement simple code that is easier to export with the workflow.

Simply clicking the run button executes a workflow. If a workflow is interactive, requiring some user inputs, the JOpera engine asks the user to enter the inputs.

5. CASE STUDIES

In this section, we investigate the effectiveness of MASH with two case studies.

Section 5.1 shows how MASH can be used to obtain a fault localization technique as the integration of the Eclemma [5] and JUnit [4] Eclipse plug-ins. This example shows that MASH can be effectively used to produce workflows that integrate regular Eclipse plug-ins. The required level of integration is achieved by designing workflow tasks and GUI tasks.

Section 5.2 presents how MASH can be used to obtain a log file analysis technique as the integration of KLFA [10] and AVA [11]. This example shows how MASH works with TB-plug-ins. In this case study, we produce hierarchical workflows that can suitably replace the interfaces otherwise needed to programmatically achieve the integration.

5.1. Building a debugging workflow with MASH

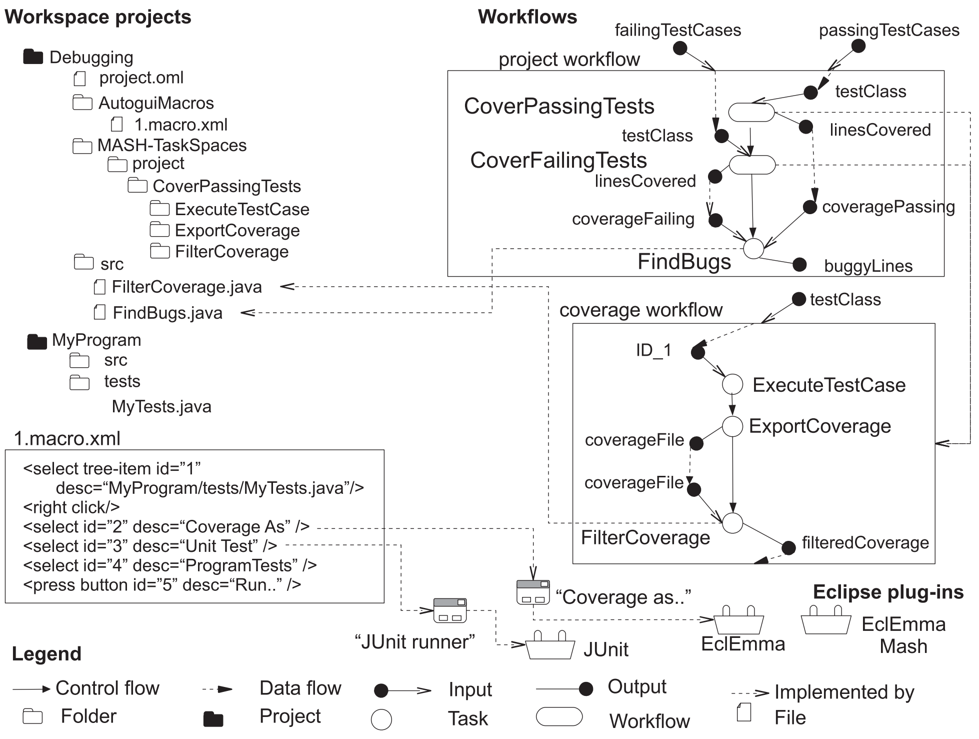
This case study seeks to integrate the JUnit [4] and Eclemma [5] Eclipse plug-ins to obtain a fault localization technique. JUnit is used to execute test cases. Eclemma is used to identify the lines of code covered by each test case. Fault localization is achieved by comparing lines of code covered by failed test cases with those covered by successful test cases according to the Tarantula algorithm [6].

Figure 7 shows the workflow we designed with MASH to combine Eclemma and JUnit. The top-left area of Figure 7 shows the Eclipse workspace. The *Debugging* project workspace contains the project created with MASH to integrate Eclemma and JUnit. The *MyProgram* project workspace contains the Java program analyzed using the workflow. The bottom-right area of Figure 7 shows the plug-ins used in the case study. The main workflow is the *project workflow*, shown in the top right corner of Figure 7. The *project workflow* executes two instances of the *Coverage* workflow (the two instances are *CoverPassingTests* and *CoverFailingTests*). The *Coverage* workflow is shown just below the *project workflow*, and the dotted arrow indicates the inclusion of one workflow into the other.

The *project workflow* receives two parameters as input: *passingTestCases* and *failingTestCases*. The input *passingTestCases* represents a set of passing test cases for the program *MyProgram*, and the input *failingTestCases* represents a set of failing test cases for the same program. The *project workflow* passes the values of *passingTestCases* and *failingTestCases* to the workflows *CoverPassingTests* and *CoverFailingTests*, respectively. These workflows return the coverage data achieved by the input test cases. The results returned by *CoverPassingTests* and *CoverFailingTests* are passed to the task *FindBugs*.

The *FindBugs* task is a workspace task implemented by the *FindBugs.java* class stored in the *Debugging* project workspace. The behavior implemented in this class compares the lines covered by passed and failed executions to localize bugs according to the Tarantula algorithm. This workspace task outputs a file with the lines of code in *MyProgram* ranked according to the probability that they contain a fault.

The *Coverage* workflow, used twice by *project workflow*, contains three tasks: *ExecuteTestCases*, *ExportCoverage*, and *FilterCoverage*. The *Coverage* workflow receives the name of a JUnit test class to be executed as input. The name of the test class is passed to the *ExecuteTestCases* task. Such a task is a GUI task recorded using MASH. This GUI task performs the following sequence



```
<select tree-item id="1"
     desc="MyProgram/tests/MyTests.java"/>
<right click/>
<select id="2" desc="Coverage As" />
<select id="3" desc="Unit Test" />
<select id="4" desc="ProgramTests" />
<press button id="5" desc="Run.." />
```

Figure 7. The MASH debugging workflow.

of actions: right click on the test case class in the Eclipse view `package explorer`, press the menu button `'Coverage as..'`, and press the button `'JUnit Test'` to execute the test case. Figure 7 reports a sketch of the macro corresponding to this GUI task. The macro is stored as an XML file. Even if the macro is readable, MASH does not require IDE users to edit the macro file because parameters are automatically identified and presented as task parameters.

The *ExportCoverage* task is a plug-in task provided by the plug-in *Eclemma-Mash*. This task simply exports the coverage data stored in 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 plug-in tasks.

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

When JOpera is asked to execute the *project workflow*, both JUnit and Eclemma are automatically executed and a ranked list of suspicious lines of codes that may contain a fault is returned to the user. We obtain an implementation of the Tarantula algorithm with almost no programming effort if the *EclEmmaMash* TB-plug-in, which exports the Eclemma functionalities as tasks, is available. [¶]

5.2. *Building a log file analysis framework with MASH*

This case study defines a log file analysis toolsuite. We call this toolsuite ALFA (Automatic Log File Analysis) and seek to use MASH to obtain this toolsuite. The different analysis routines that ALFA can execute are thus MASH workflows. Two TB-plug-ins – KLFA [10] and AVA [11] – provide the actual task implementations in the workflows.

[¶]The EclemmaMash TB-plug-in can be downloaded from the MASH website.

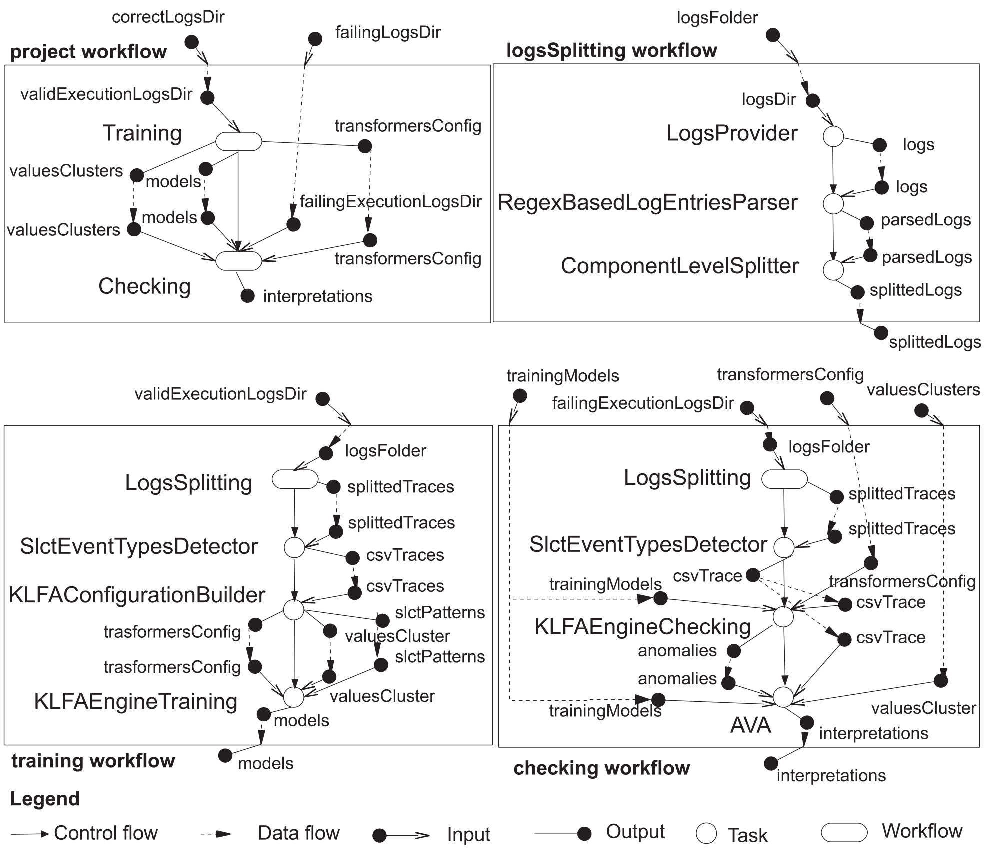


Figure 8. The ALFA workflows defined using MASH.

The log file analysis toolsuite built around KLFA and AVA contains different analysis routines based on the same idea: identifying the likely causes of failures in log files by comparing the content of log files recorded during failed executions with the content recorded during legal executions. Any anomalous event detected in the former log files is a possible failure cause.

Here, we present one ALFA workflow that required significant nesting for definition. Because the focus here is on the workflow, we do not show the project workspace content or the plug-ins used by the workflow.

Figure 8 shows the workflows considered in this case study. The *project workflow* is the main workflow. This workflow executes two other workflows: *Training* and *Checking*. The *Training* workflow analyzes logs from legal executions and produces models that summarize this observed behavior. The *Checking* workflow compares the logs from failed executions with the models returned by the *Training* workflow and returns a set of possible causes for the failure, called interpretations.

Both the *Training* and *Checking* workflows reuse the *LogsSplitting* workflow, shown in the top-right corner of Figure 8. The *LogsSplitting* workflow prepares log files for analysis. The workflow splits input log files into multiple log files where each file contains the events generated by one system component under analysis only. Both the *Training* step, for legal execution files, and the *Checking* step, for failed log files, execute this preparation phase. The Training and Checking workflows create two different instances of workflow *LogsSplitting*. Thus, even if the *LogsSplitting* task creates new files, the two instances have different task spaces and do not conflict with each other.

The *SlctEventTypesDetector* task is also used in both the *Training* and *Checking* workflows. This is a plug-in task provided by KLFA that integrates the SLCT tool [12]. This task processes a log file and automatically distinguishes the fixed part of each event entry from the parameter values that occur in it. This task output is saved into a CSV file. The ability to distinguish the constant part of an event entry and its variable part (the parameters) is important because the model inference solution used in the *Training* workflow and the analysis used in the *Checking* workflow take this distinction into consideration.

The *KLFAConfigurationBuilder* and *KLFAEngineTraining* tasks, provided by the KLFA plug-in, generate a model that summarizes the behavior represented in a set of legal logs. The *KLFAEngineChecking* and *AVA* tasks, provided by the KLFA and AVA plug-ins, compare a set of models inferred from legal traces and log files recorded in failed runs.

The four workflows provide a relevant example of how visual workflow design handles control-flow and data-flow information with reasonable simplicity and flexibility, although tasks accept multiple inputs and produce multiple outputs.

Adopting MASH to build ALFA enables users to easily customize the workflows. For example, a user can prepare log files differently by replacing the *ComponentLevelSplitter* task with the *Action-LevelSplitter* task. Once the task is replaced, the analysis can be rerun. If the user wants to change a configuration parameter in the *KLFAEngineChecking* task, the user can visually edit its configuration and rerun the analysis from that point (when a workflow is executed, MASH retains every intermediate result; it is thus unnecessary to rerun the entire workflow if applying small changes).

6. PRELIMINARY EVALUATION

To preliminary evaluate the effectiveness of MASH, we compared the effort required to define the two analyses described in Section 5 with and without using MASH. Table I summarizes the results.

Column *Case study* indicates the case name: *Bug Finder* is the case study described in Section 5.1 and *ALFA* is the case study described in Section 5.2.

Column *Coding* reports data about the effort required to implement *Bug Finder* and *ALFA* without using MASH. To measure this effort, Fabrizio Pastore, an author of this paper, implemented the Eclipse plug-ins necessary to compose the required functionalities and obtain the integration described in the case study section. Because Fabrizio Pastore is an expert developer of Eclipse plug-ins, the measured effort does not account for any cost related to learning the Eclipse plug-in architecture. In many practical situations, the effort required to learn the plug-in architecture and API is substantial.

To quantify the development effort, we report the size of the developed plug-ins in terms of lines of code (Column *LOC*), the number of commits produced by the developer (Column *CVS commits*), and the total number of lines modified by the commits to provide the amount of code written and modified before reaching the final version (Column *Lines modified*).

Column *MASH* reports the effort required to design the workflows with MASH. To measure this effort, we report the number of tasks (Column *Tasks*) and edges (Column *Edges*) to indicate the size and complexity of the defined workflows.[‖]

Although the data collected for the coding and MASH approaches are not directly comparable (the ALFA and Bug Finder plug-ins were developed before MASH, thus it is impossible to report data about the time necessary to develop them), Table I still provides some insights. With simple integration, the difference between MASH and the coding approaches is relatively large. For instance, the Bug Finder case study can be implemented by writing 250 LOCs, whereas a workflow with six tasks can define the MASH workflow. However, it should be intuitive that correctly

Table I. Effort required to implement plug-in compositions with and without MASH.

	Coding			MASH	
Case study	LOC	CVS commits	Lines modified	Tasks	Edges
Bug Finder	175	5	250	6	10
ALFA	1248	63	2216	13	35

[‖]The number of edges includes the data- and control-flow edges shown in Figures 7 and 8.

connecting six tasks required much less work than writing more than 200 LOCs. Moreover, even unskilled IDE users can work with MASH, whereas experienced developers are needed to design Eclipse plug-ins.

In the ALFA case study, the difference between the coding and MASH approaches is clearer. The number of LOCs written for ALFA is more than seven times the number of LOCs written for Bug Finder, whereas MASH required a workflow design only twice the size of the workflow designed for Bug Finder. This early evidence suggests good scaling capability for MASH compared with coding plug-ins. In this case, the effort gap between the two approaches favors MASH. The coding approach required approximately 1.2 KLOCs, whereas MASH required a hierarchical workflow designed with 13 tasks.

Finally, the *Coding* solution that produced the data shown in Table I is weaker in its provided functionalities. MASH provides features that the plug-in implementation to which we refer does not, including the abilities to re-execute workflows from an intermediate task and automatically persist the inputs and outputs produced at every computational step.

7. RELATED WORK

MASH augments IDEs with TB-plug-in workflows to support users when designing complex plug-in compositions. This section provides an overview of research results in the related fields of plug-in architectures and end-user software engineering.

7.1. Plug-in and component-based architectures

The term plug-in commonly refers to a pluggable software module that augments functionalities of the software application into which it is integrated. The definition of plug-in slightly differs in different application domains. Fowler uses the term *plug-in* to indicate a design pattern that is applied 'to link classes during configuration rather than compilation' [13]. Bouler defines an Eclipse plug-in as 'a component that provides a certain type of service within the context of the Eclipse workbench' [14]. Firefox plug-ins are defined as 'shared libraries that users can install to display content that the application itself can't display natively' [15]. The operating systems community does not use the term plug-in, although *software modules* runtime integration is a key feature of modern operating systems. For example, the Linux kernel allows users to install both loadable kernel modules that extend the kernel functionalities and device drivers that allow the system to use devices [16].

The need for modularization is not new but has long been a key concept in software system development. Modular software systems are often built as an integration of multiple software components [17]. Meyer defines a component as a software element (modular unit) that can be used by other components, comes with an official usage description, and is not tied to any fixed set of clients [18]. According to this definition, the terms *plug-in* and *component* are synonymous.

Lau and Wang [19] propose a taxonomy of existing component models in which components are classified according to the development phase during which composition takes place and the possibility for composite components' reuse. Many authors consider encapsulation and compositionality to be the major features lacking in existing component models, whereas they do not consider dynamic module loading and updating to be main features for component frameworks, although they are main features in modern plug-in architectures, such as OSGi [20] and the related implementations Eclipse-Equinox [21] and Spring [22].

MASH responds to a lack of support for designing new plug-in compositions in existing IDEs and plug-in frameworks. Component architecture research focuses on developing frameworks that enable system distribution, dynamic system reconfiguration, and adaptation. For example, Kramer and Magee [23] propose a three-tiered architectural model to develop a self-managed software system that automatically reconfigures system components to reach a high-level goal. Other authors focus on dynamic updates [24] and dynamic component discovery [25], thus allowing users to improve the system with features provided by third parties; they do not provide feature integration support as MASH does.

When integrating third-party components, a user must often produce some glue code that allows components to communicate through incompatible interfaces or data types. A well-known solution is implementing adapters for the incompatible types [26]. MASH reuses this basic design idea, supporting the definition of adapters as scripts and workspace tasks that implement the required glue code.

Automatic glue code generation further helps IDE users integrate different plug-ins. Unfortunately, the techniques currently available are inapplicable to our context, and manual integration still remains the most feasible solution. Cao *et al.* [27] automatically derive glue code by using component specifications written in two-level grammar. This approach is inapplicable to an IDE plug-in context because the plug-ins are not usually provided with any formal specification. Liu *et al.* [28] combine deductive synthesis and code patterns to automatically derive the sequences of operations that must be invoked to effectively combine components. This approach could help invoke plug-in tasks according to a given protocol. However, we expect plug-in developers to provide their plug-ins already decorated with the workflows corresponding to possible usage scenarios.

7.2. End-user software engineering

The number of end-users involved in program definition is growing: One example is the number of users writing formulas and queries at work by using spreadsheets and databases [29]. End-user software engineering is a novel research field that aims to ‘incorporate software engineering activities into users’ existing workflow, without requiring people to substantially change the nature of their work or their priorities’ [30]. End-user software engineering thus supports end-users in designing, testing, and debugging their programs. The recent survey by Ko *et al.* on end-user software engineering [30] shows that end-user software engineering program design advances primarily involve defining techniques to make user interface definition easier (e.g., spreadsheet [31] and websites [32]) or supporting programming by examples (e.g., McDaniel captures GUI events to be reproduced by the program at runtime [33]). MASH contributes to end-user software engineering by introducing TB-plug-in composition to improve IDE functionalities and flexibility.

Web applications such as Yahoo Pipes [34] show the usefulness of frameworks that enable end-users to mash-up services provided by different parties. The recent success of commercial products such as Automator [35] and Automate [36], which automate the user action workflow execution, also highlights this point. In particular, Automator is an Apple [37] tool that supports simple workflow definitions that combine a set of actions provided by the system (e.g., create a new file, take a screenshot). Automator supports script definition and recording GUI macros to replicate actions. Experienced programmers can create new Automator actions as Objective-C programs.

Some functionalities provided by MASH are also present in products such as Automator and Automate. However, MASH is specifically designed to work with IDEs and plug-in architectures. When MASH is used with TB-plug-ins, it has greater flexibility than simply replying to sequences of actions. Users can configure and compose tasks, customize recorded workflows, handle any GUI event produced within Eclipse, and are not limited by the operating system or underlying platform.

Many approaches dedicated to end-users use visual languages [30]. Workflow languages, for example, are commonly adopted to compose functionalities provided by third-party components, including web services [38] and web mash-ups [39]. MASH exploits workflow language benefits to support IDE users when integrating plug-in functionalities and results [7].

8. CONCLUSION

This paper describes how the plug-in concept can be extended to a 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 are provided by an underlying framework called MASH. We also presented an Eclipse-based MASH implementation and reported early quantitative data that show our framework’s benefits in tool integration.

Given the increasing attention paid to using plug-in frameworks as building blocks for creating rich client applications, we think the ideas presented in this paper can be extended beyond the scope of IDEs. For instance, software systems implemented with the Eclipse RCP framework could easily exploit MASH. Notable examples are CCTVnet [40], a video/monitoring recording software, and ProgmateDoc [41], a document management system.

Future work includes increasing the support to the flexible and simple workflow design to integrate plug-in functionalities and increasing our experience with MASH's effectiveness through additional case studies.

ACKNOWLEDGEMENTS

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

REFERENCES

1. Eclipse, 2011. http://www.eclipse.org/. [2012]
2. Netbeans, 2011. http://www.netbeans.org/. [2012]
3. Visualstudio, 2011. http://www.microsoft.com/visualstudio. [2012]
4. Junit, 2011. http://www.junit.org/. [2012]
5. Eclemma, 2011. http://www.eclemma.org/. [2012]
6. Jones JA, Harrold MJ. Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering,* ASE'05, ACM, 2005; 273–282.
7. Mariani L, Pastore F. Supporting plug-in mashes to ease tool integration. *First International Workshop on Developing Tools as Plug-ins (topi) - Colocated with the International Conference on Software Engineering, ICSE'11*, IEEE, 2011; 1–4.
8. Pautasso C, Alonso G. The JOpera visual composition language. *Journal of Visual Languages and Computing* 2005; **16**:119–152.
9. Eclipse test and performance tools platform project, 2011. http://www.eclipse.org/tptp/.
10. Mariani L, Pastore F. Automated identification of failure causes in system logs. *Proceedings of the International Symposium on Software Reliability Engineering*, 2008; 117–126.
11. Babenko A, Mariani L, Pastore F. AVA: automated interpretation of dynamically detected anomalies. *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, 2009; 237–248.
12. Vaarandi R. A data clustering algorithm for mining patterns from event logs. *Proceedings of the Workshop on IP Operations and Management*, 2003; 119–126.
13. Fowler M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2002.
14. Bolour A, Computing B. Notes on the eclipse plug-in architecture, 2003. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html. [2012]
15. Mozilla developers network plug-in definition, 2011. https://developer.mozilla.org/en/Plugins. [2012]
16. Bovet D, Cesati M. *Understanding the linux kernel, second edition*, 2nd ed. O'Reilly & Associates, Inc.: Sebastopol, CA, USA, 2002.
17. Parnas DL. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 1972; **15**(12):1053–1058.
18. Meyer B. The grand challenge of trusted components. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03. IEEE Computer Society: Washington, DC, USA, 2003; 660–667.
19. Lau KK, Wang Z. Software component models. *IEEE Transactions on Software Engineering* October 2007; **33**:709–724.
20. Tavares AL, Valente MT. A gentle introduction to OSGi. *ACM SIGSOFT Software Engineering Notes* August 2008; **33**:8:1–8:5.
21. Eclipse equinox, 2011. http://www.eclipse.org/equinox/. [2012]
22. Spring framework, 2011. http://www.spring.org. [2012]
23. Kramer J, Magee J. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*, Briand LC, Wolf AL (eds), FOSE '07. IEEE Computer Society: Washington, DC, USA, 2007; 259–268.
24. Gregersen AR, Jørgensen BN. Towards dynamic plug-in replacement in eclipse plug-in development. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange*, Eclipse'07. ACM: New York, 2007; 41–45.
25. Wolfinger R, Loberbauer M, Jahn M, Mössenböck H. Adding genericity to a plug-in framework. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10. ACM: New York, 2010; 93–102.
26. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1995.
27. Cao F, Bryant BR, Burt CC, Raje RR, Olson AM, Auguston M. A component assembly approach based on aspect-oriented generative domain modeling. *Electronic Notes in Theoretical Computer Science* 2005; **114**:119 –136. Proceedings of the Software Composition Workshop (SC 2004).

28. Liu J, Fu J, Zhang Y, Bastani F, Yen IL, Tai A, Chau S. Deductive glue code synthesis for embedded software systems based on code patterns. In *Proceedings of the Ninth IEEE International Symposium on Object and Component-oriented Real-time Distributed Computing*, ISORC '06. IEEE Computer Society: Washington, DC, USA, 2006; 109–116.
29. Scaffidi C, Shaw M, Myers B. Estimating the numbers of end users and end user programmers. In *Proceedings of The 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society: Washington, DC, USA, 2005; 207–214.
30. Ko AJ, Abraham R, Beckwith L, Blackwell A, Burnett M, Erwig M, Scaffidi C, Lawrance J, Lieberman H, Myers B, Rosson MB, Rothermel G, Shaw M, Wiedenbeck S. The state of the art in end-user software engineering. *ACM Computing Surveys* 2011; **43**:21:1–21:44.
31. Powell SG, Baker KR. *The Art of Modeling with Spreadsheets: Management Science, Spreadsheet Engineering, and Modeling Craft*. Wiley: New York, 2004.
32. Wolber D, Su Y, Chiang YT. Designing dynamic web pages and persistence in the WYSIWYG interface. In *Proceedings of the 7th International Conference on Intelligent User Interfaces*, IUI '02. ACM: New York, 2002; 228–229.
33. McDaniel RG, Myers BA. Getting more out of programming-by-demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: The Chi is the Limit*, CHI '99. ACM: New York, 1999; 442–449.
34. Yahoo pipes, 2011. http://pipes.yahoo.com. [2012]
35. Automator, 2011. http://www.automator.us. [2012]
36. Automate 8, 2011. http://www.networkautomation.com/automate/8/. [2012]
37. Apple, 2011. http://www.apple.com/. [2012]
38. Lin C, Lu S, Fei X, Chebotko A, Pai D, Lai Z, Fotouhi F, Hua J. A reference architecture for scientific workflow management systems and the VIEW SOA solution. *IEEE Transactions on Services Computing* 2009; **2**:79–92.
39. Biörnstad B, Pautasso C. Let it flow: Building mashups with data processing pipelines. In *Service-Oriented Computing - ICSOC 2007 Workshops*, Nitto E, Ripeanu M (eds). Springer-Verlag: Berlin, Heidelberg, 2009; 15–28.
40. Cctvnet, 2011. http://www.diligent-it.com. [2012]
41. Progmate, 2011. http://www.progmate.pl. [2012]