The Journal of Systems and Software 111 (2016) 89–104

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

On the design of a maintainable software development kit to implement integration solutions

Rafael Z. Frantz [a,*], Rafael Corchuelo [b], Fabricia Roos-Frantz [a]

[a] Department of Exact Sciences and Engineering, Unijuí University, Rua do Comércio, 3000, Ijuí 98700-000, RS, Brazil
[b] ETSI Informática, University of Seville, Avda. Reina Mercedes, s/n, Sevilla 41012, Spain

ARTICLE INFO

Article history:
Received 5 July 2013
Revised 14 July 2015
Accepted 25 August 2015
Available online 16 September 2015

Keywords:
Enterprise Application Integration
Integration framework

ABSTRACT

Companies typically rely on applications purchased from third parties or developed at home to support their business activities. It is not uncommon that these applications were not designed taking integration into account. Enterprise Application Integration provides methodologies and tools to design and implement integration solutions. Camel, Spring Integration, and Mule range amongst the most popular open-source tools that provide support to implement integration solutions. The adaptive maintenance of a software tool is very important for companies that need to reuse existing tools to build their own. We have analysed 25 maintainability measures on Camel, Spring Integration, and Mule. We have conducted a statistical analysis to confirm the results obtained with the maintainability measures, and it follows that these tools may have problems regarding maintenance. These problems increase the costs of the adaptation process. This motivated us to work on a new proposal that has been carefully designed in order to reduce maintainability efforts. Guaraná SDK is the software tool that we provide to implement integration solutions. We have also computed the maintainability measures regarding Guaraná SDK and the results suggest that maintaining it is easier than maintaining the others. Furthermore, we have conducted an industrial experience to demonstrate the application of our proposal in industry.

1. Introduction

Companies rely on applications to support their business activities. Frequently, these applications are legacy systems, packages purchased from third parties, or developed at home to solve a particular problem. This usually results in heterogeneous software ecosystems, which are composed of applications that were not usually designed taking integration into account. Integration is necessary, chiefly because it allows to reuse two or more applications to support new business processes, or because the current business processes have to be optimised by interacting with other applications within the software ecosystem. Enterprise Application Integration provides methodologies and tools to design and implement integration solutions. The goal of an Enterprise Application Integration solution is to keep a number of applications' data in synchrony or to develop new functionality on top of them, so that applications do not have to be changed and are not disturbed by the integration solution (Hohpe and Woolf, 2003).

In the last years, several tools have emerged to support the design and implementation of integration solutions. Hohpe and Woolf (2003) documented many patterns found in the development of integration solutions. These patterns basically aim to support three core concepts, namely: pipes, filters, and resource adapters. Camel, Spring Integration, and Mule range amongst the most popular open-source tools that provide support for some of these integration patterns. Camel provides a fluent API (Fowler, 2010) that software engineers can use programmatically or by means of a graphical editor. In both cases, the integration solution is implemented using a Java, Scala, or XML Spring-based configuration files. Spring Integration was built on top of the Spring Framework container, and provides a command-query API (Fowler, 2010). This tool can be used programmatically or by means of a graphical editor. Integration solutions are implemented using either Java code or an XML Spring-based configuration file. The architecture of Mule got inspiration from the concept of enterprise service bus. Software engineers count on a command-query API (Fowler, 2010) to use this tool programmatically, or a workbench to design and implement integration solutions using a graphical editor. Integration solutions are implemented using either Java code or an XML Spring-based configuration file. In earlier versions, Mule supported a limited range of integration patterns; version 3.0 resulted in a complete re-design whose focus was on supporting the majority

* Corresponding author. Tel.: +55 5533320200.
E-mail addresses: rzfrantz@unijui.edu.br (R.Z. Frantz), corchu@us.es (R. Corchuelo), frfrantz@unijui.edu.br (F. Roos-Frantz).

http://dx.doi.org/10.1016/j.jss.2015.08.044
0164-1212/

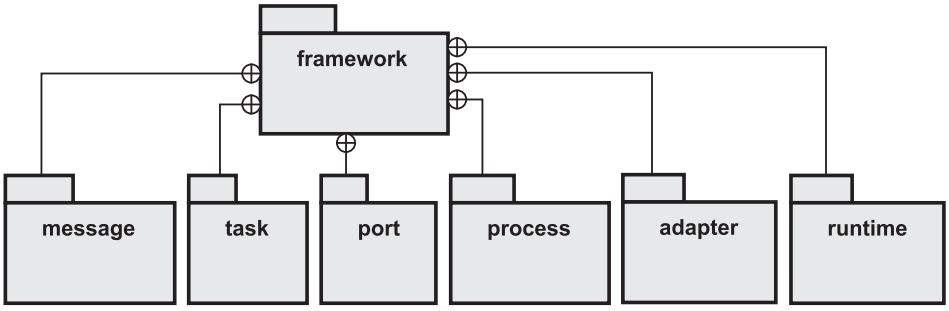

Fig. 1. Packages of which our framework is composed.

of integration patterns. As of the time of writing this article, Camel, Spring Integration, and Mule are at version 2.7.1, 2.0.3, and 3.1, respectively. In the rest of the article, we implicitly refer to these versions.

We are concerned with maintainability. According to IEEE (1990), maintenance can be classified as corrective, perfective, and adaptive. Corrective maintenance aims to repair software systems to eliminate faults that might cause them to deviate from their normal processing. Perfective maintenance aims to modify a software system, usually to improve the performance of current functionalities or even to improve the maintainability of the overall software system. Adaptive maintenance focuses on adapting a software system to use it in new execution environments or business processes.

In this article, we are interested in adaptive maintenance, which is very important for companies that need to reuse existing tools to build their own (Chen and Huang, 2009). Many companies rely on open-source tools that can be adapted to a specific context within their business domain. For example, a company that develops Enterprise Application Integration solutions may need tools that focus on specific contexts such as e-commerce, health systems, financial systems, and insurance systems to meet standards and recommendations like RosettaNet (2011), HL7 (2011), Swift (2011), and HIPAA (2011), respectively. Other authors have evaluated open-source tools from a performance point of view (García-Jiménez et al., 2010); we think that our work is complementary.

It is not new that the design and implementation of a software system has an impact on its maintenance costs (Epping and Lott, 1994; Jorgensen, 1995; Bergin and Keating, 2003; Schneidewind, 1987). International standards such as ISO 9126-1 (ISO/IEC, 2001) or more recent ISO 25010 (ISO/IEC, 2011) define quality models that help to understand what may have an impact on the maintainability of software systems. According to these standards, the maintainability of a software system can be influenced by the amount of effort to change the system (Changeability), the capability of a software to avoid collateral effects produced by changes on it (Stability), the ability to identify and diagnose failures (Analysability), and the effort to verify the software after changes (Testability). In both design and implementation, software engineers need to pay attention to readability, understandability, and complexity, since they are related to several subcharacteristics that characterise maintainability. Thus, the resulting models and source code must be easy to read and understand, because it is very common that the people who work on them shall not maintain them. The complexity of the algorithms should be kept low, not only for performance reasons, but because it makes it easier for a software engineer to follow their execution flows and debug them. Thus, to reduce the costs involved in the adaptation of a software system to a specific context, it is very important that the software system was designed taking into account issues that have a negative impact on maintenance.

How costly it is to maintain a tool depends on a variety of measurable properties. We have computed these measures on Camel, Spring Integration, and Mule, and the results do not seem promising enough. The focus of this article is only on the core implementation of these proposals, which have similar functionalities, since the core aim at providing support for the integration patterns documented by Hohpe and Woolf (2003). The results motivated us to work on a Software Development Kit (SDK) to which we refer to as Guaraná SDK.[1] The design decisions and the implementation of the core of Guaraná SDK had always maintainability in mind. The result is that its design provides better values for the maintainability measures regarding its core implementation, which suggests that its core is more maintainable and thus easier to adapt for a particular context than the core implementation of Camel, Spring Integration, or Mule. The core of our proposal also aims at providing support for the integration patterns. The core of Guaraná SDK is composed of two layers, namely: the framework and the toolkit. The former provides a number of classes and interfaces that provide the foundation to implement tasks, adapters, and workflows, as well as a Runtime System to which we deploy and run the integration solutions; the latter extends the framework to provide an implementation of tasks and adapters that is intended to be general purpose.

A six-page abstract regarding our results was presented in Frantz and Corchuelo (2012); in this article, we extend our preliminary paper as follows: we analyse 16 additional maintainability measures, we analyse an additional wide-spread open-source tool, Mule, we provide a statistical analysis based on Kolmorogov–Smirnov's test, Shapiro–Wilk's test, Iman–Davenport's test, and Bergmann–Hommel's test to confirm our intuitive conclusion from the results obtained with the maintainability measures, we provide a comprehensive description of each layer of Guaraná SDK, and we demonstrate our proposal by means of an industrial experience that has been developed in co-operation with a spin-off company. We have also developed a domain-specific language that is intended to facilitate designing integration solutions at a high level of abstraction (Frantz et al., 2011).

The rest of the article is organised as follows: Section 2 presents the framework layer of Guaraná SDK; Section 3 presents the toolkit layer; Section 4 presents the experimental study we conducted; Section 5 presents an industrial experience on which we have worked; finally, Section 6 reports on our main conclusions.

2. The framework layer

In this section, we describe the framework layer. Fig. 1 provides an overview of this layer by showing the six packages of which it is composed. In the following subsections we describe each package.

2.1. Messages

Messages are used to wrap the data that is manipulated in an integration solution. They are composed of a header, a body and one or more attachments, cf. Fig. 2.

[1] Guaraná technology is available at http://www.guaranasolutions.com.

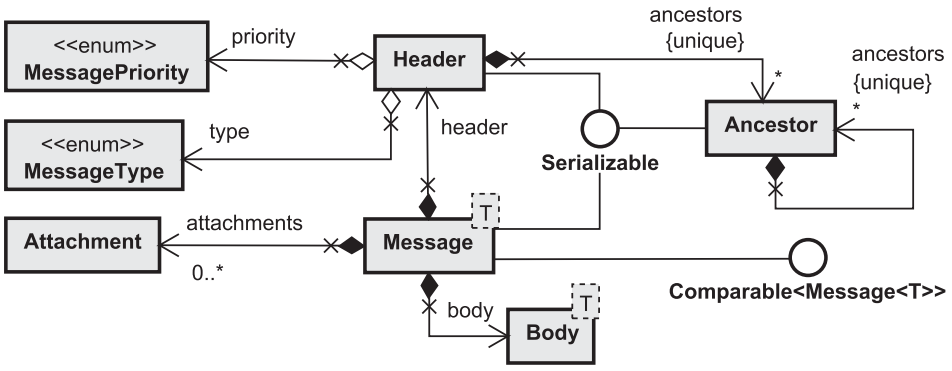

Fig. 2. Message model.

The header includes custom properties and the following predefined properties (not shown in Fig. 2 for simplicity): message identifier, correlation identifier, sequence size, sequence number, return address, expiration date, message priority, message type, and list of ancestors. The message identifier is represented using an immutable universally unique identifier value of 128-bits, which is automatically assigned to every message when they are created. The correlation identifier holds the identifier of another message to which the current message is correlated. Sequence size and sequence number are used to identify a message in a sequence of messages so that they can be grouped. The expiration date allows to set a deadline after which a message is considered outdated for further processing. The message priority is an enumerated value, namely: lowest, low, normal (default), high, and highest. The message type is an enumerated value that indicates whether the message represents a command, an event (default), a request, or a response. A command message aims to invoke an operation at its destination without expecting any responses; an event message is used for asynchronous notification purposes and carries data that keeps applications up to date; a request message is similar to a command message, however it always expects a reply that is a response message. The list of ancestors allows to track which messages originate from which ones; this is important in order to find out which messages have been processed as a whole and form a so-called correlation.

The body holds the payload data, whose type is defined by the template parameter in the message class. Attachments allow messages to carry extra pieces of data associated with the payload, e.g., an image or an e-mail message. Data in the attachments are not intended to be processed, which is not a shortcoming at all; bear in mind that messages are defined by the users, so they can freely decide which information is stored in the body and which information is carried forward as attachments.

Messages implement two interfaces so that they can be serialised and compared, respectively. Serialisation is required to deep copy, to persist, and to transfer messages; comparison enables the integration solution to process them according to their priority.

2.2. Tasks

This package provides the foundations to implement domain-specific tasks in specialised toolkits, cf. Fig. 3. Roughly speaking, a task models how a set of inbound messages must be processed to produce a set of outbound messages, e.g., routing the inbound messages, modifying them, transforming them, performing time-related actions, stream-oriented actions, mapping them to/from objects, or reading and writing messages, to name a few categories that are supported by the toolkit introduced in Section 3.

Tasks communicate indirectly by means of slots to which they have access by means of so-called gateways. A slot is an in-memory priority buffer that helps transfer messages asynchronously so that

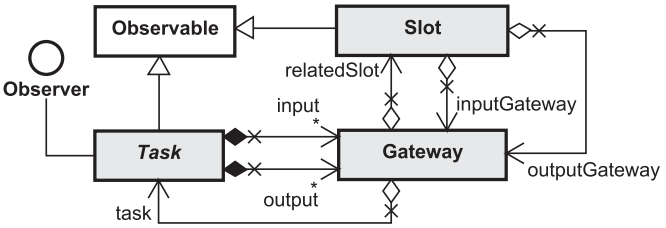

Fig. 3. Task model.

no task has to wait until the next one is ready to start working. Gateways act like a connection point between a slot and a task, by providing an interface to add/take messages to/from slots.

Tasks become ready to be executed according to a time criterion or a slot criterion. In the former case, a task becomes ready to be executed periodically, after a user-defined period of time elapses since it became ready for the last time; in the latter case, it becomes ready every time there is a new message available in every input slot. Note that becoming ready for execution just implies that the task is flagged so that the Runtime System can assign a thread to execute it; this does not entail that the task produces a set of outbound messages, but that it can examine its input slots and perform an action if the appropriate messages are found. For instance, a merger is a task that reads messages from two or more slots and merges them into one slot; this task can transfer messages as they are available. Contrarily, a context-based content enricher is a task that reads a base message and a context message from two different slots and uses the later to enrich the former; note that such a task cannot become ready to perform its enrichment action until the base and the context messages are simultaneously available.

Both slots and tasks are observable objects, which mean that they can notify other objects of changes to their state; in addition, tasks are observer objects since they monitor slots.

2.3. Ports

Ports abstract processes away from the communication mechanism in an inter-process communication or in the communication of the integration solution with an application, cf. Fig. 4.

Note that every port must be associated with a process, and that we distinguish between entry and exit ports. The former are ports that allow to read messages from an application or a process; the latter are ports that allow to write a message to a process or an application.

Internally, ports are composed of tasks and one of them must be a communicator. Communicators are the tasks that allow to actually read or write a message, namely: in communicators are used to read a message in raw form from a process or an application; contrarily,

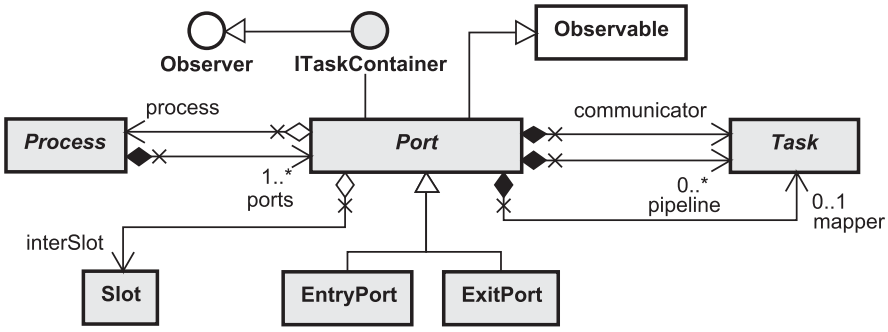

Fig. 4. Port model.

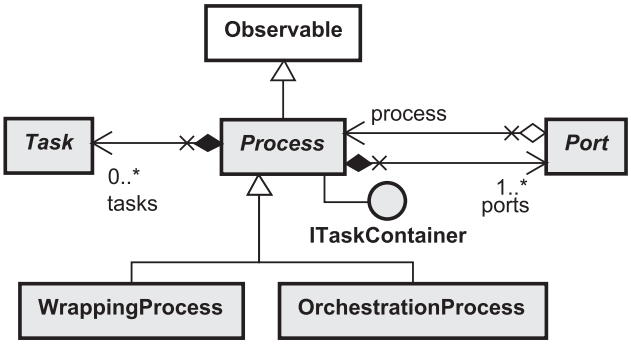

Fig. 5. Process model.

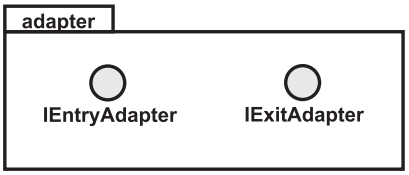

Fig. 6. Adapter model.

out communicators are used to write a message in raw form to a process or an application. By raw form, we mean a stream of bytes that is understood by the corresponding process or application. Inside ports, communicators interact with a pipeline of stream-oriented tasks, which also includes a mapper task. An in communicator passes every message read on to the pipeline; contrarily, an out communicator receives messages from the pipeline to write them. The pipeline is used as a pre-/post-processor that decrypts/encrypts, decodes/encodes, or unzips/zips this stream of bytes. The pipeline in an entry port ends with a mapper task that transforms the resulting stream of bytes into a message; the pipeline in an exit port begins with a mapper that transforms a message into a stream of bytes.

Note that ports also have a so-called inter-slot. We use this term to refer to the slots that allow the last task in an entry port to send messages to the first task in a process or the last task in a process to send messages to the first task in a port.

The ITaskContainer interface defines an interface every container of tasks must implement. It basically allows to add, remove, get, search, and count tasks. In addition, this interface extends the Observer Java interface so that a container centralises notifications received from its internal tasks. This feature is important because containers can then be notified about tasks that are ready to be executed. Not only implement ports the ITaskContainer, but they are also observable elements, i.e., they can both observe and produce notifications.

2.4. Processes

Processes are the central processing units in an integration solution, cf. Fig. 5. They are composed of ports and tasks, implement interface ITaskContainer, and extend class Observable. The reason why processes are observable is that they are just an abstraction that helps organise groups of tasks that co-operate to achieve a goal; from the point of view of the Guaraná SDK, they are just a container that reports which of their tasks are ready for execution to an external observer. A process may have several observers, e.g., to log or to monitor its activities; however, the most important one is a Runtime System, which we describe in the following section.

Processes serve two purposes, namely: there are processes that allow to wrap applications and processes that allow to orchestrate a workflow. The former are reusable processes that endow an application with a message-oriented API that simplifies interacting with it. Implementing such a wrapping process may range from using a JDBC driver to interacting with a database to implementing a scrapper that emulates the behaviour of a person who interacts with a user interface. Orchestration processes, on the contrary, are intended to orchestrate the interactions with a number of services, wrapping processes, and other orchestration processes. Independently from their role, processes are composed of ports and tasks.

2.5. Adapters

This package provides the foundations to implement adapters in specialised toolkits, cf. Fig. 6. An adapter is a piece of software that implements the low-level communication protocol that is necessary to interact with the processes or applications involved in an integration solution. A communication protocol may range from an RPC-based protocol over HTTP to a document-based protocol implemented on a database management system. Communicators rely on adapters to carry out their task. Whereas *in*-communicators have to use adapters that conform to the IEntryAdapter interface, *out*-communicators have to use adapters that conform to the IExitAdapter interface. These interfaces are provided by the framework layer and describe the operations used to read and write messages in raw form. The former interface specifies a read() operation that returns a Message that wraps the data to be manipulated in an integration solution, and the latter specifies a write() operation that takes a Message as input and writes it to a process or an application.

2.6. The Runtime System

The model of our Runtime System is presented in Fig. 7. Scheduler is the central class since its objects are responsible for coordinating all of the activities in an instance of our Runtime System. Note that this class is not a singleton since we do not preclude the possibility of running several instances concurrently. At runtime, a scheduler owns a work queue, a list of workers, and three monitors.

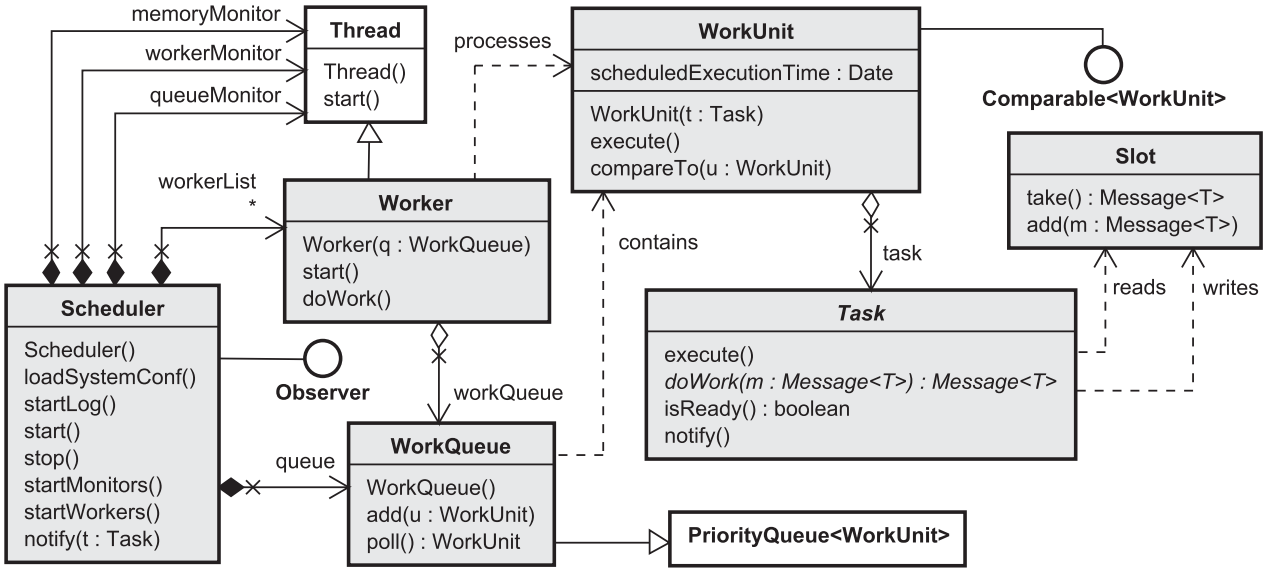

Fig. 7. Task-based runtime model.

The work queue is a priority queue that stores work units to be processed. A work unit has a reference to a task and a scheduled execution time before which it cannot execute. Note that class Task is abstract, which means that our Runtime System is not bound with a particular set of tasks; this allows to create specific-purpose task toolkits that can be plugged into the Runtime System. Usually, the scheduled execution time of a work unit is set to the current time, which means that the corresponding task can execute as soon as possible; if it is set to a time in future, then the corresponding task is delayed until that time has elapsed. This is very useful to implement tasks that need to execute periodically, e.g., a communicator that polls an application every minute.

Class Worker extends the standard Thread class, i.e., objects of this class run autonomously. Each worker is given a reference to the work queue, from which they must concurrently poll work units to process.

The monitors gather statistics about the usage of the memory, the CPU, and the work queue. The memory monitor registers information about both heap and non-heap memory; the worker monitor registers the user and the system time worker objects have consumed; and, the queue monitor registers the size of the queue and the total number of work units that have been processed. Monitors were implemented as independent threads that run at regular intervals, gather the previous information, store it in a file, and become idle as soon as possible.

Schedulers are configured using a simple XML file with information about the number of workers, the files to which the monitors dump statistics, the frequency at which they must run, and the logging system used to report warnings and errors. Fig. 8 shows the sequence of operations involved in the initialisation of a scheduler. The first operation loads the configuration file and analyses it; then, the logging system is started, and a work queue is created.

Note that engines are not started when they are created. It is the user who must decide when to start them using the start operation. This operation causes the invocation of two other operations, namely: startMonitors and startWorkers. The former starts the monitors that have been activated in the configuration file, cf. Fig. 9, and the later creates and starts the workers.

Fig. 10 shows the sequence of operations required to create and start the workers. Note that they are started asynchronously by invoking operation start. The business logic of a worker is defined inside its doWork operation. This operation implements a loop that

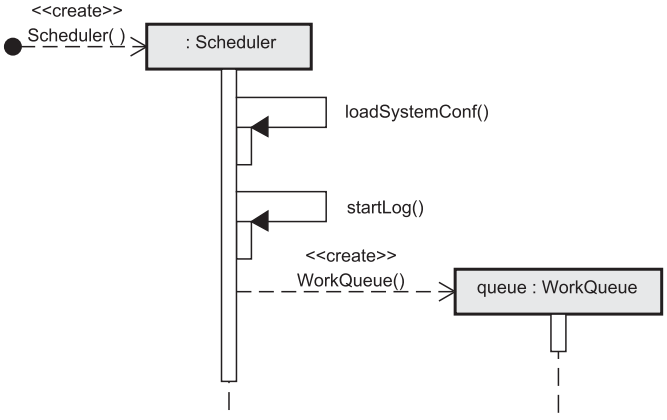

Fig. 8. Initialising the Runtime System.

enables the workers to poll the work queue as long as the scheduler is not stopped. When a work unit is polled, the worker first checks its scheduled execution time; if it has expired, then the task can be executed immediately; otherwise, the work unit is delayed until the deadline expires. Note that this strategy allows workers to keep working as long as there is a task ready to be executed.

Processing a work unit requires invoking operation execute on the associated task, which first packages the input messages and then invokes operation doWork, which depends completely on the task toolkit being used. Then, the task writes its output messages to the appropriate slot, which in turn notifies the tasks that read from them. These tasks then determine if they become ready for execution or not; in the former case, the tasks notify the container to which they belong. Containers of tasks propagate every notification they receive to the scheduler. For every task notification that the scheduler receives, it creates a new work unit and appends it to the work queue, cf. Fig. 11.

3. The general-purpose toolkit layer

The framework provides two extension points, namely: Task and Adapter. We have designed a core toolkit that provides extensions to deal with a variety of tasks that support the majority of integration patterns in the literature (Hohpe and Woolf, 2003), and provide

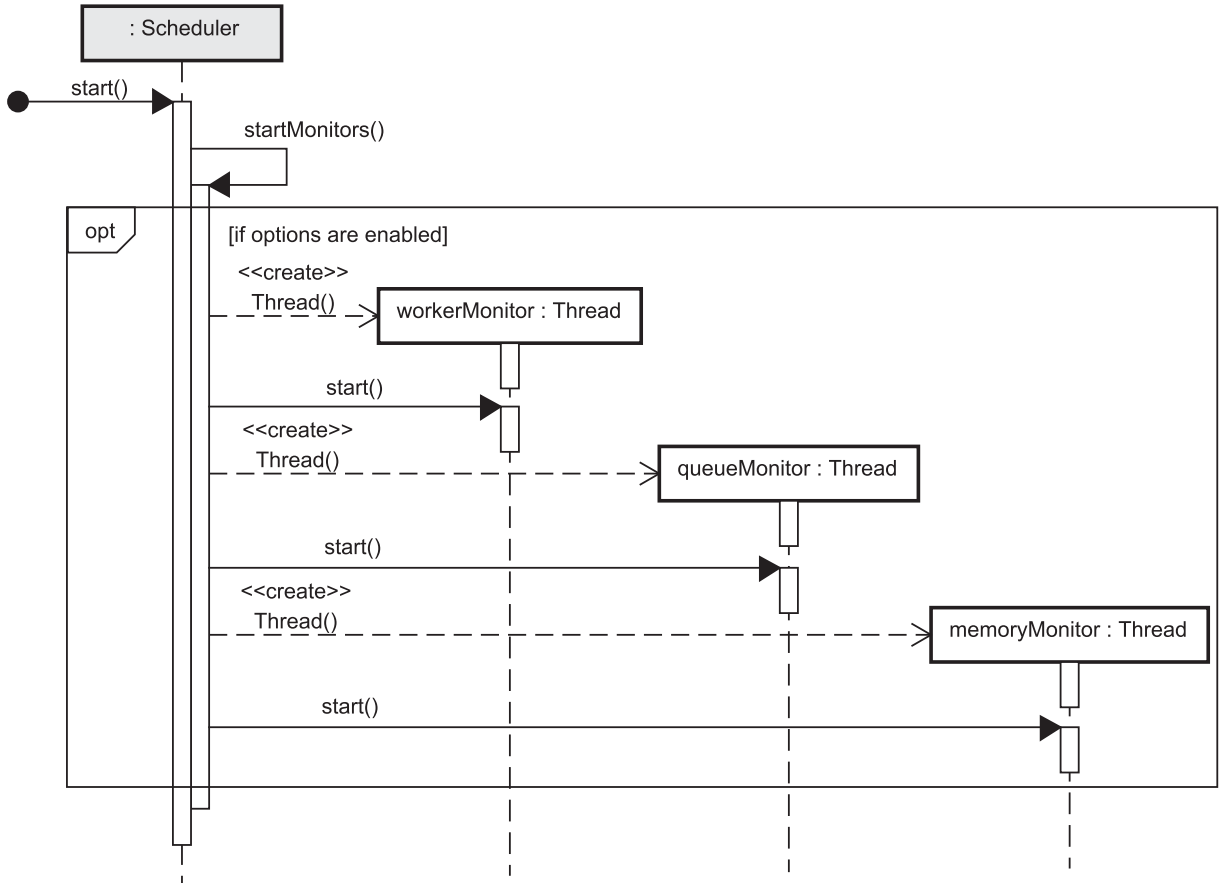

Fig. 9. Creating and starting monitors.

active and passive adapters that enable the use of several low-level communication protocols.

This toolkit provides extensions to the Task class, cf. Fig. 12. In the following descriptions we use term schema to refer to the logical structure of the body of a message. It may range from a DTD or an XML schema to a Java class. The first level of extension is composed of additional abstract classes that are intended to make it explicit several categories of integration patterns, namely:

- *Router*: a router is a task that does not change the messages it processes at all, but routes them through a process. This includes filtering out messages that do not satisfy a condition or replicating a message, to mention a few tasks in this category.
- *Modifier*: a modifier is a task that adds data to a message or removes data from it as long as this does not result in a message with a different schema. This includes enriching a message with contextual information or promoting some data to its headers, to mention a few examples in this category.
- *Transformer*: a transformer is a task that translates one or more messages into a new message with a different schema. Examples of these tasks include splitting a message into several ones or aggregating them back.
- *StreamDealer*: a stream dealer is a task that deals with a stream of bytes and helps zip/unzip, encrypt/decrypt, or encode/decode it.
- *Mapper*: a mapper is a task that changes the representation of the messages it processes, e.g., from a stream of bytes into an XML document.
- *Communicator*: a communicator is a task that encapsulates an adapter. Communicators serve two purposes: first, they allow adapters to be exported to a registry so that they can be accessed remotely; second, a communicator can be configured to poll periodically a process or application using an adapter.

There is a package associated with every of the previous tasks. They provide a variety of specific-purpose implementations in each integration pattern category (Frantz et al., 2011).

In the previous section, we mentioned that ports use communicators to communicate with other processes or applications. As we mentioned before, they rely on adapters, which can be either active or passive, cf. Fig. 13. An active adapter allows to poll the process or application with which it interacts periodically; contrarily, a passive adapter aims to export an interface to a registry, so that other applications or processes can interact with it. Note that entry and exit ports can be implemented using either active or passive adapters.

The active package is divided into two packages to provide implementations that are based on the JBI and the RMI protocols, respectively. Note that supporting JBI adapters allows to plug Guaraná SDK into a variety of ESBs; for example, our reference implementation is ready to be plugged into Open ESB (Rademakers and Dirksen, 2009). This, in turn, allows Guaraná SDK processes to have access to a variety of applications in current software ecosystems, including files, databases, web services, RSS feeds, SMTP messaging systems, JMS queues, DCOM servers, and so on. The rmi package provides several implementations that are intended to be used to interact with an RMI-compliant server.

4. Experimental study

Camel, Spring Integration, and Mule are the most closely-related proposals. They are based on the catalogue of integration patters by Hohpe and Woolf (2003), and support the core concepts of pipes,

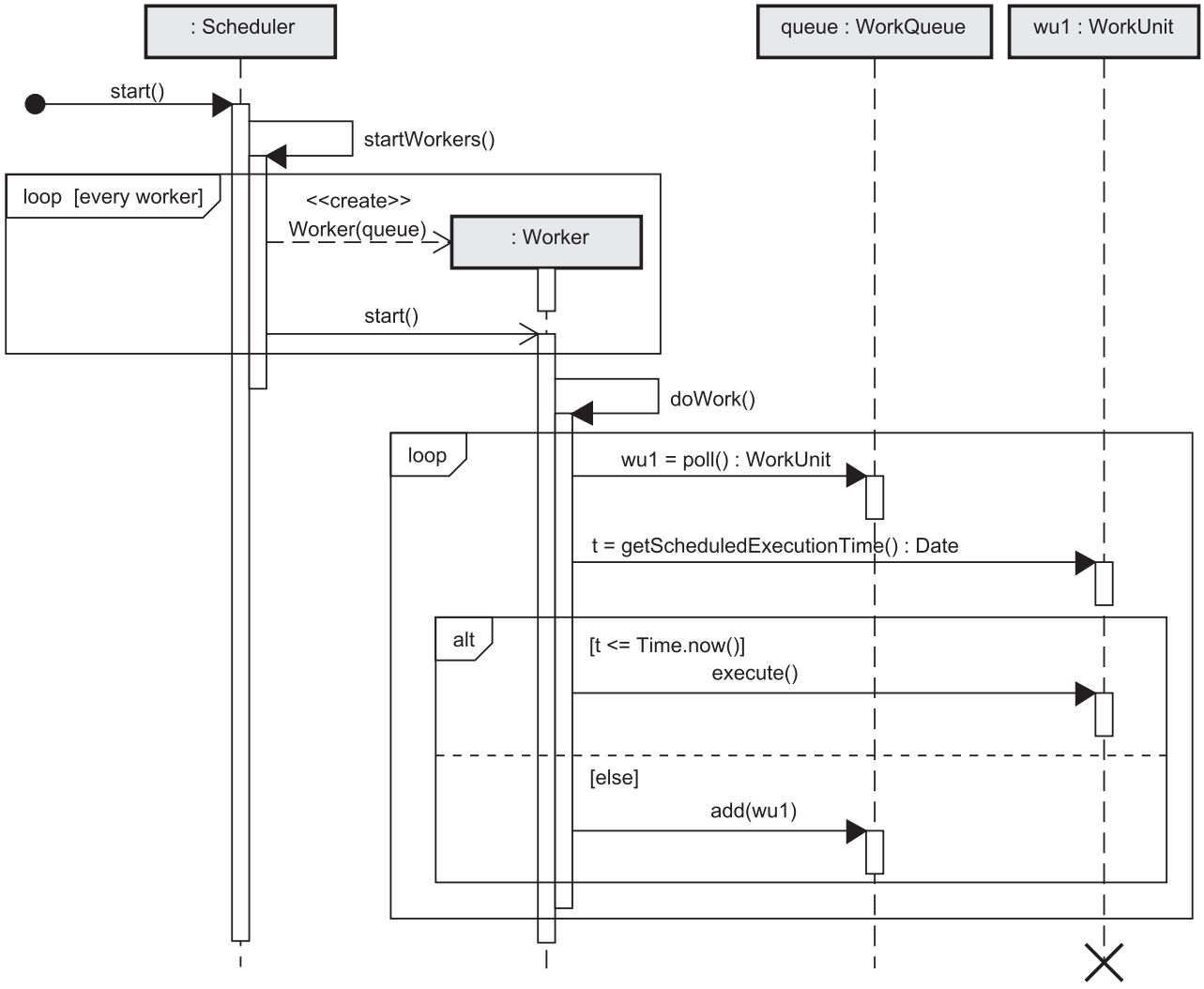

Fig. 10. Creating and starting workers.

filters, and resource adapters. These tools provide a graphical editor and an API that can be used to implement integration solutions at a high level of abstraction using the editor or at a low level of abstraction by coding integration solutions using the APIs.

Given two different software systems, the only totally accurate means to determine which one is more maintainable and adaptable is to use them in two projects in which software engineers with very similar skills are asked to maintain and adapt them for a particular purpose. Unfortunately, that does not make sense in an industrial environment because of the costs involved. This has motivated many researchers to devise measures that are correlated to the effort required to maintain and adapt a piece of software (Lanza and Marinescu, 2006; Lajios, 2009; Herraiz et al., 2009; Risi et al., 2013; Li and Henry, 1993; Sheldon et al., 2002; Bocco et al., 2005; Mouchawrab et al., 2005; Briand et al., 1998; Chidamber and Kemerer, 1994; Henderson-Sellers, 1996; Martin, 2002; McCabe, 1976). Many of which have been validated in real-world projects (Burger and Hummel, 2012; Mordal-Manet et al., 2013; Tempero et al., 2008; Balmas et al., 2009; Lanza and Marinescu, 2006). The conclusion is that these measures can be effectively used in practice to compare two software systems regarding maintainability and adaptability. In the following, we detail the experimental study we have conducted and that has demonstrated that Camel, Spring Integration, and Mule may have problems regarding maintenance. Section 4.1 introduces 25 maintainability measures from the literature; Section 4.2 presents the results for every maintainability measure regarding the analysed tools; and, Section 4.3 provides a statistical analysis on these results.

4.1. Evaluation measures

Since we are interested in how maintainable they are, we have used several measures to estimate maintainability that was proposed in the literature. The measures we have used in this article were classified either being related to Size, Coupling, Complexity, or Inheritance, based on those proposal by Lanza and Marinescu (2006). In the following we introduce these categories:

4.1.1. Size measures

The size of a software system is influenced by the number of packages, classes, interfaces, attributes, methods, and their parameters. The measures in this group allow to understand how big a software system is.

NOP: number of packages that contain at least one class or interface. This measure can be used as an indicator of how much effort is required to understand how packages are organised; note that this provides the overall picture of the design of a system (Dong and Godfrey, 2009). The greater this value, the more effort shall be required.

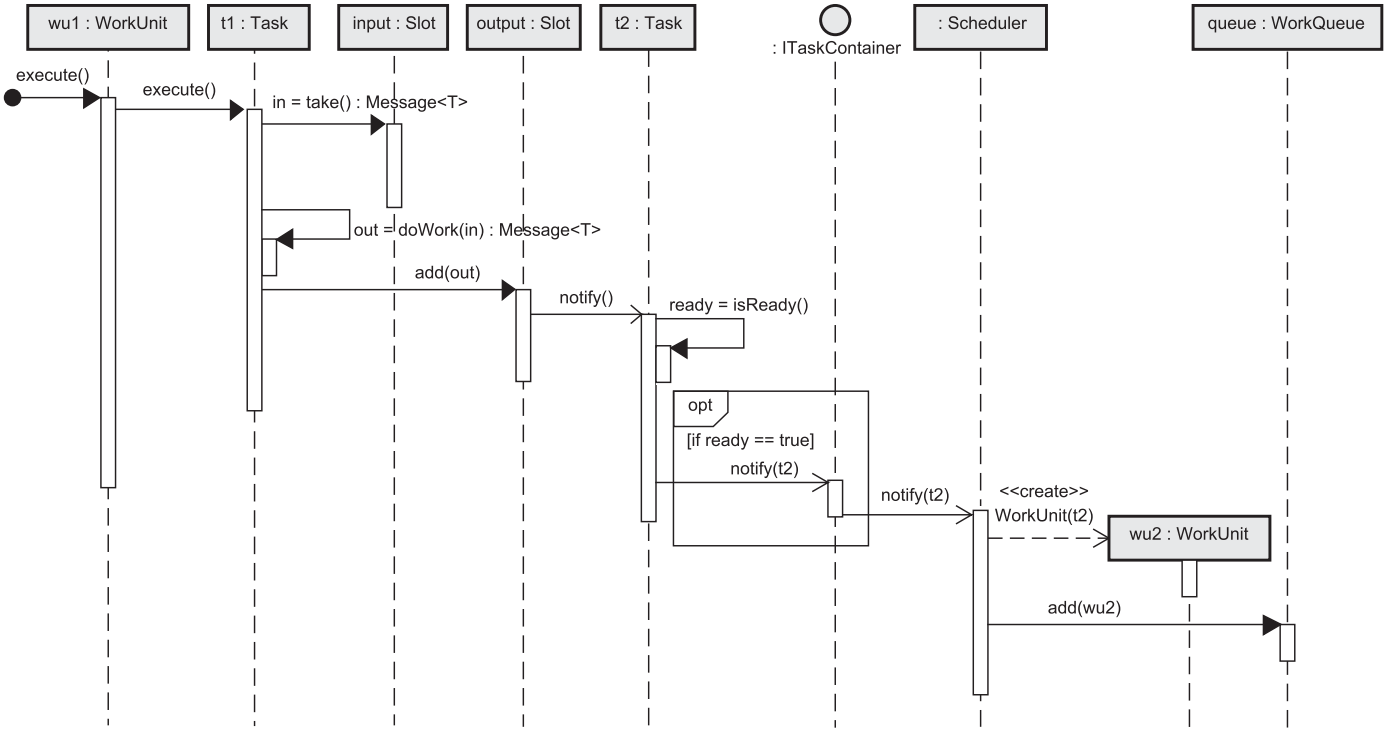

Fig. 11. Executing a work unit.

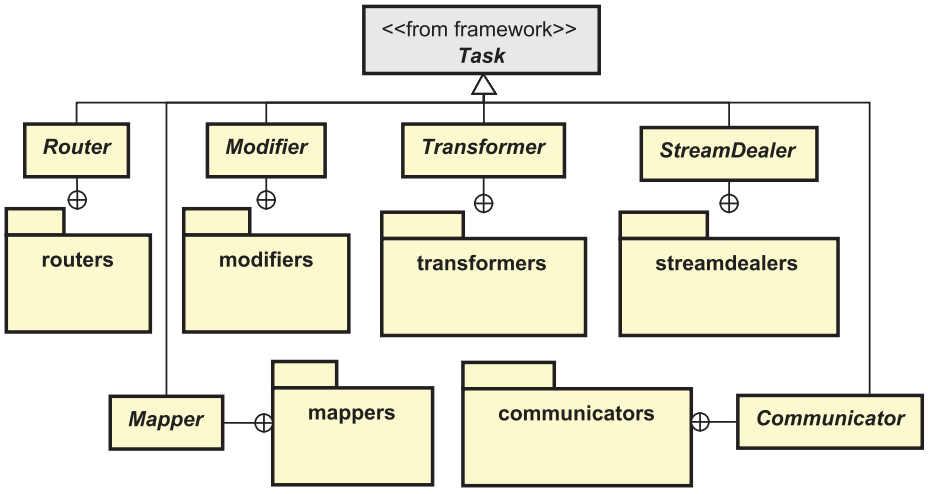

Fig. 12. Task model in the toolkit.

NOC: number of classes. This and the following measure (NOI) can be used as indicators of how much effort shall be required to understand the source code of a software system. The grater this value, the more difficult it is to understand a software system.

NOI: number of interfaces. It is commonly agreed that the lager the number of interfaces, the easier to adapt a software system.

LOC: number of lines of code, excluding blank lines and comments. In general, the greater this value, the more effort shall be required to maintain a software system.

NOM: number of methods in classes and interfaces. This measure can be used as an indicator for the potential reuse of a class. According to Lorenz and Kidd (1994), and Chidamber and Kemerer (1994), a large number of methods may indicate that a class is likely to be application specific, limiting the possibility of reuse.

NPM: number of parameters per method. This measure can be used as an indicator of how complex it is to understand and use a method. According to Henderson-Sellers (1996), the number of parameters should not exceed five. If it does, the author suggests that a new type must be designed to wrap the parameters into a unique object. The greater this value, the more difficult it is to understand a method.

MLC: number of lines in methods, excluding blank lines and comments. According to Henderson-Sellers (1996), this value should not exceed 50. If it does, the author suggests to split this method into other methods to improve readability and maintainability. The greater this value, the more difficult it is to understand and maintain a method.

NSM: number of static methods. This measure can be used as an indicator of how well implemented a piece of code is. The greater this value, the more likely that the code tends to be

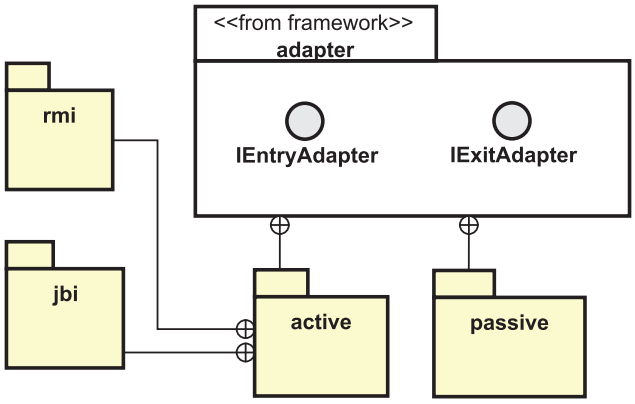

Fig. 13. Adapter model in the toolkit.

based on the classical procedural paradigm and not on the object-oriented paradigm.

NSA: number of static attributes. This measure can be used as an indicator of how difficult it is to reason about the state of a software system when testing. The greater this value, the more difficult testing.

NAT: number of attributes. This measure can be used as an indicator of how complex it is to understand a class. The greater this value, the more difficult it is to understand the state of a class.

4.1.2. Coupling measures

An important characteristic of the object-oriented paradigm is the encapsulation of data and the collaboration of objects to perform system functionalities. The measures in this group give an indication of how coupled the classes of a software system are.

LCM: lack of cohesion of methods. In this context, cohesion refers to the number of methods that share common attributes in a class. It is computed with the Henderson-Sellers (1996) LCOM∗ method. A low value indicates a cohesive class; contrarily, a value close to one indicates lack of cohesion and suggests that the class might better be split into two or more classes because there can be methods that should probably not belong to that class.

AFC: afferent coupling. This measure is defined as the number of classes outside a package that depend on one or more classes inside that package. The greater this value, the more complex maintenance becomes because there are more dependencies between classes (Martin, 2002; Offutt et al., 2008; Yu, 2008). Furthermore, larger values of afferent coupling can be used as an indicator that the package is critical for the software system and then maintenance in this package must be performed carefully not to introduce problems in the dependent classes.

EFC: efferent coupling. This measure is defined as the number of classes inside a package that depend on one or more classes outside the package. The greater this value, the more likely that maintenance shall have an impact on a package (Martin, 2002; Offutt et al., 2008; Yu, 2008).

FAN: number of called classes. This measure can be used as an indicator of how dispersed method calls are in classes of a software system (Lorenz and Kidd, 1994). The greater this value, the more complex is a method call because every call is supposed to involve other classes to be completed.

LAA: locality of attribute accesses. This measure can be used as an indicator of how dependent the methods of a class can be regarding the attributes of other classes. The greater this value, the more a method of a class uses attributes from other classes.

CDP: coupling dispersion. This measure can be used as an indicator of bad method design, since a method may be executing more than one thing and then can be split reducing its coupling. The greater this value, the more likely that there is an improper distribution of functionality amongst the methods of a software system.

CIT: coupling intensity. This measure can be used as an indicator of how dependent a method is, since it measures the number of distinct methods that are called by the measured method. The greater this value, the more likely there is an excessive coupling amongst the methods of a software system.

4.1.3. Complexity measures

The notion of complexity is important in software systems, chiefly if the software has to be maintained. The measures in this group allow to understand how complex a software system is.

ABS: degree of abstractness of a software system. This measure can be used as an indicator of how customisable a software system is (Martin, 2002). The greater this value, the easier to customise the software system.

WMC: weighted sum of the McCabe cyclomatic complexity (McCabe, 1976) for all methods in a class. This measure can be used as an indicator of how difficult understanding and then modifying the methods of a class shall be (Chidamber and Kemerer, 1994). The greater this value, the more effort is expected to maintain a class.

MCC: the McCabe cyclomatic complexity. This measure can be used as an indicator of how complex the algorithm in a method is. According to McCabe (1976), this value should not exceed ten. The greater this value, the more difficult it is to maintain a piece of code.

WOC: weight of class. This measure indicates the ratio of accessor methods regarding other methods that provide services (Marinescu, 2002). The greater this value, the more class interfaces consist of accessor methods, which indicates that classes are not too complex.

DBM: depth of nested blocks in a method. This measure can be used as an indicator of how expensive debugging a piece of code can be. According to Henderson-Sellers (1996), this value should not exceed five. If it does, the author suggests that the method should be broken into other methods. The greater this value, the more complex an algorithm is.

4.1.4. Inheritance measures

A well-known characteristic in the object-oriented paradigm is code reuse by means of the inheritance of functionalities amongst classes. Measures in this group allow to understand how much and how well the concept of inheritance is used in a software system.

DIT: depth of inheritance tree. Inheritance is a mechanism that increases core reuse (Alkadi and Alkadi, 2003). This measure can be used as an indicator of how complicated maintaining a class can be. The greater this value, the more difficult to maintain a software system.

NOH: number of immediate children classes of a class. This measure can be used as an indicator of the potential impact that a class may have in a software system if it is modified (Chidamber and Kemerer, 1994). The greater this value, the greater the chances that the abstraction defined by the parent class is poorly designed.

NRM: number of overriden methods. This measure can be used to indicate how adaptable a class is with respect to its ancestors (Lorenz and Kidd, 1994).The greater this value, the more likely that the inheritance mechanism is being used to adapt a class instead of just providing additional services to the parent class.

Table 1
Maintainability measures of Camel, Spring Integration, Mule, and Guaraná SDK.

	Measure	Camel				Spring Integration				Mule				Guaraná			
		Total	Mean	Dev.	Max	Total	Mean	Dev.	Max	Total	Mean	Dev.	Max	Total	Mean	Dev.	Max
Size	NOP	54	–	–	–	32	–	–	–	124	–	–	–	18	–	–	–
	NOC	730	13.52	19.55	96	269	8.41	10.52	58	733	5.91	7.40	51	79	4.39	3.09	11
	NOI	140	2.59	9.07	58	40	1.25	1.84	9	209	1.69	3.28	18	9	0.50	0.76	2
	LOC	62,439	–	–	–	14,929	–	–	–	67,090	–	–	–	2,878	–	–	–
	NOM	7,015	9.61	15.36	192	1,431	5.32	5.60	39	5,158	7.04	10.23	129	369	4.67	4.61	24
	NPM	–	0.93	1.05	11	–	1.13	0.94	9	–	0.92	1.07	19	–	1.20	1.04	4
	MLC	34,839	4.52	8.15	141	8,264	5.65	9.59	110	35,989	6.16	10.99	180	1,748	4.72	6.43	54
	NSM	709	0.97	4.95	74	31	0.12	0.73	8	686	0.94	9.41	244	1	0.01	0.11	1
	NSA	291	0.40	1.07	16	109	0.41	1.52	13	669	0.91	3.66	81	30	0.38	1.33	10
	NAT	1803	2.47	4.17	62	474	1.76	2.51	16	1417	1.93	3.21	31	87	1.10	2.14	12
Coupling	LCM	–	0.29	0.35	1	–	0.22	0.33	0.94	–	0.23	0.34	1.33	–	0.14	0.27	0.91
	AFC	–	30.63	89.34	542	–	12.69	26.65	146	–	22.90	56.25	493	–	6.94	14.33	47
	EFC	–	12.54	17.83	87	–	8.44	9.84	55	–	6.22	6.76	38	–	4.17	2.81	11
	FAN	3,637	3.74	–	74	642	1.73	–	40	3,765	3.60	–	65	175	1.54	–	11
	LAA	7,280.08	0.97	–	1	1,421.11	0.98	–	1	6,254.97	0.98	–	1	430.44	0.95	–	1
	CDP	874.74	0.12	–	1	124.60	0.09	–	1	940.01	0.15	–	1	35.40	0.08	–	1
	CIT	2,320	0.31	–	35	255	0.18	–	19	2,273	0.35	–	30	74	0.16	–	6
Complexity	ABS	–	0.15	0.21	1	–	0.27	0.25	1	–	0.33	0.33	1	–	0.54	0.35	1
	WMC	12,903	17.68	27.37	346	2,628	9.77	11.27	68	10,537	14.38	21.92	262	498	6.30	6.30	37
	MCC	–	1.67	2.06	46	–	1.80	2.04	30	–	1.80	2.01	33	–	1.35	0.91	8
	WOC	581.22	0.60	–	1	178.43	0.48	–	1	658.82	0.63	–	1	74.10	0.65	–	1
	DBM	–	1.37	0.79	8	–	1.44	0.86	6	–	1.43	0.87	8	–	1.24	0.74	4
Inheritance	DIT	–	2.22	1.33	6	–	2.45	1.40	6	–	2.02	1.30	7	–	3.03	1.34	5
	NOH	493	0.68	3.77	69	147	0.55	1.54	11	337	0.46	1.82	28	59	0.75	2.05	10
	NRM	357	0.49	1.06	8	69	0.26	0.66	5	351	0.49	1.02	9	70	0.89	1.01	3

Legend: NOP = Number of packages; NOC = Number of classes; NOI = Number of interfaces; LOC = Number of lines of code; NOM = Number of methods in classes and interfaces; NPM = Number of parameters per method; MLC = Number of lines in methods, excluding blank lines and comments; NSM = Number of static methods; NSA = Number of static attributes; NAT = Number of attributes; LCM = Lack of cohesion of methods; AFC = Afferent coupling; EFC = Efferent coupling; FAN = Number of called classes; LAA = Locality of attribute accesses; CDP = Coupling dispersion; CIT = Coupling intensity; ABS = Degree of abstractness of a software system; WMC = Weighted sum of the McCabe cyclomatic complexity for all methods in a class; MCC = The McCabe cyclomatic complexity; WOC = Weight of class; DBM = Depth of nested blocks in a method; DIT = Depth of inheritance tree; NOH = Number of immediate children classes of a class; NRM = Number of overriden methods.

4.2. Results of the analysis

We have conducted an experimental study in order to compute the maintainability measures regarding the core implementation of Camel, Spring Integration, and Mule, i.e., we do not take into account the code required to implement the adapters that are required to interact with the applications being integrated. We do not consider this code because it is peripheral and, more often than not, comes from other open-source projects that are maintained separately and then the comparison would be totally unfair. The core implementation of these proposals is comparable because they provide similar functionalities, which aim at providing support for the integration patterns documented by Hohpe and Woolf (2003). Table 1 summarises the results and compares them with the results for core implementation of Guaraná SDK.

The architecture of the tools we have analysed is organised into several packages: 54 in Camel, 32 in Spring Integration, and 124 in Mule. Although Mule has more than double as many packages as Camel, they have approximately the same total number of classes in their packages. Nevertheless, there are cases in which the maximum number of classes in a package reaches 96 in Camel, 58 in Spring Integration, and 51 in Mule. These values show that Camel has almost double as many classes in a package as Spring Integration or Mule. The same happens regarding the number of interfaces. Consequently, Camel has the highest standard deviation and mean values per package regarding both, classes and interfaces, which has an impact on the understandability of its packages. Spring Integration is the only tool that has a low value for the standard deviation regarding the number of interfaces. The architecture of Guaraná SDK is organised into 18 packages, and the maximum number of classes in a package is no more than 11. Furthermore, Guaraná SDK provides no more than 9 interfaces in these packages. The standard deviation computed for the number of classes and interfaces per package is very low, 3.09 and 0.76, respectively. These values indicate that maintenance in Guaraná SDK is not expected to be difficult.

Other values that are impressive for these tools are regarding the total number of lines of code, which is very high in every tool, chiefly for Camel and Mule. These tools are 62,439 and 67,090 lines of code respectively, contrarily to 14,929 in Spring Integration. The implementation of Guaraná SDK has a total number of 2,878 lines of code, which represents a big difference compared with the other tools. When analysing the methods in classes and interfaces, we found that Camel has 7,015 methods compared to the 1,431 and the 5,158 found for Spring Integration and Mule, respectively. Most probably, the difference amongst Spring Integration and the other tools is because it has less than a half the number of classes and interfaces of Camel and Mule. The values that stand out are the maximum number of methods per class/interface computed in Camel and Mule, which are 192 and 129 respectively, contrarily to 39 in Spring Integration. Guaraná SDK has 369 methods in total, with a maximum number of 24 methods per class/interface. If we look at the maximum number of parameter per method, it is also impressive how large it is, chiefly in Camel and Mule: 11 and 19 respectively. Spring Integration has a maximum of 9 parameters. These values indicate that some classes in Camel, Spring Integration, and Mule, are likely too application specific, with a limited possibility to be reused; furthermore, this makes some of their methods difficult to understand, chiefly in the case of Camel and Mule. Guaraná SDK has no more than 4 parameters per method, which indicates that classes in Guaraná SDK are expected to be more reusable and its methods not so difficult to understand. Counting the number of lines of code inside methods, we found Camel has a total number of 34,839, Spring Integration has 8,264, and Mule has 35,989, which if compared to the total number of lines of code, represents 0.55%, 0.55%, and 0.53% of these values, respectively. It means there are many attributes declared in classes. The maximum value computed demonstrate that there are some methods

with until 141 lines of code in Camel, 110 in Spring Integration, and 180 in Mule. These values indicate more effort might be necessary to maintain and understand the methods in these tools. Guaraná SDK has a total number of 1,748 lines of code inside methods, which, if compared to its total number of lines of code, represents 0.61% of this value. Furthermore, there is no method with more than 54 lines of code, being the average 4.72 lines of code per method. These values indicate that our classes are expected to be easier to understand and maintain.

If we look at the number of static methods, Camel and Mule have a similar mean value per class 0.97 and 0.94 respectively. Contrarily, Spring Integration has a mean of 0.12 static methods per class. The difference between these tools is more evident when looking at the maximum number of static methods in a class. Whereas Camel and Mule have 74 and 244 respectively, Spring Integration has 8. In Guaraná SDK these values are incredible low; the maximum number of static methods is no more than 1, and the mean value is 0.01, thus indicating the code follows correctly the object-oriented paradigm. Considering the number of static attributes, there is also a big different amongst the analysed tools. Mule has an impressive number of 669 static attributes in total, whereas Camel and Spring Integration have 291 and 109, respectively. Such values indicate it must be difficult to reason about the state of these tools when testing has to be performed. Contrarily, Guaraná SDK has only 30 static attributes in total, which indicates reasoning about its state shall be easier. Regarding the number of attributes, the total values for Camel and Mule are still very high, 1,803 and 1,417, respectively. These values correspond to a mean of 2.47 and 1.93 attributes per class, reaching Camel the impressive number of 62 attributes in a class. Spring Integration has a total of 474 attributes, a mean of 1.76, and no more than 16 attributes in a class. In Guaraná SDK the total number of attributes is 87, which corresponds to a mean of 1.10 attributes per class, suggesting it is not complex to understand the state of its classes as it can be in Camel, Spring Integration, and Mule.

The mean and the maximum values for the lack of cohesion of methods are similar in every tool. Camel has 0.29 and 0.35, Spring Integration has 0.22 and 0.33, and Mule has 0.23 and 0.34. In Guaraná SDK, the lack of cohesion of methods is very low, it presents a mean of only 0.14. Regarding the coupling of classes, the values for the afferent and efferent coupling in every tool are very high. Camel has the highest value for the afferent coupling, followed by Mule and then Spring Integration, with a mean of 30.63, 12.69, and 22.90, respectively. It is also very impressive the standard deviation, chiefly for Camel and Mule, which are 89.34 and 56.25, respectively. The maximum values are also very high, being 542 for Camel, 146 for Spring Integration, and 493 for Mule. These values suggest that much attention must be paid when performing maintenance in the classes of a package. The mean for the efferent coupling varies from 12.54 in Camel and 8.44 in Spring Integration, to 6.22 in Mule. The maximum values are not as impressive as the afferent coupling, but they are still very high. In Camel, the maximum efferent coupling is 87; in Spring Integration, it is 55; in Mule, it is 38. These figures suggest that the classes inside a package have a large number of dependencies on outside classes and maintenance has to be done carefully; as a conclusion, the impact on maintenance should not be neglected at all. Regarding the coupling of classes, the values for the afferent and efferent coupling in Guaraná SDK are not very high. The afferent coupling has values 6.94, 14.33, and 47 as mean, standard deviation, and maximum, respectively. The efferent coupling has values 4.17, 2.81, and 11 as mean, standard deviation, and maximum, respectively. The average afferent and efferent coupling in Guaraná SDK are 15.13 and 4.90 less than in other software tools, respectively. These values suggest that the classes in Guaraná SDK do not have a high number of dependencies and maintenance is expected to be easy.

Considering the number of called classes, once more Camel and Mule have very high values in total, compared to Spring Integration, 3,637, 3,765, and 642 respectively. If we look at the maximum number of calls a class receives, Camel has 74, Spring Integration 40, and Mule 65. In Guaraná SDK the total number of called classes is 175 and the maximum number of calls a class receives is no more than 11. These values indicate that method calls in Guaraná SDK are not complex. The locality of attribute accesses is similar in every tool. If we consider the mean value, Camel, Spring Integration, and Mule have 0.97, 0.98, and 0.98, respectively. The mean in Guaraná SDK is lower, 0.95. Regarding the coupling dispersion, the mean value indicates that Mule has the highest dispersion with 0.15, followed by Camel and Spring Integration, respectively with 0.12 and 0.09. Mule has also a very high value in total, 940.01, compared to Camel and Spring Integration with 874.74 and 124.60, respectively. These values indicate that chiefly Mule has an improper distribution of functionality amongst its methods. The mean value in Guaraná SDK is 0.08, which situates it close to Spring Integration. If we look at the maximum values for the coupling intensity of these software tools, these values demonstrate an excessive coupling amongst the methods in these tools, since the values in Camel, Spring Integration, and Mule are 35, 19, and 30, respectively. Contrarily, in Guaraná SDK the maximum value is 6, which indicates a low coupling amongst its methods.

The values for the degree of abstractness indicate that Camel is the less abstract tool. The mean value for Camel is 0.15, followed by 0.27 for Spring Integration, and 0.33 for Mule. The results indicate that these tools are not so easy to customise, chiefly Camel because its mean value is very low. The degree of abstractness in Guaraná SDK is very high. Its mean value is 0.54, which situates Guaraná SDK 0.29 in average more abstract than the other software tools. These values suggest that it shall not be complicated to customise Guaraná SDK. The weighted method complexity computed also demonstrates a high cyclomatic complexity within classes, chiefly for Camel and Mule. In these tools, the total weighted method complexity was 12,903 and 10,537, respectively. For Spring Integration, the cyclomatic complexity is 2,628, which is not so high when compared to Camel and Mule. Nevertheless, not only the total cyclomatic complexity is high, but also the mean, the standard deviation, and the maximum. Camel, Spring Integration, and Mule have maximum values of 346, 68, and 262, respectively. In Guaraná SDK, the total value is 498, the mean and the standard deviation were 6.30, and the maximum is 37. These values indicate a low cyclomatic complexity within the classes of Guaraná SDK. The values computed for the McCabe cyclomatic complexity indicate that there are cases in which it is extremely high. This is indicated by the maximum values, which reach 46, 30, and 33 in Camel, Spring Integration, and Mule, respectively. Consequently, they are also very complex tools, which may have a serious impact on their maintenance. The values computed for the McCabe cyclomatic complexity have indicated that the maximum value in Guaraná SDK is 8, which situates it with 28.33 less complexity than other software tools. These values indicate the architecture in Guaraná SDK is well designed and maintenance is expected to be easy.

The mean value for the weight of class indicates that classes in Spring Integration are complex. The mean value for Spring Integration is 0.48, followed by 0.60 for Camel, and 0.63 for Mule. In Guaraná SDK, the mean value is 0.65, which indicates that classes in Guaraná SDK are not too complex. The depth of nested blocks in a method is similar in every tool. If we consider the mean and maximum values, Camel has 1.37 and 8, Spring Integration has 1.44 and 6, and Mule has 1.43 and 8, respectively. In Guaraná SDK, the mean and maximum values for the depth of nested blocks is 4 and 1.24, respectively. These values indicate debugging a piece of code in Guaraná SDK shall not be expensive as in the other tools.

The depth of inheritance tree in Mule counts for a maximum value of 7, which makes more complicated to maintain a class in this tool. Camel and Spring Integration have equal values, 6. In Guaraná SDK the maximum value is no more than 5. The maximum number of

Table 2
Empirical rankings.

Tool	Total	Mean
Guaraná SDK	1.56	1.24
Spring Integration	2.56	2.08
Camel	2.64	3.16
Mule	3.24	3.52

Table 3
Results of Iman–Davenport's test.

Test	Total	Mean
Statistic	9.84	44.18
p-Value	1.61E−5	3.33E−16

immediate children classes of a class also varies very much: 69 in Camel, 11 in Spring Integration, and 28 in Mule. If considered the mean and the standard deviation values per class, Camel has the highest values, which indicates that the abstraction defined by parent classes tend to be poorly designed. The maximum number of immediate children classes of a class in Guaraná SDK is no more than 10, with a mean of 0.75 per package. These values indicate that the abstraction defined by the parent class is well designed in Guaraná SDK. Regarding the number of overriden methods, Spring Integration has the lowest mean value amongst the analysed tools, and Camel and Mule have the same value, respectively with 0.26, 0.49, and 0.49. In Guaraná SDK the mean value is 0.89, which indicates classes in this tool are more adaptable than in Camel, Spring Integration, and Mule.

From the analysis of the maintainability measures, it follows that the tools we have analysed may have problems regarding maintenance, chiefly adaptive maintenance, which is our main concern in this article. The maintainability measures computed for Guaraná SDK provide better values, which suggests that our proposal is more maintainable and thus easier to adapt to a specific context than Camel, Spring Integration, or Mule.

4.3. Statistical analysis

We have first computed the values of the measures from the source code of each tool, and we have got the results in Table 1. We first have analysed if these values can be considered sampled from a Normal distribution using the Kolmorogov–Smirnov's test and the Shapiro–Wilk's test using the standard significance level $\alpha = 0.05$. The results of these tests prove that none of the measures can be considered to be distributed normally, which justifies to perform non-parametrical tests (Sheskin, 2012).

The steps to perform the non-parametrical tests were the following: (a) compute the rank of each technique from the evaluation results after normalising the corresponding measures to interval [0, 1]; (b) determine if the differences in ranks are significant or not using Iman–Davenport's test; (c) if the differences are significant, then compute the statistical ranking using Bergmann–Hommel's test on every pair of tool. We have performed the tests on both the totals and the mean values of the measures, and we have got similar results.

Table 2 shows the empirical rankings that we got; note that Guaraná SDK ranks the first regarding both the total and the mean values. Then we used Iman–Davenport's test to check if there are statistically significant differences in these ranks at the standard significance level ($\alpha = 0.05$). Table 3 shows the results; note that the *p*-Value is largely smaller than the standard significance level, which is a strong indication that the empirical ranks are different from a statistical point of view. As a conclusion, it makes sense to perform Bergmann–Hommel's test to rank every pair of proposals. Table 4 shows the results. Regarding the total measures, note that the comparisons of Guaraná SDK with the other techniques results in adjusted *p*-Values that are always significantly smaller than the significance level, which is a strong indication that Guaraná SDK's measures are better than the others; note, too, that the adjusted *p*-Values that corresponds to the remaining comparisons are not smaller than the significance level, which indicates that there are not significant differences amongst the measures of the other tools. Regarding the mean measures, the results are similar; the only difference is that Guaraná SDK is significantly better than Spring Integration, which, in turn, is significantly better than both Camel and Mule at the standard significance level.

Table 4
Results of Bergmann–Hommel's test.

Comparison	Statistic	*p*-Value	Tool	Rank
Mule vs. Guaraná SDK	4.60	2.52E-5	Guaraná SDK	1
Camel vs. Guaraná SDK	2.95	9.29E-3	Spring Integration, Camel, Mule	2
Spring Integration vs. Guaraná SDK	2.73	0.01	–	–
Spring Integration vs. Mule	1.86	0.18	–	–
Camel vs. Mule	1.64	0.18	–	–
Camel vs. Spring Integration	0.21	0.82	–	–

(a) Total values

Comparison	Statistic	*p*-Value	Tool	Rank
Mule vs. Guaraná SDK	6.24	2.56E-9	Guaraná SDK	1
Camel vs. Guaraná SDK	5.26	4.36E-7	Spring Integration	2
Spring Integration vs. Mule	3.94	2.40E-4	Camel, Mule	3
Camel vs. Spring Integration	2.96	3.10E-3	–	–
Spring Integration vs. Guaraná SDK	2.30	4.28E-2	–	–
Camel vs. Mule	0.98	3.24E-1	–	–

(b) Mean values

As a conclusion, we have proved that there is enough statistical evidence in the measures that we have collected to prove that Guaraná SDK outperforms the other proposals regarding maintainability.

5. An industrial experience

We have worked on an industrial experience in co-operation with an spin-off to evaluate the use of Guaraná SDK in industry, within the context of health care systems, to demonstrate the viability of our proposal. The industrial experience was designed using the domain-specific language introduced in Frantz et al. (2011) and implemented using Guaraná SDK. We have measured the effort required to develop the integration solution for this industrial experience and conducted a series of experiments to evaluate its performance on the Runtime System, which is part of the core implementation of Guaraná SDK. The industrial experience consists of a real-world integration problem that builds on a project to automate the registration of new users into a unique repository of the Huelva's County Council (Huelva, Spain). This repository contains information about users that comes from both a local application and a web portal. It is expected that every new user is notified and provided with his/her digital certificate by secure e-mail. In the following sections, we first describe the integration problem tackled in the industrial experience; we then provide a solution model; we show the effort employed in the development of the integration solution; and, finally, we show the experimental results that we gathered.

5.1. The software ecosystem

The integration solution involves six applications, namely: Local Users, Portal Users, LDAP, Human Resources System, Digital

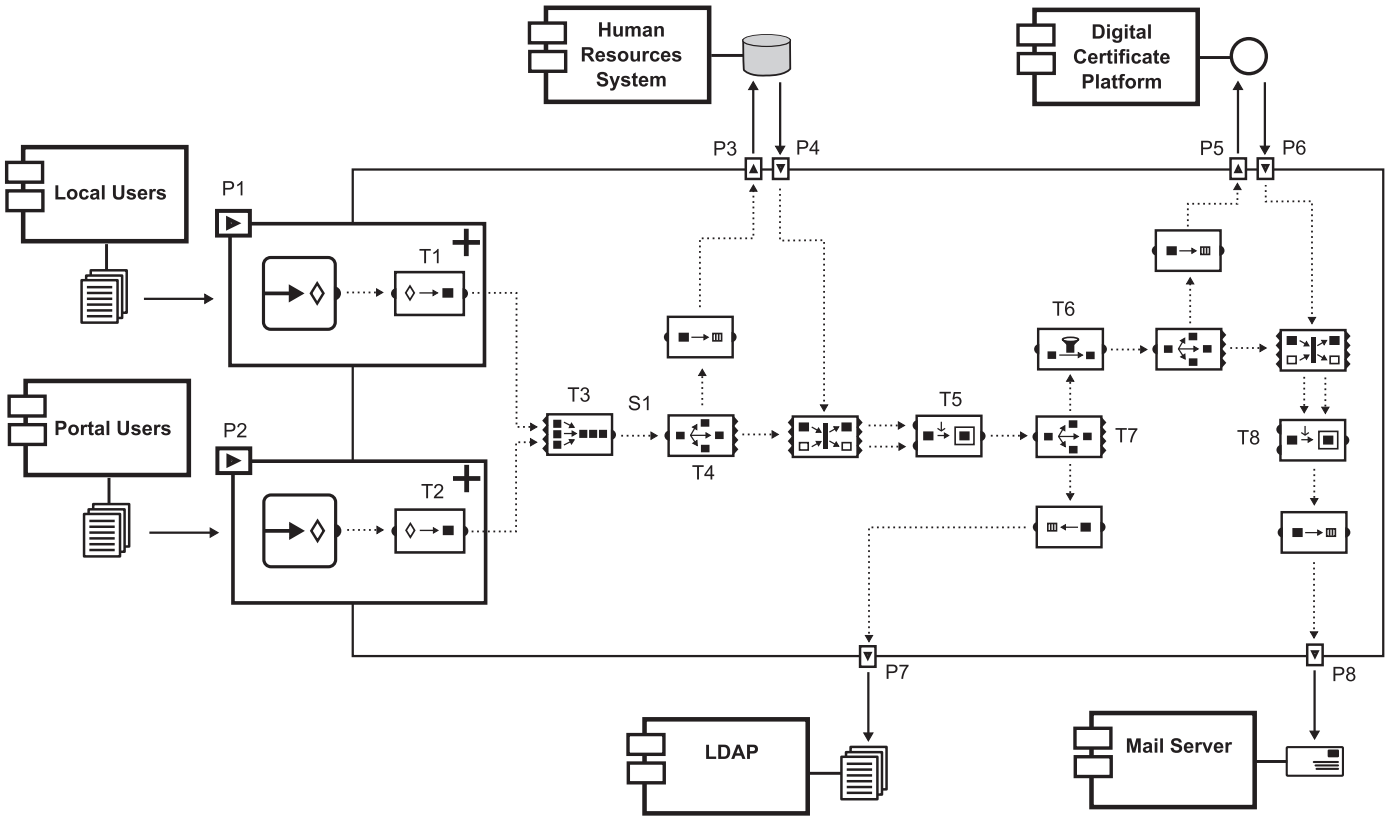

Fig. 14. The integration solution conceptual model.

Certificate Platform, and Mail Server. Each application runs on a different platform, and, except for the LDAP, the Digital Certificate Platform, and the Mail Server, they were not designed with integration concerns in mind.

The Local Users is the first application developed in house; it aims to manage the county council information systems' users. Note that, this is a standalone application and does not provide an authentication service. The Portal Users is an off-the-shelf application that the web portal uses to manage its users. In addition, a unique repository for users has been set up using an LDAP-based application, so that it can provide authentication access control for several other applications inside the software ecosystem. The Human Resources System is a legacy system developed in house to provide personal information about the employees. It is a part of the integration solution since we require information like name and e-mail to compose notification e-mails. Another application developed in house is the Digital Certificate Platform, which aims to manage digital certificates; it was designed with integration concerns in mind. Amongst other services, this application can be queried to get a URL that temporarily points to a digital certificate that users can download after authenticating. Finally, the Mail Server runs the Council's e-mail service, which is used exclusively for notification purposes.

5.2. Integration solution

The integration solution we have devised is composed of one orchestration process that exogenously co-ordinates the applications involved in the integration solution, cf. Fig. 14. Some ports use text files to communicate with Local Users, Portal Users, and LDAP; the Human Resources System is queried by means of its database management system; and, the communication with the Digital Certificate Platform and the Mail Server is performed by means of APIs. Translator tasks were used to translate messages from canonical schemas into the schemas with which the integrated applications work.

The workflow begins at entry ports P1 and P2, which periodically poll the Local Users and Portal Users logs to find new users. Every port is provided with only a communicator task, except for ports P1 and P2 that also have a mapper task. In both ports, every user record results in a message that is added by the communicators to their corresponding slots. The body of the message holds the data that has been polled as a stream. Thus, mappers T1 and T2 map the inbound messages onto outbound messages that conform to a canonical XML schema that represents user records. Inside the process, task T3 gets messages coming from both ports and adds them to slot S1. Replicator task T4 creates two copies of every message it gets from this slot, so that one copy can be used to query application Human Resources System by means of ports P3 and P4, for information about the employee who owns a user record. Next, task T5 enriches the other correlated copy with the information returned by the Human Resources System and then task T7 replicates this enriched message with copies to the LDAP and the Digital Certificate Platform. The new user record is written to the LDAP by means of exit port P7. Before querying the Digital Certificate Platform, task T6 filters out messages that do not include an e-mail address. Messages that go through task T8, which enriches them with the corresponding certificate. Finally, exit port P8 communicates with the Mail Server application to send the certificate and notify the employee about his/her inclusion in the LDAP.

5.3. Development effort

We have used six measures to empirically estimate the amount of effort involved in the development of the proposed integration solution. A software engineer from the partner spin-off was randomly selected, amongst seven candidates, to develop the proposed

Table 5
Effort to develop the integration solution.

Measure	Value
Time to study the integration problem	15 min
Time to design the integration solution	32 min
Time to design the message schemas	13 min
Time to implement the design of integration solution	87 min
Number of bugs detected	4
Number of modifications in the design	4

industrial experience. Every engineer candidate had more than one year experience in developing integration solutions using the integration patterns documented by Hohpe and Woolf (2003). The design was assisted by a graphical editor and the implementation was carried out by coding the integration solution using the command-query API provided by Guaraná SDK. We measured the following variables:

Time to study the integration problem. Measures the total time an engineer has spent to understand the integration problem. In this activity, the applications involved, their communication layer, and the data they share with the integration solution must be identified.

Time to design the integration solution. Measures the total time an engineer has spent to produce a complete and ready to implement design of an integration solution for an integration problem.

Time to design the message schemas. Measures the total time an engineer has spent to create the XML schemas used to represent the information with which and integration solution deals.

Time to implement the design of integration solution. Measures the total time an engineer has spent to produce the code that implements the design of an integration solution; this time includes the time to test it and correct bugs, besides the time to configure its binding components.

Number of bugs detected. Registers the number of errors detected and corrected by the engineer during the development of an integration solution.

Number of modifications in the design. Registers the number of times an engineer made important modifications to the design of an integration solution (e.g., new tasks were added or removed to/from the design or data schemas were modified) after the first version was produced.

Table 5 presents the values obtained for each measure. The times spent to study the integration problem and to design the message schemas were quite short. The time to design the integration solution was expected to be shorter, since this activity was assisted by a design tool. However, due to changes in the data schemas, four modifications had to be done in the design, which had a negative impact on the time spent in the design of the integration solution. Otherwise, this time should be significantly reduced. The majority of the time was spent in the implementation. It was already expected because the implementation was carried out using a command-query API installed into the Eclipse IDE tool.

5.4. Experimental results

We have conducted a series of experiments to evaluate the integration solution on the Runtime System of Guaraná SDK. We used mock adapters, i.e., adapters implemented in memory that simulate the functionality of a real-world adapter and not on external software. This is a common feature provided by integration frameworks in their core implementation. The mock adapters allowed us to save the processing time required by the real-world adapters based on JBI. Furthermore, by using mock adapters the execution of the integration solution depends only on the core implementation of Guaraná SDK, and not on other external software. In each experiment we measured the following variables:

Consumption of CPU time per thread: We use this variable to measure the average CPU times that the integration solution has consumed to process all of the messages of an experiment. Note that we measured CPU time per thread, i.e., the actual time the available threads took to process the workload, including user and operating system time. To measure this variable, we run the integration solution with a fixed message production rate, a varying the number of threads (t), and a varying number of messages (m). We introduced a 60-second delay between every two experiments. The message production rate considered was one message every 5 milliseconds, we varied t in the range 1, 2, 4, 6, 8 threads, and m in the range 20,000, 40,000, 60,000, …, 200,000 messages. In total, we ran a total of 125 experiments for the integration solution to draw our conclusions on this variable.

Pending messages: This variable measures the number of messages that had not been processed yet right after the message production finishes. The experiments conducted to measure this variable consisted of running the integration solution with a fixed number of messages per experiment, a varying number of threads (t), and a varying message production rate (r) to simulate heavily-loaded scenarios. We introduced a 60-second delay between every two experiments. The total number of messages in each experiment was 100,000, we varied t in the range 1, 2, 4, 6, 8 threads, and r in the range 200, 400, 600, …3,000 messages per second. In total, we ran 375 experiments for the integration solution to draw our conclusions on this variable.

We ran these experiments on a machine that was equipped with an Intel Core i7 with four physical CPU threads that run at 2.93 GHz, and had 8GB of RAM, Windows 7 Professional Service Pack 1, and Java Enterprise Edition 1.6 64-bit installed. Each experiment was repeated 5 times and the results were averaged in order to diminish the effects of unpredictable events in the operating system. In every experiment the body of the messages hold an actual document in XML format. Note that the size of a message being processed by the integration solution varies, since it is modified and transformed throughout the workflow. Thus, we have computed the average size of the messages that belong to a same correlation processed in the integration solution. The result is an average message size of 1, 317.75 bytes for this industrial experience.

Fig. 15 presents our experimental results. The consumption of CPU time grows linearly as the number of messages m increases, independently from the number of threads t available. We performed a linear regression analysis and confirmed the previous claim since the values we got for the R^2 coefficient were 0.994, 0.994, 0.996, 0.996, and 0.997 for 1, 2, 4, 6, and 8 threads, respectively. The graph depicted for this variable shows that the consumption of CPU time per thread reduces considerably when adding more threads until the limit of 4 threads. This behaviour is attributed to the limit of four physical CPU threads in the processor. This explains why adding more threads to the integration solution, does not result in a significant reduction of the total CPU time per thread.

The graph depicted to show the number of pending messages, indicates that the integration solution supports a message production rate r until 800 messages per second when using 4, 6, or 8 threads, since there are not any pending messages when the message production finishes. A higher message production rate r causes the integration solution to accumulate messages, independently from the number of threads with which we have experimented. If the message production rate r ranges from 1,600–3,000, then there is not much difference in using 1 or 8 threads. With $r = 200$, messages do not accumulate even if running the integration solution with only 1 thread. If running the integration solution with 2 threads, no messages are

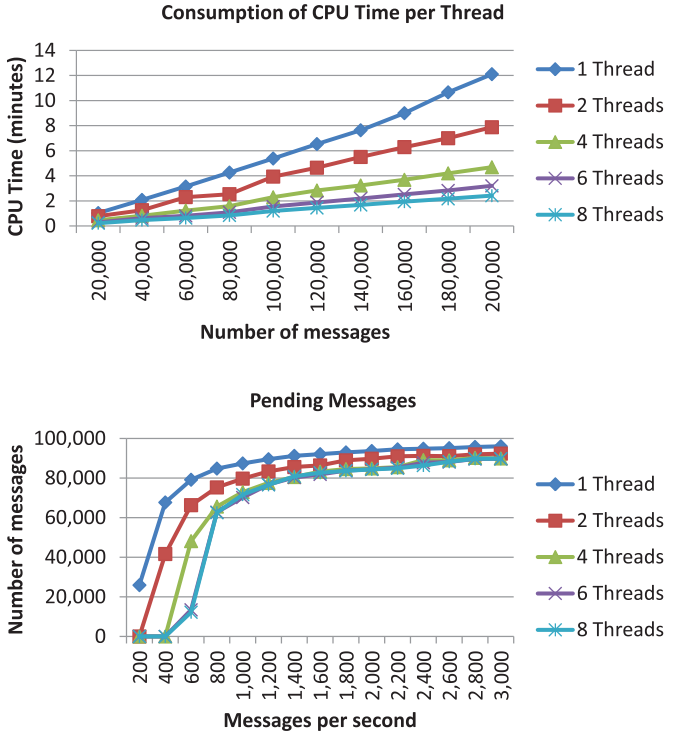

Fig. 15. Experimental results for the integration solution.

accumulated until $r = 600$. A similar behaviour when using 4–8 threads can be observed in this experiment, which is attributed to the limit of four physical CPU threads in the processor. Note that, despite the experiments have indicated a weak performance in scenarios with a workload around 500 messages per second, the Runtime System is able to handle a workload as high as 400 messages per second without getting collapsed.

6. Conclusions

Enterprise Application Integration is a corner-stoner for companies that aim at reusing the applications that are available within their software ecosystems to support and optimise their business processes. The catalogue of integration patterns proposed by Hohpe and Woolf (2003) was adopted by the Enterprise Application Integration community as a cookbook to design and implement integration solutions. Furthermore, Camel, Spring Integration, and Mule range amongst the most popular tools available to design and implement integration solutions building on Hohpe and Woolf's catalogue.

Companies that provide Enterprise Application Integration solutions are interested in software tools that can be easily adapted to focus on specific contexts. We have used 25 of the measures proposed by Lanza and Marinescu (2006); Lajios (2009); Herraiz et al. (2009); Risi et al. (2013); Li and Henry (1993); Sheldon et al. (2002); Bocco et al. (2005); Mouchawrab et al. (2005); Briand et al. (1998); Chidamber and Kemerer (1994); Henderson-Sellers (1996); Martin (2002); and McCabe (1976) to evaluate the maintainability of Camel, Spring Integration, and Mule. The results that we obtained indicate that adapting these software tools for particular contexts may be costly.

In this article we have presented Guaraná SDK, which is our software development kit to implement integration solutions. Guaraná SDK provides a number of classes and interfaces that implement the abstractions of the domain-specific language introduced by Frantz et al. (2011), which we have developed to design integration solutions building on integration patterns.

We have computed the maintainability measures regarding Guaraná SDK and the results suggest that maintaining our proposal is easier than maintaining Camel, Spring Integration, or Mule.

To confirm this findings we performed a statistical analysis based on Kolmorogov–Smirnov's test, Shapiro–Wilk's test, Iman–Davenport's test, and Bergmann–Hommel's test on the results obtained with the maintainability measures. As a conclusion we proved that there is enough statistical evidence in the measures that we have collected to prove that Guaraná SDK outperforms the other proposals regarding maintainability.

Our proposal was applied to a real-world project in industry in collaboration with a spin-off company. The integration solution in this industrial experience was designed using the domain-specific language introduced in a previous work (Frantz et al., 2011) and implemented using Guaraná SDK, and run under very high workloads in the Runtime System that is part of the core implementation of Guaraná SDK. Despite the experiments have indicated a weak performance in scenarios with a workload around 500 messages per second, the Runtime System is able to handle a workload as high as 400 messages per second without getting collapsed.

Acknowledgements

The research work which we report in this article was supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programmes (Grants TIN2007-64119, P07-TIC-2602, P08-TIC-4100, TIN2008-04718-E, TIN2010-21744, TIN2010-09809-E, TIN2010-10811-E, and TIN2010-09988-E). Rafael Z. Frantz was also supported by the Evangelischer Entwicklungsdienst e.V. (EED). We would like to thank company i2Factory, S.L. for trusting our results and developing a commercial version of Guaraná SDK.

References

Alkadi, G., Alkadi, I., 2003. Application of a revised dit metric to redesign an oo design. J. Object Technol. 2 (1), 127–134. doi:10.5381/jot.2003.2.3.a3.

Balmas, F., Bergel, A., Denier, S., Ducasse, S., Laval, J., Mordal-Manet, K., Abdeen, H., Bellingard, F., 2009. Software Metric for Java and C++ Practices (Squale Deliverable 1.1). Technical report. French Institute for Research in Computer Science and Automation.

Bergin, S., Keating, J., 2003. A case study on the adaptive maintenance of an Internet application. J. Softw. Maint. 15 (4), 254–264. doi:10.1002/smr.275.

Bocco, M.G., Piattini, M., Calero, C., 2005. A survey of metrics for UML class diagrams. J. Object Technol. 4 (9), 59–92. doi:10.5381/jot.2005.4.9.a1.

Briand, L.C., Daly, J.W., Wüst, J., 1998. A unified framework for cohesion measurement in object-oriented systems. Empirical Softw. Eng. 3 (1), 65–117. doi:10.1023/A:1009783721306.

Burger, S., Hummel, O., 2012. Applying maintainability oriented software metrics to cabin software of a commercial airliner. In: Proceedings of Conference on Software Maintenance and Reengineering. CSMR, pp. 457–460. doi:10.1109/CSMR.2012.58.

Chen, J.-C., Huang, S.-J., 2009. An empirical analysis of the impact of software development problem factors on software maintainability. J. Syst. Softw. 82 (6), 981–992. doi:10.1016/j.jss.2008.12.036.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object-oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493. doi:10.1109/32.295895.

Dong, X., Godfrey, M.W., 2009. Understanding source package organization using the hybrid model. In: Proceedings of International Conference on Software Maintenance, pp. 575–578. doi:10.1109/ICSM.2009.5306366.

Epping, A., Lott, C.M., 1994. Does software design complexity affect maintenance effort? In: Proceedings of NASA/Goddard 19th Annual Software Engineering Workshop, pp. 297–313.

Fowler, M., 2010. Domain-Specific Languages. Addison-Wesley.

Frantz, R.Z., Corchuelo, R., 2012. A software development kit to implement integration solutions. In: Proceedings of the 27th Symposium on Applied Computing, pp. 1647–1652. doi:10.1145/2245276.2232042.

Frantz, R.Z., Reina-Quintero, A.M., Corchuelo, R., 2011. A domain-specific language to design enterprise application integration solutions. Int. J. Cooperative Inf. Syst. 20 (2), 143–176. doi:10.1142/S0218843011002225.

García-Jiménez, F., Martínez-Carreras, M., Gómez-Skarmeta, A., 2010. Evaluating open source enterprise service bus. In: Proceedings of the IEEE 7th International Conference on e-Business Engineering, pp. 284–291. doi:10.1109/ICEBE.2010.12.

Henderson-Sellers, B., 1996. Object-Oriented Metrics, Measures of Complexity. Prentice Hall.

Herraiz, I., Izquierdo-Cortazar, D., Rivas-Hernández, F., 2009. Flossmetrics: free/libre/open source software metrics. In: Proceedings of Conference on Software Maintenance and Reengineering. CSMR, pp. 281–284. doi:10.1109/CSMR.2009.43.

HIPAA, 2011. Health Insurance Portability and Accountability Act Home. http://www.hipaa.com
HL7, 2011. Health Level Seven International Home. http://www.hl7.org
Hohpe, G., Woolf, B., 2003. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley.
IEEE, 1990. IEEE Standard Glossary of Software Engineering Terminology. IEEE Computer Society.
ISO/IEC, 2001. International Standard ISO/IEC 9126, Software engineering – Product Quality – Part1: Quality Model. Technical Report. International Standard Organization.
ISO/IEC, 2011. International Standard ISO/IEC 25010, Systems and Software Engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. Technical report. International Standard Organization.
Jorgensen, M., 1995. An empirical study of software maintenance tasks. J. Softw. Maint. 7 (1), 27–48. doi:10.1002/smr.4360070104.
Lajios, G., 2009. Software metrics suites for project landscapes. In: Proceedings of Conference on Software Maintenance and Reengineering. CSMR, pp. 317–318. doi:10.1109/CSMR.2009.22.
Lanza, M., Marinescu, R., 2006. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer.
Li, W., Henry, S.M., 1993. Object-oriented metrics that predict maintainability. J. Syst. Softw. 23 (2), 111–122. doi:10.1016/0164-1212(93)90077-B.
Lorenz, M., Kidd, J., 1994. Object Oriented Software Metrics. Prentice Hall.
Marinescu, R., 2002. Measurement and Quality in Object-Oriented Design. Department of Computer Science, Politehnica University of Timisoara (Ph.D. thesis).
Martin, R.C., 2002. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall.
McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. 2 (4), 308–320. doi:10.1109/TSE.1976.233837.
Mordal-Manet, K., Anquetil, N., Laval, J., Serebrenik, A., Vasilescu, B., Ducasse, S., 2013. Software quality metrics aggregation in industry. J. Softw.: Evol. Process 25 (10), 1117–1135. doi:10.1002/smr.1558.
Mouchawrab, S., Briand, L.C., Labiche, Y., 2005. A measurement framework for object-oriented software testability. Inf. Softw. Technol. 47 (15), 979–997. doi:10.1016/j.infsof.2005.09.003.
Offutt, J., Abdurazik, A., Schach, S.R., 2008. Quantitatively measuring object-oriented couplings. Softw. Quality J. 16 (4), 489–512. doi:10.1007/s11219-008-9051-x.
Rademakers, T., Dirksen, J., 2009. Open-Source ESBs in Action. Manning.
Risi, M., Scanniello, G., Tortora, G., 2013. Metric attitude. In: Proceedings of Conference on Software Maintenance and Reengineering. CSMR, pp. 405–408. doi:10.1109/CSMR.2013.59.
RosettaNet, 2011. RosettaNet home. URL http://www.rosettanet.org
Schneidewind, N.F., 1987. The state of software maintenance. IEEE Trans. Softw. Eng. 13 (3), 303–310. doi:10.1109/TSE.1987.233161.
Sheldon, F.T., Jerath, K., Chung, H., 2002. Metrics for maintainability of class inheritance hierarchies. J. Softw. Maint. 14 (3), 147–160. doi:10.1002/smr.249.
Sheskin, D.J., 2012. Handbook of Parametric and Nonparametric Statistical Procedures, fifth ed. Chapman and Hall/CRC.
Swift, 2011. Society for Worldwide Interbank Financial Telecommunication Home. http://www.swift.com
Tempero, E.D., Noble, J., Melton, H., 2008. How do java programs use inheritance? An empirical study of inheritance in java software. In: Proceedings of European Conference on Object-Oriented Programming. ECOOP, pp. 667–691. doi:10.1007/978-3-540-70592-5_28.
Yu, L., 2008. Common coupling as a measure of reuse effort in kernel-based software with case studies on the creation of MkLinux and Darwin. J. Braz. Comput. Soc. 14, 45–55. doi:10.1007/BF03192551.

Rafael Z. Frantz is an Associate Professor who is with the Department of Exact Sciences and Engineering of the Unijuí University, Brazil, and leads the Applied Computing Research Group since 2013. He was awarded a Ph.D. degree in Software Engineering by the University of Seville, Spain. His current research interests focus on the integration of enterprise applications and search-based software engineering.

Rafael Corchuelo is a Reader in Computer Science who is with the Department of Computer Languages and Systems of the University of Sevilla, Spain. He received his Ph.D. degree from this University, and he leads its Research Group on Distributed Systems since 1997; his current research interests focus on the integration of web data islands; previously, he worked on multiparty interaction and fairness issues.

Fabricia Roos-Frantz is an Associate Professor who is with the Department of Exact Sciences and Engineering of the UNIJUÍ University, Brazil. She received her Ph.D. in Software Engineering from the University of Seville, Spain. Her current research interests include software product lines and search-based software engineering.