A Software Development Kit to Implement Integration Solutions

Rafael Z. Frantz
UNIJUÍ University, Department of Technology
Rua do Comércio, 3000, Ijuí, 98700-000, RS, Brazil
rzfrantz@unijui.edu.br

Rafael Corchuelo
Universidad de Sevilla, ETSI Informática
Avda. de la Reina Mercedes, s/n, Sevilla 41012, Spain
corchu@us.es

ABSTRACT

Typical companies rely on their software ecosystems to support and optimise their business processes. There are a few proposals to help software engineers devise enterprise application integration solutions. Some companies need to adapt these proposals to particular contexts. Unfortunately, our analysis reveals that they are not so easy to maintain as expected. This motivated us to work on a new proposal that has been carefully designed in order to reduce maintainability efforts.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Software libraries; D.2.11 [**Software Architectures**]: Domain-specific architectures; D.2.13 [**Reusable Software**]: Domain engineering

Keywords

Enterprise Application Integration; Business Modelling, Integration Framework; Domain-Specific Languages.

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 (EAI) provides methodologies and tools to design and implement EAI solutions. The goal of an EAI 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 EAI solution [9].

In the last years, several tools have emerged to support the design and implementation of EAI solutions. Hohpe and Woolf [9] documented many patterns found in the development of EAI solutions. Camel and Spring Integration are two well-known proposals to support these patterns.

We are concerned with maintainability. According to IEEE [10], 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 paper, we are interested adaptive maintenance, which is very important for companies that need to build their own tools building on existing tools. Many companies rely on open-source tools that can be adapted to a particular context within their business domain. For example, a company that develops EAI solutions may need tools that can be adapted to particular contexts, e.g., e-commerce (RosettaNet [16]), health (HL7 [8]), or insurance (HIPPA [7]).

It is not new that how a software system was designed and implemented, has an impact on its maintenance costs [1, 4, 11, 17]. In both design and implementation, software engineers need to pay attention to readability, understandability, and complexity. 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 are not involved in maintenance tasks. 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 execution flows and debug them. Thus, to reduce the costs involved in the adaptation of a software system to a particular context, it is very important that the software system was designed taking into account properties that have a negative impact on maintenance.

How costly it is to maintain a tool depends on a variety of properties. We have calculated these measures on Camel and Spring Integration, and the results do not seem promising enough. This motivated us to work on a Software Development Kit (SDK) to which we refer to as Guaraná SDK [1] that provides a well-designed architecture that is intended

[1] http://www.guarana-project.net

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

to reduce maintainability costs. Guaraná SDK aims to support the implementation of EAI solutions. Its design provides better values for the maintainability measures, which suggests that Guaraná SDK is more maintainable and thus easier to adapt for a particular context than Camel or Spring Integration.

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. The latter extends the framework to provide an implementation of tasks and adapters that is intended to be general purpose; other toolkits for specific contexts are currently under development.

The rest of the paper is organised as follows: Section §2 presents the related work and compares other proposals to ours; Section §3 presents the framework layer of Guaraná SDK; Section §4, presents the toolkit layer, which extends the framework layer for a particular context; finally, Section §5 reports on our main conclusions.

2. RELATED WORK

The Software Engineering community uses a number of measures proposed by Chidamber and Kemerer [2], Henderson-Sellers [6], Martin [13] and McCabe [14] to have an overall idea of how difficult maintaining a system can be.

Camel and Spring integration are the most closely-related proposals. They both are based on the catalogue of integration patters by Hohpe and Woolf [9]. Camel provides a graphical editor and an API that can be used in Java, Scala, and Spring configuration files. Spring integration also provides a graphical editor and an API that can be used in Java or Spring configuration files.

Since we are interested in how maintainable they are, we have studied the following maintainability metrics:

NOP: Number of packages that contain at least one class or interface. This measure can be used as indicator of how much effort it is required to understand how packages are organised; note that this provides the overall picture of the design of a system [3]. The greater this number, the more effort shall be required.

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 is this value, the more difficult it is to understand a software system.

NOI: Number of interfaces.

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 Chidamber and Kemerer [2], Lorenz and Kidd [12], a large number of methods may indicate a class is likely to be application specific, limiting the possibility of reuse.

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.

MLC: Number of lines in methods, excluding blank lines and comments. According to Henderson-Sellers [6], this value should not exceed fifty. If it does, the author suggests to split this method into other methods to improve readability and maintainability. The greater this number, the more difficult it is to understand and maintain the method.

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 [6], the number of parameters should not exceed five. If it does, the author suggest that a new type must be designed to wrap the parameters into a unique object. The greater this number, the more difficult it is to understand the method.

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 calculated with the Henderson-Sellers LCOM* method [6]. A low value indicates a cohesive class; contrarily, a value close to one indicates lack of cohesion and suggests the class might better be split into two or more subclasses because there can be methods that should probably not belong to that 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 [14], this value should not exceed ten. The greater this value, the more difficult it is to maintain a piece of code.

We have calculated these measures regarding the core implementation of Camel, Spring Integration, and Guaraná SDK i.e., we do not take into account the code required to implement the adapters necessary to interact with the applications being integrated. We do not considered this code because it is peripheral to the tools' main architecture, and in most cases comes from other open-source projects that are maintained separately. Table §1 summarises the results.

The architecture of Camel and Spring Integration is organised in several packages, 54 and 32, respectively. There are cases in which the maximum number of classes in a package reaches 96 in Camel and 58 in Spring Integration. These values show that Camel has almost double as many classes in a package as Spring Integration. 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. The highest standard deviation calculated for Camel, indicates that there are packages containing many more classes than the average calculated per package, which may have an impact on the understandability of the package. Spring Integration has a low value for the standard deviation regarding the number of interfaces. The architecture of Guaraná is organised into 18 packages, and the maximum number of classes in a package is no more than 11. Furthermore, Guaraná provides no more than 9 interfaces distributed in these packages. The standard deviation calculated for the number of classes and interfaces per package is very low, 3.09 and 0.76, respectively.

When analysing the methods in classes and interfaces, we found that Camel has $7,015$ methods compared to the $1,431$ found for Spring Integration. Most probably the difference amongst Spring Integration and Camel is because it has less than a half the number of classes and interfaces of

	Camel				Spring Integration				Guaraná SDK			
Measure	**Total**	**Mean**	**Dev.**	**Max**	**Total**	**Mean**	**Dev.**	**Max**	**Total**	**Mean**	**Dev.**	**Max**
NOP	54	-	-	-	32	-	-	-	18	-	-	-
NOC	730	13.52	19.55	96	269	8.41	10.52	58	79	4.39	3.09	11
NOI	140	2.59	9.07	58	40	1.25	1.84	9	9	0.50	0.76	2
NOM	7,015	9.61	15.36	192	1,431	5.32	5.60	39	369	4.67	4.61	24
LOC	62,439	-	-	-	14,929	-	-	-	2,878	-	-	-
MLC	34,839	4.52	8.15	141	8,264	5.65	9.59	110	1,748	4.72	6.43	54
NPM	-	0.93	1.05	11	-	1.13	0.94	9	-	1.20	1.04	4
LCM	-	0.29	0.35	1	-	0.22	0.33	0.94	-	0.14	0.27	0.91
MCC	-	1.67	2.06	46	-	1.80	2.04	30	-	1.35	0.91	8

NOP = Number of packages; NOC = Number of classes; NOI = Number of interfaces; NOM = Number of methods; LOC = Number of lines of code; MLC = Number of lines in methods; NPM = Number of parameters per method; LCM = Lack of cohesion of methods; MCC = McCabe's Cyclomatic Complexity.

Table 1: Comparison of maintainability measures.

Camel. The values that stand out are the maximum number of methods per class/interface calculated in Camel, which is 192, against to 39 in Spring Integration. We have analysed the methods in classes/interfaces and found that Guaraná has in total 369 methods, with a maximum number of 24 methods per class/interface.

Other values that are impressive for these tools are regarding the total number of lines of code, which is very high, chiefly for Camel. There are 62,439 lines of code in Camel and 14,929 in Spring Integration. The implementation of Guaraná has a total number of 2,878 lines of code.

Counting the number of lines of code inside methods, we found Camel has a total number of 34,839, Spring Integration has 8,264, which if compared to the total number of lines of code, represents 0.55% and 0.55% of these values, respectively. It means there are many attributes declared in classes. The maximum value calculated demonstrate that there are some methods with up to 141 lines of code in Camel and 110 in Spring Integration. These values indicate there may be necessary more effort to maintain and understand the methods in these tools. Counting the number of lines of code inside methods, we found Guaraná has a total number of 1,748, which, if compared to the total number of lines of code in Guaraná, 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 classes shall be easier to understand and maintain.

If we look at the maximum number of parameter per method, it is also impressive how large it is, chiefly in Camel. This tool has up to 11 and Spring Integration has up to 9 parameters per method. These values indicate some classes in these tools 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. Guaraná has no more than 4 parameters per method. These values indicate that classes in Guaraná are expected to be more reusable and its methods not so difficult to understand.

The lack of cohesion of methods is similar in every analysed tool. Camel has 0.29 and 0.35 and Spring Integration has 0.22 and 0.33. Guaraná presents a mean of only 0.14.

The values calculated for the McCabe cyclomatic complexity have indicate that there are cases in which they are extremely high. This is indicated by the maximum values

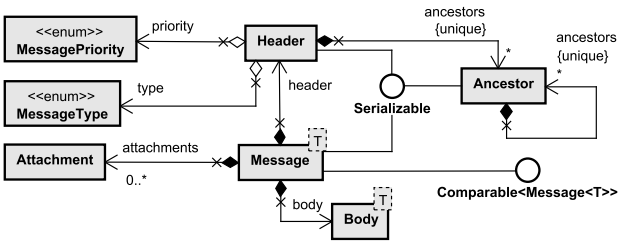

Figure 1: Message model in the framework.

calculated for the tools, which reaches 46 and 30 in Camel and Spring Integration, respectively. Consequently, they are also very complex tools, which may have a serious impact on their maintenance. The values calculated for the McCabe cyclomatic complexity have indicated that the maximum value is 8, which indicates the architecture is well designed and maintenance is expected to be easy.

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 paper. It is not difficult to see that the values of these properties are generally significantly smaller in Guaraná SDK, which suggests that maintaining our proposal is easier than maintaining the other technologies.

3. THE FRAMEWORK LAYER

In the following sections, we describe the packages in the framework layer.

3.1 Messages

Messages are used to wrap the data that is manipulated in an EAI solution. They are composed of a header, a body and one or more attachments, cf. Figure §1. The header holds several meta-data properties, including: message identifier, expiration time, and list of parents. The message identifier is a UUID that uniquely identifies every message; the expiration time allows to set a deadline after which a message is considered outdated for further processing; the list of parents allows to track which messages originate from which ones, i.e., it helps find correlated messages. The body holds

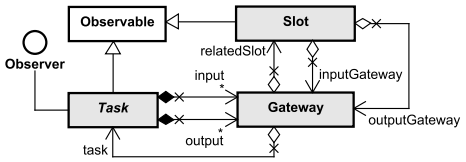

Figure 2: Task model in the framework.

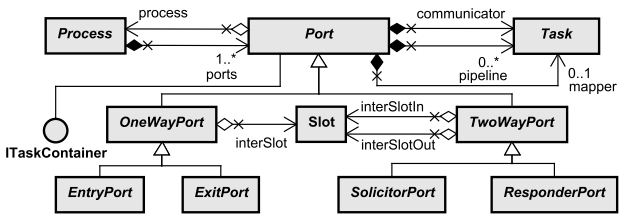

Figure 3: Port model in the framework.

the payload data. Attachments allow messages to carry extra pieces of data associated with the payload, e.g., an image or an e-mail message. 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, and comparison enables a workflow to process messages according to their priority.

3.2 Tasks

The task package provides the foundations to implement domain-specific tasks in specialised toolkits. Figure §2 shows the task model in our proposal. A task models how a set of inbound messages must be processed to produce a set of outbound messages. Tasks communicate indirectly by means of slots. A slot is an in-memory priority buffer that helps transfer messages asynchronously so that no task has to wait until the next one is ready to start working. 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 later case, it becomes ready when the appropriate set of messages is available in its input slots. 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 slots and uses the later to enrich the former; this task cannot become ready to be executed until two messages are available in its input slots. Slots and tasks are both observable objects, which means that they can notify other objects of changes to their state; in addition, tasks are observer objects since they monitor slots.

3.3 Ports

Ports abstract away from the communication mechanism used to interact with the environment, cf. Figure §3. Note that every port must be associated with a process, and that we distinguish between one way and two way ports. The former are unidirectional ports that allow to read/write mes-

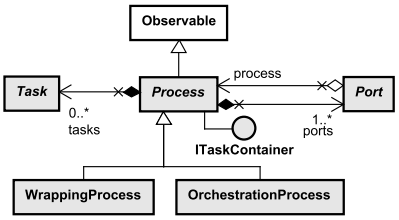

Figure 4: Process model in the framework.

sages; the latter are bidirectional ports that allow to either send a message and wait for the answer (responder ports) or to receive a message and produce a response message (solicitor port). Internally, ports are composed of tasks, namely: a communicator, a pipeline, and a mapper. Communicators are the tasks that allow to actually read or write a message, namely: in communicators are used to read a message in raw from; contrarily, out communicators are used to write a message in raw form. By raw form, we mean a stream of bytes that is understood by the corresponding process or asset. The pipeline helps pre- or post-process a message in raw form to decrypt/encrypt, decode/encode, or unzip/zip it. The pipeline in an input port ends with a mapper task that transforms the resulting stream of bytes into a text document, for instance; the pipeline in an output port begins with a mapper that transforms the text document into a stream of bytes. Bidirectional ports can actually be seen as a combination of an input and an output port.

3.4 Processes

Processes are the central processing units in an integration solution, cf. Figure §4. They are composed of ports and tasks, extend class Observable, and implement interface ITaskContainer. 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 our proposal, they are just a container that reports which of their tasks are ready for execution to an external observer. The ITaskContainer defines the interface to add, remove, and search for tasks in objects that may contain tasks. 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 the 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 interact 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.

3.5 Adapters

This package provides the foundations to implement adapters in specialised toolkits, cf. Figure §5. Adapters are the piece of software that implements the low-level communication

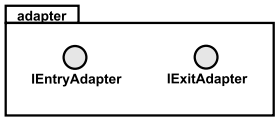

Figure 5: Adapter model

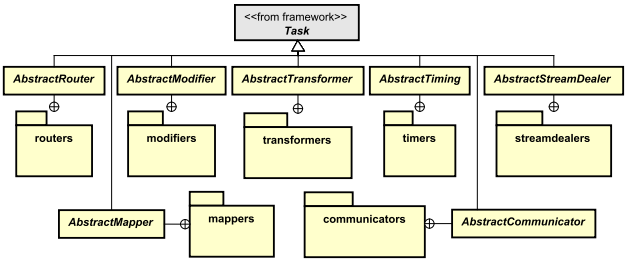

Figure 7: Task model in the toolkit

protocol necessary to interact with the processes or assets involved in an integration solution. The framework layer provides four interfaces to describe the operations used by ports to read, write, solicit, and respond.

3.6 The execution engine

The engine package provides an implementation to the task-based runtime system on which Guaraná SDK relies, cf. Figure §6. Scheduler is the central class in this package. At run-time, a scheduler owns a work queue, a list of workers, and three monitors. The work queue is a priority queue that contains work units to be processed. A work unit consists of a task to be executed and a deadline. In most cases, the deadline is set to the current time, which means that the corresponding task can execute as soon as possible; there are a few time-dependent tasks for which the deadline is set to a time in future, e.g., a timer that ticks every minute or a communicator that polls a service every ten minutes. Workers are an extension to the Thread class and they have a reference to the work unit that they have to execute. The monitors are intended to gather statistics about how memory is used, the time tasks take to complete, and the size of the work queue. They have been implemented as independent threads that run at regular intervals, gather the previous information, dump it to a file, and become idle the sooner as possible.

4. THE 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 [9], and provide active and passive adapters that enable the use of several low-level communication protocols.

Figure §7 presents the extensions to the Task class. Below we provide an explanation in which term schema is used 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.

- 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.
- Timer: a timer is a task that performs a time-related action, e.g., delaying a message or producing a message at regular intervals.
- 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.

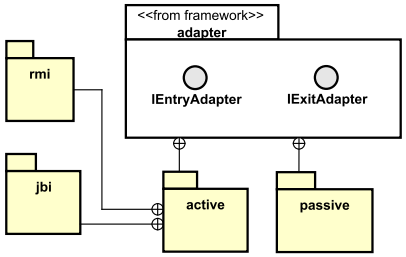

Figure 8: Adapter model in the toolkit

There is a package associated with every of the previous tasks, which provide a variety of specific-purpose implementations in each integration patten category [5].

Figure §8 shows the extension to the adapters. They can be either active or passive. An active adapter allows to poll periodically the process or application with which it interacts; contrarily, a passive adapter is intended 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 JBI and RMI protocols, respectively. Note that supporting JBI adapters allows to plug Guaraná SDK into a variety of ESBs; for instance, our reference implementation is ready to be plugged into Open ESB [15]. This, in turn, allows Guaraná SDK processes to have access to a variety of applications in current software ecosystems, including files, databases, web services,

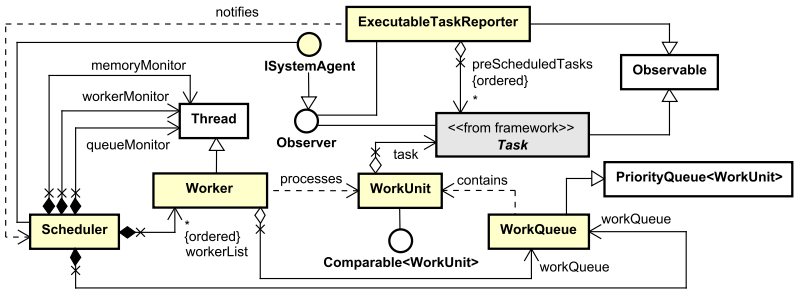

Figure 6: Engine model in the toolkit.

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.

5. CONCLUSIONS

Companies rely on applications to support their business activities. The integration of these applications is necessary, chiefly because it allows to reuse applications to support new business processes or to optimise existing ones. Enterprise Application Integration (EAI) provides methodologies and tools to design and implement EAI solutions. Although it is possible to use current tools to design and implement EAI solutions, it is still necessary to provide domain-specific tools that are easy to maintain in order to customise them for a particular context.

In this paper we have reported on nine maintainability measures that can be used as an indicator regarding how expensive a software tool can be to maintain. We have analysed two widely-used tools in the EAI marked: Camel and Spring Integration. Our results indicate that these tools are expected to be more difficult to maintain than our proposal.

We have introduced Guaraná SDK, which is our software tool to design and implement EAI solutions. Guaraná SDK is organised in two layers, namely: framework and toolkit. An abstract implementation of the core concepts in the domain-specific language introduced in [5] are provided by the former layer, and, although we also provide a toolkit implementation, the framework can be reused by engineers to build other new specialised toolkits. We have also calculated the values for the maintainability measures regarding Guaraná SDK. Our conclusion is that Guaraná SDK is much easier to maintain than Camel or Spring Integration.

6. ACKNOWLEDGMENTS

The work in this paper was supported by the Evangelischer Entwicklungsdienst e.V. (EED), 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), and the European Commission (FEDER).

References

[1] S. Bergin and J. Keating. A case study on the adaptive maintenance of an Internet application. *Journal of Software Maintenance*, 15(4):254–264, 2003

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6): 476–493, 1994

[3] X. Dong and M. W. Godfrey. Understanding source package organization using the hybrid model. In *International Conference on Software Maintenance*, 2009

[4] A. Epping and C. M. Lott. Does software design complexity affect maintenance effort? In *Goddard Space Flight Center*, pages 297–313, 1994

[5] R. Z. Frantz, A. M. Reina-Quintero, and R. Corchuelo. A domain-specific language to design enterprise application integration solutions. *International Journal of Cooperative Information Systems*, 20(2):143–176, 2011

[6] B. Henderson-Sellers. *Object-Oriented Metrics, Measures of Complexity*. Prentice Hall, 1996

[7] Health Insurance Portability and Accountability Act Home, Oct 2011. Available at http://www.hipaa.com/

[8] Health Level Seven International Home, Oct 2011. Available at http://www.hl7.org

[9] G. Hohpe and B. Woolf. *Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003

[10] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Computer Society, 1990

[11] M. Jørgensen. An empirical study of software maintenance tasks. *Journal of Software Maintenance*, 7(1):27–48, 1995

[12] M. Lorenz and J. Kidd. *Object Oriented Software Metrics*. Prentice Hall, 1994

[13] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002

[14] T. J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976

[15] T. Rademakers and J. Dirksen. *Open-Source ESBs in Action*. Manning, 2009

[16] RosettaNet Home, Oct 2011. Available at http://www.rosettanet.org

[17] N. F. Schneidewind. The state of software maintenance. *IEEE Trans. Software Eng.*, 13(3):303–310, 1987