Empirical Software Engineering manuscript No.
(will be inserted by the editor)

A Qualitative Human-Centric Evaluation of Flexibility in Middleware Implementations

Renato Maia · Renato Cerqueira · Clarisse Sieckenius de Souza · Tomás Guisasola-Gorham

Received: date / Accepted: date

Abstract Today middleware is much more powerful, more reliable and faster than it used to be. Nevertheless, for the application developer, the complexity of using middleware platforms has increased accordingly. The volume and variety of application contexts that current middleware technologies have to support require that developers be able to anticipate the widest possible range of execution environments, desired and undesired effects of different programming strategies, handling procedures for runtime errors, and so on. This paper shows how a generic framework designed to evaluate the *usability* of notations (the Cognitive Dimensions of Notations Framework, or CDN) has been instantiated and used to analyze the cognitive challenges involved in adapting middleware platforms. This human-centric perspective allowed us to achieve novel results compared to existing middleware evaluation research, typically centered around system performance metrics. The focus of our study is on the process of adapting middleware implementations, rather than in the end product of this activity. Our main contributions are twofold. First, we describe a qualitative CDN-based method to analyze the cognitive effort made by programmers while adapting middleware implementations. And second, we show how two platforms designed for flexibility have been compared, suggesting that certain programming language design features might be particularly helpful for developers.

Keywords Middleware evaluation · API evaluation · Programmer experience · Qualitative methods · Cognitive Dimensions of Notations.

1 Introduction

In the last two decades middleware platforms have been continually improved by researchers and developers working with distributed systems. Today middleware is much more powerful, more reliable and faster than it used to be. Nevertheless, for application programmers,

Renato Maia · Renato Cerqueira · Clarisse Sieckenius de Souza · Tomás Guisasola-Gorham
Department of Informatics
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
22453-900 – Rio de Janeiro – RJ – Brazil
E-mail: {maia,rcerq,clarisse,tgorham}@inf.puc-rio.br

the complexity of dealing with middleware platforms has increased accordingly. The volume and variety of application contexts that must be supported require that programmers be able to anticipate the widest possible range of execution environments, desired and undesired effects of different programming strategies, handling procedures for runtime errors, and so on. Flexible middleware implementations, which usually means platforms that are easy to adapt or customize for different purposes and contexts, have become a fundamentally required feature in contemporary distributed systems. However, the available tools for *building* such systems have apparently failed to keep in pace.

During the same two decades of distributed systems evolution, the Human-Computer Interaction (HCI) community has completely transformed the way how end users interact with computer technologies. This tremendous progress has of course affected programmers, who now have sophisticated programming environments at their disposal, with graphical user interfaces, visualization tools, tracing facilities, and many other resources and features that were not in place before. However, the *programming concepts* supported by the programming languages that they use haven't evolved as much. Should they have? Could they have?

As pointed out by Vinoski (2003b), the traditional approach to evaluate middleware focuses on measuring the system performance and scalability with a specific workload, typically neglecting aspects such as flexibility or ease of programming. As a result, the answers to the questions above are not fully known yet. This can be partially explained by the interdisciplinary character of the research needed to find the answers. The distributed systems and the HCI community must come together to achieve the task, a point raised previously by Edwards et al (2003), Arnold (2005) and Henning (2009), for example. Yet, another aspect of the explanation may be the methodology that must be used in this sort of investigation. Although HCI researchers are by definition familiar with experimental and analytical methods applied to understand or explain *human* behavior, the same is not true of researchers working with distributed systems. In particular, HCI has now extensively embraced the use of *qualitative* methods borrowed mainly from the Social Sciences (Cairns and Cox, 2008; Lazar et al, 2010), whereas research publications using qualitative methods in Software Engineering are still scarce (Dittrich et al, 2007). Qualitative methods are especially important for our discussion because they explore *meanings*, how they evolve and how they affect human activity (Denzin and Lincoln, 2000).

Our work was motivated by informal observations of the learning process of graduate and undergraduate students in distributed systems disciplines. We have been using two different platforms as the base for pedagogical projects where students must implement additional features to each platform in order to learn specific concepts in the domain of middleware design and implementation. We were intrigued by the fact that, as a rule, students did better in one platform than in the other, although both were *designed for flexibility*. They were, however, implemented in different programming languages, which students should know in order to do the project. An evident approach to finding out the causes of this difference in student performance is to investigate, for instance, the existence of a correlation between a student's familiarity with one or another programming language and her performance with the programming tasks. This approach relies on quantitative methods of analysis to support or refute a correlation hypothesis in this context. Another approach is to follow a qualitative method and to make an in-depth analysis of one instance of the phenomenon of interest, and produce a rich grounded interpretation of what is the case. We chose the latter alternative, and carried out an *exploratory* research study (Robson, 2002).

In this research we applied the Cognitive Dimensions of Notations framework (Green, 1989) to analyze and compare the two implementations of middleware platforms we use in

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

our teaching. Since CDN, as its name says, requires an instance of *notation* for the analysis, our first step was to define what constitutes the notation of an implemented middleware platform. Thus we defined our *object of analysis*. Next, we selected typical adaptation tasks that can be used to decide whether a platform is more or less flexible than the other. The use of tasks in CDN-based analyses is canonical (Green and Petre, 1996). The third step consisted of instantiating each one of the 14 cognitive dimensions (Green, 1989; Green and Petre, 1996) in the context of middleware adaptation. The fourth step was a comparison between both notations, one for each of the platforms under examination. The final step was a comprehensive interpretation of findings from previous stages. The value of this approach was to find certain properties of programming languages and middleware architectures that clearly distinguish one implementation from the other. This, we think, is a factor that must be considered independently of whether there is or is not a correlation between students' performance while adapting middleware platforms and their familiarity with the programming language in which the adaptation must be expressed. Our findings in this study point in the direction of programming concepts that haven't yet been implemented in mainstream programming languages. Similar concepts have already been proposed by work in component-oriented programming (Rouvoy and Merle, 2009), architectural descriptions (Aldrich et al, 2002) and gradual typing (Siek and Taha, 2007).

The main contributions of this paper are twofold and can be summarized as: a description of a qualitative CDN-based method to analyze the cognitive effort required to adapt middleware implementations; and a comparison of two platforms designed for flexibility, suggesting that certain programming language design features might be particularly helpful for developers in order to improve flexibility.

In the remainder of this paper we present related work (Section 2), the methodological perspective we adopted (Section 3), how we instantiated CDN to evaluate the flexibility of middleware implementations (Section 4), our experimental study using CDN (section 5), and finally our conclusions, contributions and future work (section 6).

2 Related Work

Arnold (2005) and Henning (2009) warn us about common usability-related misconceptions in API design that could be avoided. Both authors underline the value of good API design. Good APIs are easier to learn and easier to use, and because of this, may be effectively and efficiently used to produce software which, in turn, should have fewer bugs. Thus, good APIs will be more productive. Their arguments are mainly based on practical experience and personal reflections. Therefore, they serve as inspiration but yet not as a basis for scientific research, since the authors do not use a systematic method to analyze or produce the evidence that grounds their opinions.

In this section, we briefly discuss previous research work doing a systematic evaluation of middleware platforms or programming issues in general, but using two very distinctive approaches.

2.1 Quantitative Metrics for Evaluation of Middleware Usability

Scientific research in usability of middleware platforms is rare. However, there are studies that apply traditional software engineering metrics or other quantitative methods to evaluate aspects of middleware that impact its usability, like modularity, implementation complexity,

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

etc. One example of traditional software engineering metrics applied to middleware evaluation is the study by Cacho et al (2006). The authors analyze Java and AspectJ compliant implementations of OpenORB — a reflective middleware architecture — using three metrics: Concern Diffusion over Components, Concern Diffusion over Operations and Concern Diffusion over Lines of Code. They claim that the modularity of reflective middleware can be improved by aspectizing the reflection-specific crosscutting concerns.

Another example of using software engineering metrics to evaluate middleware is presented by Costa et al (2007). The authors propose a middleware platform for wireless sensor networks, called TeenyLime, based on a tuple space mechanism. They argue that TeenyLime yields simpler, cleaner, and more reusable implementations, when compared to mainstream solutions for wireless sensor networks. This comparison is based on quantitative source level metrics (explicit application states, lines of code, and the percentage of the application information moved from the application component into the tuple space) applied to a reference application. Similarly, Ramdhany et al (2009) use the degree of code reuse as the basis for their evaluation of the extent to which a component-based middleware architecture for mobile ad-hoc networks minimizes the time needed to develop and port new ad-hoc routing protocols. They assume that the number of reused components and lines of code has a direct impact on the development time and effort.

A rather different approach is inspired by computational complexity of algorithms. Eden and Mens (2006) follow such analytical approach, trying to measure software flexibility by counting the number of modifications in the code. They also conclude from an experimental test involving seven participants that their predictions with the formulae are corroborated.

Another study following the quantitative approach and explicitly focused on distributed systems is reported by Ranganathan and Campbell (2007). The authors set out to describe which factors determine the complexity of a distributed computing system from the people's point of view. In order to do that, they propose to measure different aspects of a system. Additionally, they propose a set of guidelines derived from measures that can help reduce the system's complexity. Although the authors cover more aspects than traditionally analyzed with software engineering metrics, their proposal is still grounded in computational characteristics of the systems, like size, number of steps to doing tasks and probabilistic interactions between other software through the network.

While the quantitive approach used in these studies can be very suitable for quantifying specific aspects of software, trying to infer the usability of a programming tool or abstraction based solely on the final product (the software), without taking into account the programming process that led to it, is not the only strategy, nor necessarily the best one.

2.2 Qualitative Methods to Evaluate Usability of Programming Tools and Abstractions

Regarding API usability in general, there is relatively more research work coming from the HCI community. Unlike the traditional approach used by the Software Engineering community, which is based on source-level metrics, approaches using HCI techniques focus on the programming activity (the usage) instead of the final product (the implementation).

More than a decade ago, McLellan et al (1998) made experiments with programmers using an API. Their methodology was based on user observation. Users were shown a contrived example with the use of the API and asked to point out what kind of knowledge was required to produce it. At the end of the activity they were requested to answer a questionnaire where they could talk about their impressions and rationale. The authors produced a

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

detailed report with recommendations for changes that was passed on to the team responsible to the APIs. Another contribution of their work was a good description of the procedures adopted to test the APIs and examples that could be used as a basis for users' tasks in observation sessions.

More recently, Clarke (2001) conducted a series of studies on API usability. Like McLelland and co-authors, Clarke also observed programmers using an API and the development environment, but he used tasks that might be done within a certain period of time. At the end of programming sessions, the subjects answered a questionnaire proposed by the Cognitive Dimensions of Notations framework (CDN) (Green, 1989; Blackwell and Green, 2003), a technique developed originally to evaluate programming languages and interactive languages, but adapted to evaluate application programming interfaces by Clarke and Becker (2003). An interesting approach of Clarke's study was the classification of programmers according to their behavior (he called these groups *personas*) in order to account for different programmers' demands (Clarke, 2004). Clarke concluded with a set of recommendations to the development team at his company.

Ko et al (2006) tried to understand programmers' habits and strategies, but this study focuses more on the immediate improvement in the interface of development environments. The authors propose a model for patterns of activity in programming, but they assume that the programmer already knows what to do — that she has a plan — and is just trying to execute this plan.

Work based on qualitative techniques typically focuses on the identification of relevant (meaningful) aspects of software usability in order to recommend how to improve it, rather than provide measurements of usability for comparison and quality assurance purposes. Both approaches are valid and complementary, although their distinctions are not always clear. In the next section, we discuss these two approaches and their suitability for the goals of our research.

3 The Role of Qualitative and Quantitative Approaches

Qualitative and quantitative methods serve different purposes in research, and disputes about which one is more scientific than the other have prevented the advancement of disciplines more than contributed to the validity of research results. Very briefly, qualitative methods aim at *finding* meanings and variety in specific instances of human activity, whereas quantitative methods aim at *measuring* the impact and extent of meanings and variety. The procedures involved in qualitative and quantitative methods are radically different, and therefore require from researchers radically different skills. This may contribute to explain some of the unproductive dispute: there aren't many researchers with the time or inclination to become skillful in both types of methods.

Qualitative methods require the ability to perform systematic interpretation, constant self-surveillance against biases, consistent identification and separation of different categories of meanings, identification of direct and indirect expression of invariant meanings in the variety of discourse, articulation of meanings into novel interpretive schemas, systematic exploration of opportunities for antithetical evidence, and so on. Quantitative methods require the ability to formulate testable hypotheses, identify measurable variables that express the essence of hypotheses, control measurements and isolate cross-dependencies, understand how measurements translate aspects of the hypotheses, analyze the validity of measurements, interpret the impact of measures and tested hypotheses in the overall design of a research project, and so on.

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

A separate but related discussion, often confused with the use of qualitative and quantitative methodologies, is about the very purpose of scientific knowledge. Some believe that the sole purpose of science is to *predict* events and properties of elements in reality, whereas others believe that there are additional purposes for science given that some aspects of reality seem to be inherently difficult to predict consistently. Human creativity, for instance, a *sine qua non* condition for discovering and proposing predictive models of reality, is the paradigmatic example of unpredictability in nature.

Quantitative methods are the only ones equipped to analyze large amounts of data required for predicting aspects of reality. Hence, researchers whose purpose is to generate predictive knowledge use quantitative methodology. However, even in the stream of predictive knowledge generation there are prime opportunities for contributions coming from qualitative research. The combination of approaches is often referred to as *mixed methods approach* (Creswell, 2009). One such opportunity, which we take in this paper, is to use qualitative research to shed light on hidden meanings and issues involved in the phenomena under investigation so as to feed potential hypotheses that can be tested with quantitative methods later on.

Qualitative research is carried out in very specific contexts, focusing on a small number of evidence sources, privileging in-depth analysis rather than wide-spread coverage, and requires intensive systematic interpretation of meanings present in the evidence. Systematic interpretation can be built from the ground (Glaser and Strauss, 1967) or be informed by theories, models and frameworks. In the latter case, a considerable part of the research effort is devoted to mapping conceptual structures provided by theories, models and frameworks onto interpretive procedures that can be systematically applied to the analysis of the specific situation under investigation. Thus, the contribution of this particular form of research is actually twofold. On the one hand, by making their interpretive procedures explicit, researchers allow other colleagues to analyze methodological decisions and eventually use them to investigate other questions. On the other, of course, the results and conclusions achieved in qualitative research informed by theories, models and frameworks represent new knowledge that can be further used, inspected and discussed to advance the state of the art in a particular discipline.

Our own research follows this path. We have used the CDN framework (Green, 1989) to make an in-depth analysis of two middleware implementations in order to understand the characteristics that differentiate them in the context of software flexibility. To accomplish that, we had to reinterpret the CDN framework according to this context, which is rather different than what it was originally proposed for. The following sections describe both the CDN framework and our interpretations of its dimensions, and also show the results we obtained with its application.

4 CDN Instantiation for Evaluating Middleware Flexibility

For some time, we have been investigating the use of the Lua language (Ierusalimschy, 2006) in the development of middleware and other programming tools for distributed applications (Rodriguez et al, 1996; Ierusalimschy et al, 1998; Cerqueira et al, 1999; Moura et al, 2002; Maia et al, 2004, 2005, 2006). Our main interest has been on the use of Lua's high-level constructs and dynamic features to facilitate the development of distributed applications and middleware systems.

Similar to other programming languages like TCL, JavaScript, Python, and Ruby, Lua is typically classified as a *dynamic language*. The term dynamic language is not precisely de-

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

fined in the literature, thus the classification of programming languages as dynamic or static might vary. However, we consider that four characteristics are fundamental to classifying a language as a *dynamic language*: the ability to incorporate new portions of sourde code at runtime (*interpretation*), type verification at runtime (*dynamic typing*), support for metaprogramming at runtime (*computational reflection*), and automatic memory management (*garbage collection*).

The most representative example of our work on the use of dynamic languages in distributed systems, specially in middleware design and implementation, is the development of OiL (Maia et al, 2006), a middleware platform with support for distributed objects as the main distributed programming abstraction. Based on the object model and communication protocols specified in the CORBA standard (OMG, 2008), OiL was written in Lua and specifically designed to take advantage of Lua's facilities in order to provide an implementation that can be easily modified and adapted to different uses.

We have been using OiL together with other open-source middleware implementations, such as JacORB[1] and MICO[2], as the base for pedagogical projects where graduate and undergraduate students are asked to implement additional features to these middleware platforms in order to learn specific concepts in the domain of middleware design and implementation. Our general observation in different learning situations is that students seem to make changes and adaptations more easily in OiL than in similar implementations written in other languages more commonly used for middleware development, like Java or C++. However, it has never been quite clear *what* made OiL's implementation easier to be modified than others, and how or if certain characteristics of Lua effectively contributed to this.

Motivated by this observation, we decided to conduct a study with the goal of identifying the factors that justify the presumably greater flexibility of OiL. A broader goal of this study is to bring more light into the understanding of the major factors in the improvement of middleware flexibility.

The study was carried out using the Cognitive Dimensions of Notations framework, which is "a set of discussion tools for use by designers and people evaluating designs" (Blackwell and Green, 2003, pp. 106). This definition highlights the qualitative nature of CDN. The distinctive product of qualitative research is a reasoned interpretation of a situated phenomenon. Thus, a set of "discussion tools" is precisely what we need to build the targeted interpretation. The CDN framework provides a shared vocabulary to name the main cognitive aspects of notational systems, in order to improve the exchange of experience, opinions, criticism and suggestions. CDN defines 14 cognitive dimensions that constitute the basic vocabulary (Blackwell and Green, 2003, pp. 115-118), as summarized and listed in Figure 1. CDN also defines a method to inspect the system, as described in Section 5.

When we tried to apply the CDN's inspection method, defined by Blackwell and Green (2003), we faced two main challenges. First, the CDN framework was originally proposed to evaluate notational systems for designing artifacts. However, our object of study — a middleware implementation — is generally seen as an artifact, not a notation. So, we had to reinterpret a middleware implementation to view it as a notational system, which the developer can adapt according to his or her needs.

A second challenge is that the 14 cognitive dimensions in CDN (CDs) are conceptual tools defined to help the designer or evaluator to reason about the system. As such, their definition is intentionally broad and subject to different interpretations, so that they can effectively cover a wide range of issues in many different domains. Depending on the inter-

[1] `http://www.jacorb.org`
[2] `http://www.mico.org`

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

Fig. 1 Cognitive Dimensions originally defined by CDN.

Cognitive Dimension	Description
Viscosity	resistance to change
Visibility	ability to view entities easily
Premature Commitment	constraints on the order of doing things
Hidden Dependencies	important links between entities are not visible
Role-Expressiveness	the purpose of an entity is readily inferred
Error-Proneness	the notation invites mistakes and the system gives little protection
Abstraction	types and availability of abstraction mechanisms
Secondary Notation	extra information in means other than formal syntax
Closeness of Mapping	closeness of representation to the domain
Consistency	similar semantics are expressed in similar syntactic forms
Diffuseness	verbosity of language
Hard Mental Operations	high demand on cognitive resources
Provisionality	degree of commitment to actions or marks
Progressive Evaluation	work-to-date can be checked at any time

pretation of each dimension, CDs can overlap or interfere with one another. For example, if we consider a programming environment where the user cannot easily see two files at the same time, the separation of a C++ function declaration and its definition in different files can be seen either as a case of reduced Visibility or as a case of Hidden Dependency, or both. Moreover, if Visibility is interpreted as the user's ability to view whole portions of the system at once, then the Diffuseness of the language can directly influence the Visibility by making portions of the system too large to fit in the viewing area.

These two challenges clearly show that we had to adapt the CDN framework for our particular case. We must define how we are going to systematically interpret each one of the CDs and adapt the steps of the inspection method according to the purpose of the analysis. This is what we call the instantiation of the CDN framework. In the next sections, we describe the instantiation we have used for evaluation of OiL's flexibility, and that we propose to be used in analyzing middleware as notation in general.

4.1 Middleware as a Notational System

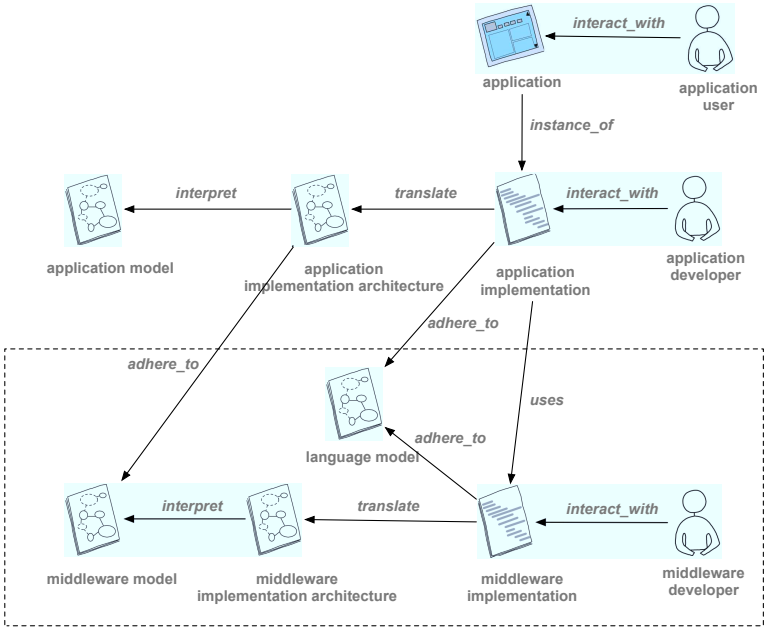
In accordance with Emmerich et al (2007), middleware today is regarded as a layer between network operating systems and applications that aims to resolve heterogeneity and distribution issues, providing appropriate abstractions that application programmers use when building the interacting components of distributed systems. This multi-layered architecture — applications, middleware, network and operating system infrastructure — has different kinds of users that have to deal with multiple models and abstractions in order to accomplish their tasks, as depicted in Figure 2.

We can identify at least three kinds of users of middleware platforms. The first one is the *middleware developer*, who is responsible for:

- defining a *middleware model*, which specifies the abstractions that will be provided to application developers in order to implement their applications;
- designing a *middleware implementation architecture*, which represents a possible realization of the middleware model;
- coding this architecture using a specific programming language in order to provide a specific *middleware implementation*.

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

Fig. 2 The multiple users of middleware platforms and their related models.



The resulting middleware implementation will be the programming platform of the second middleware user, the *application developer*, who is responsible for:

- defining an *application model*, which specifies the entities of the application domain and their relationships;
- designing an *application implementation architecture* based on the underlying application and middleware models;
- coding this architecture using a programming language in order to provide a specific *application implementation*.

Finally, the application implementation will be executed by the *application user* in a specific environment (computer, network, etc.) in order to support some of her/his activities.

Although the scenario depicted in Figure 2 involves multiple actors, models and software artifacts, in this paper we will focus on the activities conducted by the middleware developers and the models they have to tackle in order to accomplish their tasks. As we said before, we are interested in tasks related to the adaptation of the middleware implementation.

A determining factor when interpreting a middleware implementation as a notational system is the programming language used to implement it. The *notation* includes not only the source code, but also the programming language in which the middleware implementation is written. The programming language, and the abstract programming model it represents, must be considered because it potentially influences how the middleware is implemented and organized, and it must be used by the developers to describe (and implement) their intended additions and modifications.

When developing, adapting or extending a middleware implementation, developers have to continuously map and correlate concepts and abstractions from different models. First,

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

there is the mapping (*interpretation*) of the middleware model onto a specific implementation architecture; and both models have to be represented (*translated*) and manipulated in the middleware implementation through concepts and abstractions provided by the programming language. These different mappings are also fundamental factors when considering a middleware implementation as a notational system. We can expect that the closer these models are, the easier it will be to perform a programming task.

4.2 The Interpretation of the Cognitive Dimensions

An important step in the instantiation of the CDN framework is the interpretation of the cognitive dimensions. This interpretation consists of defining our understanding of each dimension in face of the system we want to inspect — middleware implementations — and the tasks we are considering the user performs — adaptation of the implementation. Below, we describe our interpretation of the cognitive dimensions we used in our study.

Viscosity **the amount of necessary changes in the source code to adapt it for a different use.** When we evaluate the Viscosity we look for the number of elements of the implementation that must be modified in each change. These elements can be modules, classes, objects, etc. High Viscosity directly degrades the flexibility of the implementation.

Visibility **how easy it is to visualize related portions of the implementation.** We do not consider the navigation facilities between different parts of the implementation as a Visibility issue because it can be greatly influenced by the tools used by the developer, like text editors and Integrated Development Environments. High Visibility generally increases the flexibility because it contributes for the developer to manipulate the implementation.

Diffuseness **how much text or elements are necessary to implement some part of the system.** If one programming language requires longer descriptions to implement components or if an implementation imposes the definition of more elements like classes or interfaces then we say it is more diffuse. Diffuseness degrades flexibility because it requires more effort to write the new code.

Premature Commitment **the need to assume some aspect of an unknown part of the implementation while developing another.** For example, while writing a component that uses other components developed by third parties, one must assume some interface these components provide, even though they are not available yet. This interpretation is a little different from the one defined originally by CDN, which is more related to the constraints on the order of actions. Premature Commitments are generally problematic for the flexibility because they force the developer to think about something without enough information about it; the conjunction with high Viscosity could increase the problem.

Hidden Dependencies **unexpressed dependencies between different parts of the implementation.** For example, relations determined at runtime are considered Hidden Dependencies, since all static relations can usually be inferred by analyzing the source code. Hidden Dependencies make the implementation more difficult to understand and manipulate thus generally degrade flexibility.

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

Error-Proneness **the amount of possible errors that cannot be detected in an early stage of the development process or are detected inadequately.** For example, errors that can only be detected during the actual use of the middleware are undesirable, because when the middleware is in use, it cannot be easily fixed. Error-Proneness decreases flexibility because the developer has to worry about mistakes that cannot be properly identified in advance.

Progressive Evaluation **the ability to test part of the implementation during development.** We consider that a part of the implementation is complete when it is written and properly verified, either by testing or by some kind of formal verification. Progressive Evaluation lets the developer modify the implementation more easily by doing it incrementally.

Role-Expressiveness **whether the purpose of each element can be inferred by its representation.** A high Role-Expressiveness implies that is easy to determine the purpose an element was designed for, and also to find out if an element is a function, an object, a class, a data structure, or a component. Role-Expressiveness contributes to the flexibility because it makes the implementation easier to understand.

Abstraction **the ability to create new abstractions (Abstraction Tolerant) and the number of abstractions the developer must understand and use (Abstraction Barrier) or create (Abstraction Hunger).** We see as abstractions both the concepts provided by the programming language, like functions or classes, as well as the concepts of the middleware architecture like Mico's invocation adapters. Originally, this CDN dimension comprises three aspects: abstraction tolerance, abstraction barrier, and abstraction eagerness/hungriness. The ability to create new abstractions can help the developer manage the complexity of the implementation thus improving its flexibility. However, the necessity of creating or using too many abstractions can make the implementation very difficult to understand and modify. Since these aspects may have different impacts on software flexibility, in this work we will consider them as three distinct cognitive dimensions (Abstraction Tolerant, Abstraction Barrier and Abstraction Hunger).

Closeness of Mapping **how close the implementation is to the conceptual domain.** In this paper we consider two mappings: the suitability of the implementation architecture to represent the concepts and abstractions defined in the middleware model; and the suitability of the programming language to implement the underlying architecture and to represent and manipulate the concepts and abstractions of the middleware model. A higher suitability implies a higher Closeness of Mapping. Closeness of Mapping improves the flexibility because the implementation is more readily described.

Consistency **how similar is the implementation of elements with similar roles.** For example, if some components are implemented as classes and others are implemented as functions, this is a lack of Consistency. Consistency contributes to the flexibility because it makes the implementation easier to understand.

Hard Mental Operations **operations that require the developer to think about many elements at the same time.** For example, if some information the developer seeks is scattered across the code as a complex structure, then we consider it a Hard Mental Operation. We consider structures with many indirections as complex. The abundance of Hard Mental Operations degrades considerably the flexibility, because the implementation becomes more difficult to manipulate.

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

Provisionality **the ability to change/adapt parts of the implementation in the future.** This dimension is very close to the notion of flexibility. A high Provisionality can alleviate or even avoid some Premature Commitments. For example, the implementation of data structures using C++ templates avoids the need to predict the type of data contained in the structure. In a similar way, dynamic typing allows very generic implementations with dynamic languages like Lua. The interpretive character of Lua also represents a very powerful mechanism to support Provisionality, since it allows the easy incorporation of new code and redefinition of parts of the implementation at runtime.

Secondary Notation **the support for additional information without formal syntax.** It basically involves the programming language support for comments and metadata. A Secondary Notation can improve the flexibility by providing extra ways of explanation.

Figure 3 presents a summary of the correlation between the CDs and the flexibility of a middleware implementation. Since the different aspects of the Abstraction dimension may have different impacts on software flexibility, these aspects are listed in this figure as three distinct cognitive dimensions (Abstraction Tolerant, Abstraction Barrier and Abstraction Hunger). Thus, in the remainder of this paper, we will consider 16 CDs, instead of the 14 original ones.

Fig. 3 The correlation between the CDs and the flexibility of a middleware implementation.

Improve Flexibility	Reduce Flexibility
Visibility	Viscosity
Progressive Evaluation	Diffuseness
Role-Expressiveness	Premature Commitments
Abstraction Tolerant	Abstraction Hunger
Closeness of Mapping	Abstraction Barrier
Provisionality	Hidden Dependencies
Consistency	Error-Proneness
Secondary Notation	Hard Mental Operations

5 CDN Inspection

Once we defined our instantiation of the CDN framework, we proceeded to the actual analysis of the middleware implementations using the CDN inspection method, which is defined by Blackwell and Green (2003) as follows:

1. Get to know the system.
2. Decide what the user will be doing with the notation.
3. Choose some representative tasks.
4. For each step in each task, ask whether the user can choose where to start, how a mistake will be corrected, what if there are second thoughts, what abstractions are being used, and so on for the other dimensions. This will generate an observed profile.
5. Compare the observed profile with the ideal profile for that type of activity.

In the following sections, we describe how we applied the CDN inspection method, presenting the selected middleware implementations (the *objects of analysis*), the selected adaptation tasks, the inspection procedure, the inspection results, an overall analysis of these results, and a triangulation with related work.

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

5.1 Objects of Analysis

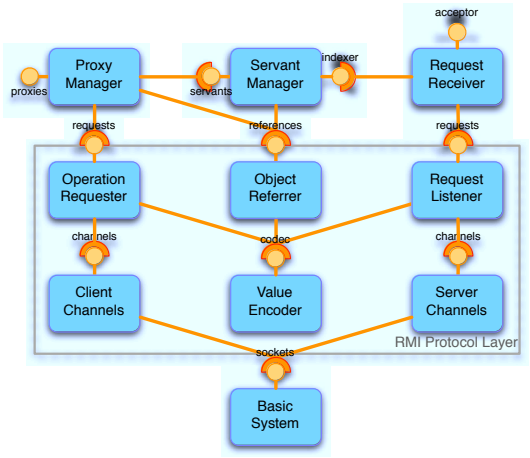
To perform our study, we chose two middleware implementations that we have been using in pedagogical and research projects with graduate and undergraduate students: OiL[3] and Mico[4]. Although the goal of our study is the identification of the factors that justify the presumably greater flexibility of OiL, Mico served as a reference for comparison.

In accordance with their authors (Maia et al, 2006; Puder et al, 2006), both middleware systems were *designed for flexibility*, in order to provide a platform to experiment with middleware implementation techniques. They also implement the same middleware model, defined by the CORBA standard (OMG, 2008). This model specifies a *common distributed object model*, as the main programming abstraction provided to the application developers. Additionally, CORBA specifies a set of communication protocols to enable interoperability between CORBA implementations in different platforms and programming languages.

Despite the fact that OiL and Mico share some design goals, they try to achieve these goals following very different paths. OiL was written in Lua in order to investigate whether the features of a dynamic language like Lua can facilitate or not the development of a middleware system. In this work we consider that a dynamic programming language is characterized by interpretation, dynamic typing, reflection, and autonomic memory management.

OiL's implementation follows a component-based architecture. The ultimate goal of its design is to provide flexibility for further extensions and modifications to adapt the middleware to different uses. A minimum set of components implements the core functionality, like invocation dispatching, remote object proxies, and synchronous or asynchronous invocations. This basic set of components does not implement features related to a specific RMI protocol. Instead, it only defines receptacles where components that implement different RMI protocols can be connected to. The set of components that implements a specific RMI protocol defines the OiL's *RMI Protocol Layer*. Figure 4 illustrates the main components that define OiL's component-based architecture, which can be reconfigured to adapt its implementation.

Fig. 4 OiL's component-based architecture.

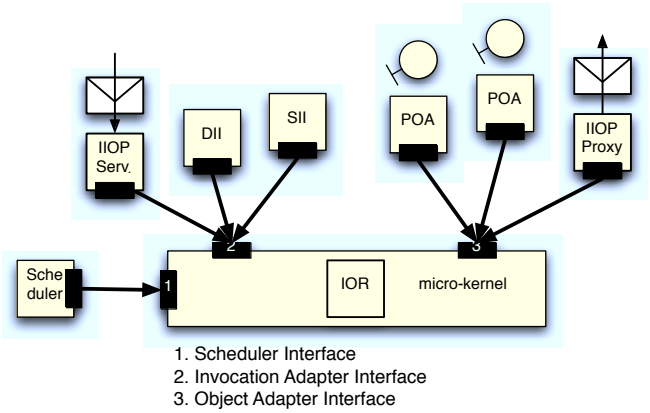


[3] http://oil.luaforge.net/
[4] http://www.mico.org

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

Mico is a CORBA implementation written in C++ and its architecture is based on the concept of a microkernel, a core component all other components (*plugins*) are attached to, as illustrated in Figure 5. These components can be invocation adapters, which receive invocations from the application or from the network, or object adapters, which dispatch invocations to local servants or to remote peers through the network. The microkernel gets requests from *invocation adapters* and dispatch them to a proper *object adapter*. The microkernel also uses a scheduler component to manage independent tasks and system events. The microkernel architecture provides extensibility through the addition of new plugins.

Fig. 5 Mico's micokernel-based architecture.



We can summarize the differences between OiL and Mico in two key factors:

- *the programming language*: OiL was written in Lua to take advantage of features that are common among dynamic programming languages in order to improve flexibility; Mico was written in C++, which is a more static programming language, frequently used to implement middleware systems due to its good performance.
- *the implementation architecture*: OiL follows a (fully) component-based architecture, while Mico adopts a microkernel architecture.

5.2 Selected Tasks

Since our purpose in this study is to evaluate flexibility, that is, how easy it is for a developer to change an implementation for different purposes, we selected representative adaptation tasks that exercice three common types of adaptation: change of an internal aspect of the middleware implementation; change in the programming interface provided to the application; change in the set of functionalities provided by the middleware. The three selected adaptation tasks are described below.

Switch the underlying RMI (remote method invocation) protocol. This task represents a change of an internal aspect of the middleware implementation. We considered the implementation of two significantly different protocols: CORBA's GIOP and LuDO (Lua Distributed Objects). We specifically devised LuDO to mimic Lua's method invocations, in

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

similar fashion to other language-specific RMI protocols like JRMP (Wollrath et al, 1996) and PYRO[5]. We believe such a change would have a major impact on the overall implementation, affecting multiple middleware components.

Add support for asynchronous invocations. This task represents a change in the programming interface provided to the application. In fact, it extends not only the middleware API, but also the programming model supported by the middleware. Such changes should have little impact in the middleware core, but should considerably impact the provided API.

Add and remove features like support to issue remote method invocations (client-side), creation of remotely accessible objects (servants), multithreaded invocation dispatching, invocation interception, and logging. This task represents a change in the set of functionalities provided by the middleware. This sort of change aims to make the middleware lighter and possibly more adequate to situations where many functionalities are unnecessary.

5.3 Inspection Procedure

Figure 6 summarizes the main information about the setup of our CDN inspection study. As stated in Section 4.1, in this paper we focus on middleware developers that have to perform some modification in a middleware implementation. We will target an expert developer that can be either a professional developer adapting the middleware for some particular application or an advanced student or researcher doing an experimental modification in the middleware. In this initial study we have not considered eventual learning issues faced by novice developers.

The inspection reported in this paper was performed by four analysts: two experts in middleware implementation and the programming languages used in OiL and Mico (MW Expert A and B); an expert in Lua programming, with previous knowledge in qualitative research methods (Lua Expert); and an expert in HCI, linguistics and qualitative research methods (HCI Expert).

MW Expert A, which is the lead developer of OiL, performed the selected adaptation tasks in both middleware implementations. During the implementation of these tasks, MW Expert A tried to identify the necessary actions and possible design decisions a developer might make, and discussed with MW Expert B and Lua Expert the main challenges he faced. Then, MW Expert A reported his observations guided by the dimensions proposed by the CDN framework. This initial report was revised by MW Expert B and Lua Expert in order to enrich the report with their perspectives. Finally, HCI Expert revised the second report in order to strengthen the cognitive interpretation. HCI Expert did not have previous knowledge about middleware models and their implementations.

The combination of multiple analysts with different perspectives and expertise in different subjects was very important to refine the inspection report, and worked as a mechanism for internal triangulation. Since MW Expert A is the lead developer of OiL, this internal triangulation was important to filter any perceived bias in favor of OiL. The following sections present our consolidated inspection report.

[5] `http://pyro.sourceforge.net/`

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

Fig. 6 Summary of the inspection setup.

research question	which are the major factors that affect the flexibility of a middleware implementation? how are they related to each other?
inspected notations	OiL and Mico
target group	expert middleware developers
observed tasks	switch the underlying RMI; add support for asynchronous invocations; add and remove middleware features
inspection procedure	an expert developer performed the selected tasks and reported his observations in accordance with the CDs; his report was revised by another middleware expert, a Lua expert, and a HCI expert specialized in linguistics and qualitative research methods

5.4 Inspection Results

After the proper instantiation of the CDN framework to support our study on middleware implementation flexibility, we implemented the three tasks previously defined, using the two middleware platforms. The inspection of these implementations allowed us to draw several conclusions and observations about the level of flexibility provided by both platforms being evaluated. As described below, many of our observations are common to all three tasks, but some cognitive dimensions could be better examined in some tasks than in others.

Viscosity The modularity provided by OiL's components and the separation of Mico's implementation in microkernel and attached adaptors simplified some of the modifications, because it allowed some changes to be introduced simply by adding connections to new components. For example, the support for a new RMI protocol can be done by new components of the RMI Protocol Layer in OiL or as a new pair of invocation adapter and object adapter in Mico. On the other hand, Mico's microkernel presents a high viscosity because it is built as a monolithic component. Thus, changes in functionalities inside the microkernel usually imply changes in many of the classes that implement it. For example, the object reference model used in Mico is based on CORBA's IOR and is implemented by the microkernel. In OiL, the functionality of Mico's microkernel is modularized in separate components that can be replaced more easily. Moreover, the absence of type declarations due to Lua's dynamic typing gives reduced Viscosity to OiL as compared to Mico/C++. However, the dynamic typing also brings difficulties as will be discussed later.

Visibility While reading the different portions of code to perform the tasks in both implementations, it became apparent that the main issues related to Visibility in both implementations are the size of the code and the separation in multiple files. The larger the code is, the more difficult it is to read it. In that sense, C++ code tends to be larger than Lua code due to the necessity of type declarations. The size concern is also subject to the Diffuseness dimension, but Visibility also includes the separation of declaration and definition in different files — `.cpp` and `.h` — which also makes it more difficult to view the whole implementation in general.

Diffuseness As stated previously, the need for explicit type declarations significantly contributes to make Mico's implementation longer than OiL, where such declarations do not

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

exist. Lua tables also contribute to keep OiL's implementation concise because it makes the implementation of dynamically typed data structures trivially implemented in a few lines of code. One example of this is the representation of object references in the implementation of CORBA's GIOP. In OiL, these references are represented by a couple of Lua tables, while in Mico it is necessary to define classes to implement a dynamic data structure to cope with the dynamic nature of these references. In C++, the implementation of similar structures is more complex and lengthy, however this is in part a consequence of the low Closeness of Mapping of the language.

Premature Commitments One important characteristic in Mico's implementation is the adherence to the CORBA standard, which is a very general model that can accommodate different features. However, many aspects of the CORBA model that are incorporated by Mico's implementation can be inadequate for other components or for future modifications (*e.g.* the mandatory support for multi-component profiles in object references, which is not used anywhere in the current implementation of CORBA's GIOP). In our interpretation, we see the strict adoption of the CORBA model throughout Mico's implementation as an attempt to predict necessities of future modifications. The main problem with such Premature Commitments is that they can become superfluous or limiting. In the implementation of the LuDO protocol in Mico, most of these features were superfluous (*e.g.* location forward) or inadequate (*e.g.* stringfied IORs). Since these features are implemented in the micro-kernel, they cannot be easily removed and the developer adapting the middleware must take them into consideration most of the time.

OiL avoids the Premature Commitment to the CORBA model by using dynamically typed structures and objects that are opaque to core components and can be dynamically extended or inspected by components that implement CORBA specific features. A similar approach can be adopted in Mico but with a reduction of the effectiveness of C++ static type checking. Generally, we believe that static typing promotes Premature Commitments in the implementations, reducing flexibility in favor of a more robust type system.

Hidden Dependencies During the task of adding and removing selected features from OiL, it was useful that OiL's components do not make explicit reference to other components, so they can be reused in different contexts. This makes the assembling of different middleware implementations with distinct features easier (reduces Viscosity). However, it introduces Hidden Dependencies between components, which reduces readability. Mico makes some of these dependencies explicit by creating all the required objects inside the component where they are used. These explicit relations also allow static type verification by the compiler.

In case of Mico, the use of explicit type declarations helps to make some dependencies more explicit. For example, if a component creates a data structure with a formal type, we can find other components that manipulate this structure by looking for variables of that formal type. In Mico, this is was apparent when we tried to remove the support for object adaptors, which all implement interface `CORBA::ObjectAdapter`, so to identify all parts of the code that implement or require such component, one can simply search for this interface. In OiL, such dependencies are not so obvious without relying on code documentation. For that reason, we believe dynamic typing leads to more Hidden Dependencies.

Likewise, computational reflection can also hide more dependencies, since it allows runtime manipulation of metadata related to the implementation's entities. This metadata manipulation may establish new relationships between the implementation's entities that cannot be identified through code inspection. In OiL, this was particularly confusing when

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

trying to understand the implementation of object proxies to implement asynchronous invocations, which relies heavily on computation reflection mechanisms of Lua.

Error-Proneness C++'s static typing can identify some errors that, in Lua, can only be identified at runtime through dynamic typing. This was more evident while trying to assemble different middleware implementations with distinct features. In Mico, the compiler can easily identify when a component is removed but some reference to it remains in the code. In OiL, this kind of error is usually only identified at runtime. Therefore, without an automatic test suite that exercises the middleware functionalities, the use of a dynamically typed language increases the Error-Proneness. On the other hand, errors like incorrect type casting or memory management errors are not detected by the C++ compiler nor by the runtime environment[6]. Therefore, the use of a test suite is also important to detect these errors. In general, we consider that both implementations have equivalent Error-Proneness.

Progressive Evaluation An important advantage of dynamic typing and interpretation is the possibility of testing incomplete or partial implementations. This allows the developer to evaluate some design decisions earlier and to avoid wasting time following unsuitable approaches. Such characteristic was very useful while implementing support for different protocols in OiL, making OiL well suited for exploratory modifications. Since Mico must be type-corrected to be executed, the experimentation of different implementation approaches takes more time. In practice, it was necessary to write almost an entire implementation of LuDO support for Mico before we could run any test. Moreover, the possibility of quickly and continuously testing the OiL's implementation were used to proof frequently the implementation against type errors, thus reducing the impact of the Error-Proneness of dynamic typing.

Role-Expressiveness Both implementations seem to have a low Role-Expressiveness. While reading code to perform all the three tasks, it was evident that objects, data structures, interfaces and components of the architecture are all implemented as classes in Mico and as Lua tables in OiL, thus looking very similar to each other. We believe both implementations are equivalent in this dimension.

Abstraction In both Mico and OiL, the developer can create new abstractions, either using C++ classes or Lua tables and metatables. Moreover, the modifications considered in this study did not require the use of new abstractions. However, the set of abstractions the developer must master to perform each selected task varies according to the middleware implementation.

Abstraction Barrier in Mico consists basically of the abstractions defined by the language (*e.g.* C++ classes), CORBA abstractions that are used in the implementation (*e.g.* IOR profiles) and abstractions defined by its architecture (*e.g.* microkernel and adapters). Both C++ and CORBA are usually regarded as feature-rich models that define many abstractions. Lua, on the other hand, is a simpler language with less abstractions than C++, and OiL tries to avoid many of the complexities of CORBA, which results in a smaller Abstraction Barrier. However, OiL defines new abstractions that extend Lua with support for classes, components and architecture templates that the developer must learn prior to understand and

[6] Incorrect type casting can be detected at runtime in C++ by the use of the `dynamic_cast` operator, however its usage is avoided in Mico's implementation, probably due to performance and portability issues.

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

adapt its implementation. OiL uses Lua's support for computational reflection to create dynamic proxies for remote objects, thus the modification to support asynchronous invocations requires that the developer knows and uses Lua's abstractions related to this functionality (i.e. computational reflection and coroutines).

Closeness of Mapping In order to discuss this dimension of OiL and Mico, we should take into account the multiple mappings between different models that a developer has to face when working on a middleware implementation, as described in Section 4.1. More specifically, we have to analyze how the implementation architecture is represented by the programming language abstractions, how the middleware model is represented by the programming language abstractions, and how the implementation architecture interprets the middleware model.

The implementation of Mico and OiL are based on plug-able components that exchange dynamically typed information through a core component. In that sense, Lua's dynamic typing and built-in support for dynamically typed structures (tables) make the implementation of component interactions easier. In other words, the dynamic typing nature of Lua is suitable to represent some of the common component interactions in middleware implementations. However, neither Lua nor C++ has proper mechanisms to represent explicitly many of their architectural elements, specially the overall architecture.

At first sight, Lua and C++ are suitable to represent the programming abstractions defined by the CORBA (programming) model, since both languages support object-oriented programming. However, our inspection showed that Mico is more suitable to implement the CORBA's GIOP protocol, due to similarities between GIOP's invocation model and the standard method call mechanism of C++. GIOP was designed to provide RMI for statically typed languages and was heavily influenced by C-based RPC systems. On the other hand, the inspection also showed that OiL is more suitable for modifications that diverge from CORBA's model, such as the implementation of the LuDO protocol, which is designed to mimic Lua's method invocations over a network.

OiL was also more suitable to represent and implement the support for asynchronous method invocations, mainly due to the modularization provided by OiL's architecture and to some characteristics of Lua, such as dynamic proxies, coroutines and functions as first class values.

Mico adopts an architecture strongly influenced by CORBA, with a direct mapping of most elements of CORBA's model onto the implementation architecture. On the other hand, OiL follows an architecture based on more generic concepts, which is more suitable to implement more divergent middleware models, but also introduces extra mapping overhead when representing elements of CORBA's model.

Hard Mental Operations The implementation of many data structures in Mico are complex, involving multiple indirections implemented as pointers. The management of these structures generally requires a lot of cognitive resources from the developer — especially for memory management. This concern with this kind of complexity was present in all the three tasks we performed in the implementation since we had to understand the original implementation and keep track of the implications of our modifications over the code. In OiL, that problem is completely absent due to the support for automatic memory management of Lua. Moreover, Lua tables usually simplifies the implementation of data structures, reducing the number of indirections when compared to their C++ equivalents.

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

Provisionality Lua provides many mechanisms to support Provisionality. The possibility of interpreting new code anytime actually allows the modification of almost any part of the implementation at runtime. Lua tables can be seen as generic elements that can be adapted to many purposes. For example, while object references in OiL are created as tables with some predefined fields, components can extend this structure with additional fields or operations according to its needs. The combination of dynamic typing and reflection mechanisms in Lua enables the implementation of *generic proxies*, which can easily handle different types of crosscutting concerns. Moreover, the reflection support helps to define more specific behaviors for elements created earlier. For instance, such feature was used to implement the support for interception of remote method invocations in the task to introduce and remove features in the middleware.

Provisionality in Mico is restricted to anticipated extensions, and it is mainly supported by plugins to its microkernel-based architecture and by basic C++ features, such as templates, polymorphism, late-binding and generic pointers that hold arbitrary data.

Secondary Notation Secondary Notation is supposed to play an important role in any programming task, since it provides additional information about the implementation. The material provided by Puder et al (2006) was fundamental to understanding the Mico's implementation architecture, since high-level design choices like this one are not properly represented in the source code. With regard to OiL, in our study we could not observe any specific issue related to the Secondary Notation dimension, because the developer that performed the analyzed tasks is the lead developer of OiL.

Consistency In our study, we could not observe any specific issue related to the Consistency dimension.

Figure 7 presents a summary of the inspection results. In this figure, the cognitive dimensions are organized in two groups: The first one (left side) has the dimensions that contribute to improve flexibility of middleware implementations; The second one (right side) has the dimensions that contribute to reduce flexibility. We indicate how much each middleware implementation has contributed to strengthen or weaken each particular dimension. Considering our goal of improving flexibility, more bullets (● symbol) and less circles (○ symbol) in the left side implies more flexibility, while more bullets and less circles in the right side implies less flexibility. The results summarized in the Figure 7 support our original observation that OiL is more flexible than Mico.

Fig. 7 Summary of the flexibility evaluation in accordance with CDN.

Improve Flexibility	**OiL**	**Mico**	**Reduce Flexibility**	**OiL**	**Mico**
Visibility	●	○	Viscosity	○○	●●
Progressive Evaluation	●●	○	Diffuseness	○○	●●
Role-Expressiveness	○	○	Premature Commitments	○○	●●
Abstraction Tolerant	●	●	Abstraction Hunger	NO	NO
Closeness of Mapping	●	●	Abstraction Barrier	●	○
Provisionality	●●	○○	Hidden Dependencies	●●	○○
Consistency	NO	NO	Error-Proneness	●	●
Secondary Notation	NO	●	Hard Mental Operations	○○	●●

NO = not observed
● = strengthens the dimension
○ = weakens the dimension

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

5.5 Conclusion of the Analysis

The objective of our inspection was not only to identify the characteristics of each middleware platform under consideration that influence their flexibility from a human-centric perpective, but also to provide means to explain their cognitive impact. In the previous section, we identified many characteristics of both middleware platforms that can increase the cognitive overload faced by a developer when adapting middleware. This cognitive overload is especially critical when the developer wants to quickly try different design and implementation alternatives (*exploratory programming*).

In our interpretation of the cognitive dimensions, we identified a relationship between them and the linguistic aspects of the notations under analysis. For instance, Visibility is directly related to syntactical aspects of the notations, while Abstraction Barrier is related to semantic aspects. Provisionality, in its turn, has to do with use situation, which we classify as a pragmatic issue. Figure 8 reorganizes the results of our inspection in accordance with the relation between the cognitive dimensions and the linguistic aspects (syntax, semantics and pragmatics). In this figure, having more bullets implies more contribution to strengthen flexibility and having more circles implies more contribution to reduce flexibility.

Fig. 8 Summary of the flexibility evaluation organized by linguistic aspects.

Syntax-related Dimensions	**OiL**	**Mico**
Visibility	●	○
Viscosity	●●	○○
Diffuseness	●●	○○
Premature Commitments	●●	○○
Hidden Dependencies	○○	●●
Consistency	NO	NO
Semantics-related Dimensions		
Abstraction Tolerant	●	●
Abstraction Hunger	NO	NO
Abstraction Barrier	○	●
Pragmatics-related Dimensions		
Role-Expressiveness	○	○
Error-Proneness	○	○
Progressive Evaluation	●●	○
Hard Mental Operations	●●	○○
Closeness of Mapping	●	●
Provisionality	●●	○○
Secondary Notation	NO	●

NO = not observed
● = strengthens flexibility
○ = weakens flexibility

We can observe in Figure 8 that OiL outperforms Mico in syntax- and pragmatics-related dimensions, while Mico has a slightly better result in semantics-related dimensions. Considering only syntax-related dimensions, OiL strengthens flexibility in all dimensions, except in Hidden Dependencies. The poor result in this dimension encourages further investigation, in order to explore possibilities for improvement.

The best OiL's results among the pragmatics-related dimensions were with Provisionality, Progressive Evaluation and Hard Mental Operations. These dimensions, together with a low Viscosity, are especially important to exploratory programming. Such dimensions are important to allow implementations to be modified quickly and with less effort, so different

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

approaches can be easily evaluated. Moreover, the possibility of quickly and continuously testing the middleware implementation may reduce, or even counterbalance, the high Error-Proneness of dynamic typing. The high degree of Provisionality is also very meaningful, since our interpretation of this dimension places it very close to the notion of flexibility.

As discussed in Section 5.4, although OiL seems to have drawn with Mico in the Closeness of Mapping dimension, they performed very differently when we consider the multiple mappings that the developer has to deal with. Their performance in this dimension also varies a lot according to the nature of the adaptation task. OiL aims to provide an architecture that enables the implementation of more divergent middleware models. Mico adopts an architecture strongly influenced by CORBA, whose main points of extensibility are directly related to the points of variability defined in the CORBA's model (protocols, concurrency models, etc.). Mico will probably perform better if the adaption task fits CORBA's model, otherwise OiL should present better results.

Our CDN inspection also casts light on the reasons behind such observations, identifying how each OiL's characteristic influences its flexibility. Considering our inspection, OiL's most distinguishing characteristics in contrast to Mico are the component-based architecture, lack of type declarations, dynamic typing, computational reflection, interpretation, automatic memory management, and the use of Lua tables as a unified data structure. Excepted for the architectural style adopted by OiL, all the other characteristics are related to the programming language used to implement it. The majority of these language-related characteristics are common to other dynamic languages.

Figure 9 presents the influence of OiL's features on each cognitive dimension. The second and third columns indicate OiL's features that influence positive and negatively its flexibility, respectively. Each feature is represented by an acronym, and the features related to dynamic languages are in boldface. This figure allows us to draw many conclusions.

Fig. 9 The influence of OiL's features on its flexibility.

Syntax-related Dimensions	**strengthen flexibility**	**weaken flexibility**
Visibility	LS,**DT**	
Viscosity	SC,**DT**	
Diffuseness	LS,**DT**,LT	
Premature Commitments	SC,**DT**	
Hidden Dependencies		SC,**DT**,**CR**
Semantics-related Dimensions		
Abstraction Tolerant	**CR**,LT	
Abstraction Barrier		SC,**CR**
Pragmatics-related Dimensions		
Role-Expressiveness		LT
Error-Proneness		**DT**
Progressive Evaluation	**DT**,**CI**	
Hard Mental Operations	**MM**,LT	
Closeness of Mapping	**DT**,LT,OP,CO,FF,SC	**DT**,SC
Provisionality	**CI**,**CR**,**DT**,LT,SC	

Legend:

CI	code interpretation	SC	software components
DT	dynamic typing	LS	Lua syntax
CR	computational reflection	LT	Lua tables
MM	automatic memory management	CO	coroutines
OP	object-oriented programming	FF	functions as first-class values

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

First, the component-based implementation architecture improves flexibility as a means of reducing Viscosity and Premature Commitments, and increasing Provisionality and the Closeness of Mapping between the implementation architecture and the middleware model. However, such architecture introduces some Hidden Dependencies, which are harder to find due to the lack of type declarations.

The lack of a proper support for components in the programming language also represents an Abstraction Barrier and makes the mapping between the implementation architecture and the programming language more difficult. In fact, this is one of the most frequent complains reported by students that modified OiL's implementation.

Considering the four characteristics derived from the dynamic nature of Lua, we can observe that dynamic typing has a strong positive influence on syntax-related dimensions. However, dynamic typing and computational reflection, together with the component-based architecture, are responsible for a high degree of Hidden Dependencies, which was the worst result of OiL when compared to Mico in all dimensions.

While dynamic typing has no influence on semantics-related dimensions, it plays an important role from a pragmatic perspective, supporting Progressive Evaluation and Provisionality, and simplifying the representation of some common component interactions in middleware implementations (Closeness of Mapping between the implementation architecture and the programming language). Dynamic typing also provides, together with code interpretation, a good support for Progressive Evaluation. As described in Section 5.4, we used this feature to test frequently OiL's implementation against type errors. Further quantitative studies can be carried out in order to assess the impact of dynamic typing on the overall implementation process, considering its contributions to Progressive Evaluation and Error-Proneness.

Computational reflection is a major adaptation mechanism, allowing the runtime inspection and modification of the middleware implementation. However, it introduces some Hidden Dependencies and can represent a major Abstraction Barrier. The use of reflective mechanisms should be considered carefully, to avoid a high increase in the cognitive overload faced by a developer.

The high degree of Provisionality can be easily explained by the amount of OiL's features that contribute to this dimension. Again, the dynamic features of Lua are the main determinants of this good result.

Since the Closeness of Mapping dimension in our CDN instantiation has to deal with multiple mappings, some of OiL's features appear in both columns. As discussed in Section 5.4, features like dynamic typing and software components can help to close the gap between some models, but they can also increase the gap in other cases. For instance, the use of a component-based architecture can help to represent the middleware model, but it can make the translation of the implementation architecture in terms of the programming language's abstractions more difficult.

With this analysis, we achieved our original goal of identifying the factors that justify a presumably greater flexibility of OiL. The implementation architecture and the programming language used to implement the middleware are major determinants of high flexibility. The dynamic features of Lua play a very important role, providing means to simplify not only the syntactical manipulation of the notation, but also the execution of common programming tasks, at least in the domain of middleware implementation.

But there is room for improvement. Our observations encourage the investigation of better programming language abstractions and constructions to support component-based programming. For instance, Eichberg et al (2005) and Rouvoy and Merle (2009) tried to provide better support for component-based programming through *annotations* in the source

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

code (a Secondary Notation). The evaluation of this approach with our CDN instantiation is an interesting direction for future work.

The lack of mechanisms in the programming language to represent and manipulate the implementation architecture in a proper way can also hinder adaptation tasks, by introducing some Hidden Dependencies. Approaches like the one proposed by Aldrich et al (2002) can be applied in order to solve or mitigate this problem.

However, all these alternative approaches are based on adding more type information to the code, what could be antithetical to the idea of dynamic typing as a major determinant of flexibility. In fact, although dynamic typing has been shown to be very effective to improve flexibility in many dimensions, it also increases the cognitive overload in other dimensions. Siek and Taha (2007) propose a promising approach, called *gradual typing*, to tackle this apparent contradiction. Gradual typing allows parts of a program to be dynamically typed and other parts to be statically typed.

5.6 Triangulation with Related Work

Traditionally, middleware implementations have tried to achieve flexibility either through architectures with a rich set of features provided by the middleware (e.g. OMG, 2008) or through architectural mechanisms that allow middleware adaptation and extension (e.g. Kon et al, 2002; Ramdhany et al, 2009). The results of our CDN inspection corroborate the importance of the implementation architecture to the middleware's flexibility.

The results we observed in this work are also consistent with results of other researchers working on flexibility of middleware implementations. For example, some related work (Cacho et al, 2006; Costa et al, 2007; Ramdhany et al, 2009), which applies software engineering metrics to evaluate middleware flexibility, relies on the assumption that characteristics like low coupling, high conciseness and cohesion, and good separation of concerns, contribute to increased flexibility. This assumption is generally compatible with the instantiation of the CDN framework used in this work and with some conclusions of our analysis.

In particular, low coupling can contribute to minimizing the likeliness of Hidden Dependencies and reduce Viscosity — changes in one part of the implementation are less likely to propagate or affect others. High conciseness is directly related to reduced Diffuseness, which is also a CDN dimension that reduces flexibility in our view. Finally, high cohesion and good separation of concerns is likely to reduce Hard Mental Operations, since the programmer does not have to keep many concerns or functionalities in mind when learning or modifying the code. In this view, we consider our work complementary in the sense that the CDN embraces additional aspects that cannot be easily measured quantitatively, but may have great influence on the flexibility of the implementation. Moreover, qualitative analysis can also help to improve the understanding of how these quantitative metrics must be interpreted.

Our work is also complementary in the sense that it considers not only the impact of the middleware implementation architecture, but also the influence of the programming language. In accordance with Vinoski (2003a), most of the middleware complexity resides precisely in the mapping between the middleware's underlying model and the abstractions of a particular programming language. This is consistent with our interpretation of the Closeness of Mapping dimension.

Our observations about the role of the dynamic characteristics of Lua are also consonant with related work on dynamic languages. For example, Nierstrasz et al (2005) claim that most languages are static in the sense that they assume the world is consistent, but in reality

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

most complex systems cannot be consistent, and they later suggest that dynamic languages can be a good solution to this problem. This line of reasoning is consistent with our claim that dynamic languages provide better Progressive Evaluation because they allow the system to run even when it is not entirely correct or consistent. Mechanisms for Provisionality can be also very helpful in such situations.

They also claim that static typing "is the enemy of change". This is consistent with our interpretation that static typing contributes to more Premature Commitments and higher Viscosity. Nierstrasz et al also highlight the fact that, although computational reflection is useful in many situations, it also leads to less stability. This is also consistent with our observation that the use of computational reflection in OiL introduces Hidden Dependencies, which may lead eventually to higher Error Proneness.

Hirschi (2007) reports a low Error Proneness in Lua, due to more informative error messages, which is consistent with our claim that Lua's dynamic typing catches errors at runtime that are never identified by C++'s static typing. He also recognizes the Lua's meta-mechanisms as powerful Abstraction Tolerance mechanisms that allow implementation of classes and inheritance in Lua, which does not provide direct support for such object-oriented concepts. Hirschi also reports, based on his experience using Lua, the advantages of Progressive Evaluation by stating he could set up tests faster and identify bugs sooner than when using C or C++. Another advantage of Lua reported by Hirschi is the Provisionality provided by the language's dynamic nature, that ensured he "could always implement new insights at a moment's notice". Finally, he also suggests that most of Lua advantages may not be exclusive of this language, but shared by most dynamic languages. This is consistent with our correlation between CDN dimensions and dynamic language features.

Our results are consistent with the findings of Prechelt (2003) too. The author evaluates the use of four scripting languages (Perl, Python, Rexx and Tcl) and three more conventional programming languages (C, C++ and Java). In his study, Prechelt uses a quantitative analysis based on 80 implementations of the same program. Prechelt's findings indicate that the programs in scripting languages are very much smaller than the other ones. Examining the code of these programs, he identifies three possible reasons: the lack of type declarations (similarly to our analysis of the Diffuseness dimension); more powerful constructs for common operations like the hashing algorithm of associative arrays — a common feature in scripting languages — that were extensively used in programs written in scripting languages. We can interpret the last two reasons as a higher expressiveness of the scripting languages evaluated and a closer mapping between the proposed program and these languages, which is similar to our findings about the suitability of OiL's features to implement the modifications defined in our study.

Moreover, Prechelt's findings also indicate that programs in scripting languages are written in less time than in other languages. This is consistent with our idea that dynamic typing promotes Progressive Evaluation and helps identify inadequate approaches earlier and reach a satisfactory solution faster. Although the consistency between our results and Prechelt's findings does not assure the results, the absence of divergencies is more important because it indicates that there are no obvious misconceptions in our study. This is a very illustrative example of the synergism that can be achieved when qualitative and quantitative research methods are combined.

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

6 Final Remarks and Future Work

Traditionally, the evaluation of middleware and other programming infrastructures focuses mainly on aspects like performance or scalability, and issues like how easy it is for the programmer to use or comprehend middleware are left behind, even though these are recognized as important concerns in middleware design (Vinoski, 2003b). We believe one reason for this is the difficulty of using quantitative approaches, which are traditionally used in middleware research, to evaluate usability. Qualitative methods on the other hand stand out as a promising and complementary alternative, since they have been successfully used by the HCI community to evaluate the user experience in other scenarios.

In this paper, we presented our first experience in investigating the use of qualitative research methods to evaluate middleware. We based our investigation on the CDN framework due to its broad applicability, ranging from evaluation of visual programming languages to graphical interfaces and software in general. However, the generality of the CDN framework also makes its application very difficult for the novice. In our case, we had to define a specific instantiation of CDN for evaluation of middleware flexibility. Such an instantiation consists of more specific interpretations of the CDN framework in the target evaluation context. These interpretations include: a view of middleware as a cognitive artifact being manipulated by the programmer; more precise definitions of the cognitive dimensions adjusted to this scenario; and a systematic method for evaluation based on specific cases of middleware modification. A proper instantiation of CDN requires a deep understanding of the CDN framework and a careful analysis of the objectives of the study.

The main contributions of this work are twofold. First, we described a qualitative CDN-based method to analyze the cognitive effort made by programmers while adapting middleware implementations. And second, we reported on the inspection of two platforms designed for flexibility, suggesting that certain programming language design features might be particularly helpful for developers in order to improve flexibility.

Although there is no guarantee that our instantiation of CDN is the most adequate for evaluation of middleware flexibility, the results we obtained were very interesting, pointing out specific characteristics of Lua or of OiL's design that influence the flexibility of the implementation. For example, Lua features like interpretation, dynamic typing and reflection are useful for exploratory changes. Moreover, automatic memory management (garbage collection) simplifies changes in general. The organization of OiL's implementation as modular components makes the implementation easier to modify yet more difficult to understand and investigate. In summary, the dynamic language features of Lua generally contribute to OiL's flexibility, and the use of components contributes to modularizing its implementation.

Despite the fact that other authors have already promoted dynamic languages as an instrument to improve flexibility (Ousterhout, 1998; Nierstrasz et al, 2005; Hirschi, 2007), to the best of our knowledge no previous work has established precisely the relationships between the features of dynamic languages and the flexibility these languages provide.

On the other hand, the comparison with Mico can be deceiving in some aspects. First because C++ is not the ideal candidate to praise for the benefits of statically typed languages. Other languages may play a better role in offering important features that Lua lacks and would be useful to improve the flexibility of a middleware implementation. Nevertheless, the comparison between OiL and Mico is still pertinent since C++ is often considered the language of choice for implementing high quality middleware. The results of our analysis can also be seen as identification of the particular characteristics of the C++ language and C++ coding techniques that affect the implementation's flexibility. For example, the lack of support for automatic memory management and dynamically typed structures compromises

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

the flexibility of Mico's implementation by making it more difficult to understand, change and introduce independent developed components.

As a typical qualitative study, this work was carried out in a very specific context, focusing on a small number of evidence sources, privileging in-depth analysis rather than widespread coverage. The conclusions drawn from our CDN inspection are restricted to a context defined by the two middleware implementations, the selected tasks, and the backgrounds of the four analysts. These conclusions cannot be directly generalized to other context without further qualitative and quantitative studies. As said before, there is no guarantee that our instantiation of CDN is the most adequate for evaluation of middleware flexibility. The application of this instantiation to other programming tasks and to other middleware platforms would be very important to validate and improve our interpretation of the CDN dimensions. Although we have already performed triangulation with related work and internal triangulation, through the combination of multiple analysts with different perspectives and expertise in different subjects, further triangulation is still needed in order to control bias and to test (or maximize) the validity and reliability of our study.

To the best of our knowledge, this work is a first attempt to define an evaluation methodology for programability of middleware platforms. It needs further improvement and validation, however we believe it supports our argument that the use of qualitative methods can be regarded as a valuable approach to understand the way developers interact with middleware platforms and other programming tools. This understanding is an important step to guide future middleware research that takes into account the programmer's perspective. We can envision many directions in which this work can evolve, as listed below:

- Perform similar studies to evaluate the flexibility of other middleware platforms, such as OpenCom (Coulson et al, 2002). OpenCom is a reflective middleware architecture also designed to provide a high degree of flexibility, and has implementations in different languages, such as Java and C++. Since reflective middleware provides an API (or a *meta-object protocol*) that allows the inspection and modification of the middleware at runtime, a reflective middleware platform would allow studies where the application developer performs the adaptation tasks at runtime. This sort of scenario would enable the investigation of other user perspectives and other relationships between abstract models and their representations.
- Repeat the instantiation process to evaluate other programability aspects, such as the ease of developing applications using middleware. These studies can investigate issues related to the distributed runtime environment, concurrency models, and the multi-language support provided by some middleware platforms.
- Perform similar studies to investigate whether the enhancement of an object-oriented language with component-oriented programming abstractions and architecture descriptions as a first class entity can improve the programmer experience when implementing component-based architectures.
- Perform similar studies to investigate whether gradual typing (Siek and Taha, 2007) effectively combines the advantages of static and dynamic typing in order to provide more flexible programming platforms.
- Extend CDN with new dimensions to support deeper analysis of programming abstractions and platforms. In particular, we are interested in the identification of possible epistemic dimensions that deal with the ability of the notation to make the user think more about the design being made. We are also interested in investigating whether the correlation we identified in this work between the CDs and the linguistic aspects appears in other CDN instantiations.

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

– Perform experiments based on user (programmer) observation. Such experiments should provide new insights into the inspections already accomplished.
– Investigate how qualitative and quantitative methods can be combined in a symbiotic manner to provide a better understanding of how to design programming abstractions and platforms.

Acknowledgements This work was partially supported by grants from PETROBRAS and CNPq. Clarisse de Souza is partially supported by grants from FAPERJ (E-26/102.400/2009) and CNPq (308964/2006-3).

References

Aldrich J, Chambers C, Notkin D (2002) Architectural reasoning in ArchJava. In: Magnusson B (ed) Proceedings of ECOOP'02, Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol 2374, pp 185–193

Arnold K (2005) Programmers are people, too. ACM Queue 3(5):54–59

Blackwell A, Green TRG (2003) Notational systems — the cognitive dimensions of notations framework. In: HCI Models, Theories, and Frameworks : Toward a Multidisciplinary Science (Morgan Kaufmann Series in Interactive Technologies), Morgan Kaufmann, chap 5, pp 103–133

Cacho N, Batista T, Garcia A, Sant'Anna C, Blair G (2006) Improving modularity of reflective middleware with aspect-oriented programming. In: SEM '06: Proceedings of the 6th international workshop on Software engineering and middleware, ACM, New York, NY, USA, pp 31–38, DOI http://doi.acm.org/10.1145/1210525.1210534

Cairns P, Cox AL (2008) Research Methods for Human-Computer Interaction. Cambridge University Press, New York, NY, USA

Cerqueira R, Cassino C, Ierusalimschy R (1999) Dynamic component gluing across different componentware systems. In: Tari Z, Meersman R, Soley R, Bukhres O (eds) Proceedings of DOA'99, IEEE Computer Society, Washington, USA, pp 362–373

Clarke S (2001) Evaluating a new programming language. In: Kadoda G (ed) 13th Annual Workshop of the Psychology of Programming Interest Group, pp 275–289

Clarke S (2004) Measuring API Usability. Dr Dobb's Journal pp S6–S9, URL `http://www.ddj.com/windows/184405654`

Clarke S, Becker C (2003) Using the Cognitive Dimensions Framework to evaluate the usability of a class library. In: Petre M, Budgen D (eds) 15th Annual Workshop of the Psychology of Programming Interest Group, pp 359–366

Costa P, Mottola L, Murphy AL, Picco GP (2007) Programming wireless sensor networks with the teenylime middleware. In: Cerqueira R, Campbell RH (eds) Middleware 2007, ACM/IFIP/USENIX 8th International Middleware Conference, Springer, Newport Beach, CA, USA, Lecture Notes in Computer Science, vol 4834, pp 429–449

Coulson G, Blair G, Clarke M, Parlavantzas N (2002) The Design of a Configurable and Reconfigurable Middleware Platform. Distributed Computing 15(2):109–126, DOI http://dx.doi.org/10.1007/s004460100064

Creswell JW (2009) Research Design: Qualitative, Quantitative, and Mixed Methods Approaches. Sage Publications

Denzin NK, Lincoln YS (eds) (2000) Handbook of Qualitative Research, 2nd edn. Sage Publications

Dittrich Y, John M, Singer J, Tessem B (2007) Editorial: For the special issue on qualitative software engineering research. Information and Software Technology 49(6):531–539, DOI http://dx.doi.org/10.1016/j.infsof.2007.02.009

Eden AH, Mens T (2006) Measuring software flexibility. Software, IEE Proceedings 153(3):113–125, DOI 10.1049/ip-sen:20050045

Edwards WK, Bellotti V, Dey AK, Newman MW (2003) Stuck in the middle: The challenges of user-centered design and evaluation for infrastructure. In: CHI 2003: Proceedings of the ACM Conference on Human Factors in Computing Systems, ACM, New York, NY, USA, pp 297–304, DOI http://doi.acm.org/10.1145/642611.642664

Eichberg M, Schäfer T, Mezini M (2005) Using annotations to check structural properties of classes. In: Cerioli M (ed) FASE, Springer, Lecture Notes in Computer Science, vol 3442, pp 237–252

Emmerich W, Aoyama M, Sventek J (2007) The impact of research on middleware technology. SIGOPS Operating Systems Review 41(1):89–112, DOI http://doi.acm.org/10.1145/1228291.1228310

Glaser B, Strauss A (1967) The discovery of grounded theory: Strategies of Qualitative Research. Wiedenfeld and Nicholdon, London, UK

Green TRG (1989) Cognitive dimensions of notations. In: Sutcliffe A, Macaulay L (eds) People and Computers V, Cambridge University Press, Cambridge, United Kingdom, pp 443–460

Green TRG, Petre M (1996) Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. Journal of Visual Languages and Computing 7:131–174

Henning M (2009) API design matters. Communications of ACM 52(5):46–56, DOI http://doi.acm.org/10.1145/1506409.1506424

Hirschi A (2007) Traveling light, the Lua way. IEEE Software 24(5):31–38

Ierusalimschy R (2006) Programming in Lua, second edition edn. Lua.org

Ierusalimschy R, Cerqueira R, Rodriguez N (1998) Using reflexivity to interface with CORBA. In: Proceedings of the IEEE International Conference on Computer Languages 1998, IEEE Press, Chicago, USA, pp 39–46

Ko AJ, Myers BA, Coblenz MJ, Aung HH (2006) An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on Software Engineering 32(12):971–987, DOI http://dx.doi.org/10.1109/TSE.2006.116

Kon F, Costa F, Blair G, Campbell RH (2002) The case for reflective middleware. Communications of ACM 45(6):33–38

Lazar J, Feng JH, Hochheiser H (2010) Research Methods in Human-Computer Interaction. Wiley Publishing, Chichester, UK

Maia R, Cerqueira R, Rodriguez N (2004) An infrastructure for development of dynamically adaptable distributed components. In: Meersman R, Tari Z (eds) Proceedings of DOA'04, OTM 2004, Springer-Verlag Heidelberg, Berlin, Germany, Lecture Notes in Computer Science, vol 3291, pp 1285–1302

Maia R, Cerqueira R, Kon F (2005) A middleware for experimentation on dynamic adaptation. In: Proceedings of ARM'05 workshop, ACM Press, New York, USA

Maia R, Cerqueira R, Calheiros R (2006) OiL: An object request broker in the Lua language. In: Proceedings of SBRC'06 - Salão de Ferramentas, pp 1439–1446

McLellan SG, Roesler AW, Tempest JT, Spinuzzi CI (1998) Building More Usable APIs. IEEE Software 15(3):78–86, DOI http://dx.doi.org/10.1109/52.676963

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)

Moura AL, Ururahy C, Cerqueira R, Rodriguez N (2002) Dynamic support for distributed auto-adaptive applications. In: Wagner R (ed) Proceedings of ICDCS 2002 Workshops, IEEE Computer Society, Washington, USA, pp 451–458

Nierstrasz O, Bergel A, Denker M, Ducasse S, Gaelli M, Wuyts R (2005) On the revival of dynamic languages. In: Gschwind T, Aßmann U, Nierstrasz O (eds) Proceedings of SC'2005, Springer-Verlag Heidelberg, Berlin, Germany, Lecture Notes in Computer Science, vol 3628, pp 1—13

OMG (2008) The Common Object Request Broker Architecture (CORBA) Specification - Version 3.1. Object Management Group, document: formal/2008-01-04

Ousterhout JK (1998) Scripting: Higher-level programming for the 21st century. Computer 31(3):23–30

Prechelt L (2003) Are scripting languages any good? A validation of Perl, Python, Rexx, and Tcl against C, C++, and Java. Advances in Computers 57:207–271

Puder A, Römer K, Pilhofer F (2006) Distributed Systems Architecture: A Middleware Approach. Elsevier, San Francisco, USA

Ramdhany R, Grace P, Coulson G, Hutchison D (2009) MANETKit: supporting the dynamic deployment and reconfiguration of ad-hoc routing protocols. In: Bacon J, Cooper BF (eds) Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, Springer-Verlag, New York, NY, USA, Lecture Notes in Computer Science, vol 5896, pp 1–20, DOI http://dx.doi.org/10.1007/978-3-642-10445-9_1

Ranganathan A, Campbell RH (2007) What is the complexity of a distributed computing system? Complexity 12(6):37–45, DOI 10.1002/cplx.20189

Robson C (2002) Real World Research: A Resource for Social Scientists and Practitioner-researchers, 2nd edn. Wiley-Blackwell Publishers, Oxford, UK

Rodriguez N, Ururahy C, Ierusalimschy R, Cerqueira R (1996) The use of interpreted languages for implementing parallel algorithms on distributed systems. In: Bougé L, Fraigniaud P, Mignotte A, Robert Y (eds) Proceedings of Euro-Par'96 Parallel Processing — Second International Euro-Par Conference, Springer-Verlag, Lyon, France, pp 597–600, Volume I, (LNCS 1123)

Rouvoy R, Merle P (2009) Leveraging component-based software engineering with Fraclet. Annales des Télécommunications 64(1-2):65–79, DOI http://dx.doi.org/10.1007/s12243-008-0072-z

Siek J, Taha W (2007) Gradual typing for objects. In: ECOOP '07: Proceedings of the 21st European conference on ECOOP 2007, Springer-Verlag, Berlin, Heidelberg, pp 2–27, DOI http://dx.doi.org/10.1007/978-3-540-73589-2_2

Vinoski S (2003a) It's just a mapping problem. IEEE Internet Computing 7(3):88–90

Vinoski S (2003b) The performance presumption. IEEE Internet Computing 7(2):88–90

Wollrath A, Riggs R, Waldo J (1996) A distributed object model for the java system. USENIX Computing Systems 9

ISSN: 1382-3256 (print version) ISSN: 1573-7616 (electronic version)