ISR Institute for Software Research
University of California, Irvine

An Analysis of Publish/Subscribe Middleware Versatility

Roberto S. Silva Filho
University of California, Irvine
rsilvafi@ics.uci.edu

David F. Redmiles
University of California, Irvine
redmiles@ics.uci.edu

August 2009

ISR Technical Report # UCI-ISR-09-3

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

An Analysis of Publish/Subscribe Middleware Versatility

Roberto S. Silva Filho and David F. Redmiles
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
{rsilvafi, redmiles}@ics.uci.edu

ISR Technical Report # UCI-ISR-09-3

August 2009

Versatility is an important quality that enables software to serve multiple purposes in a usable and useful way. As such, versatility is central to middleware in general and publish/subscribe infrastructures specifically. The development of versatile software, however, is difficult. It must achieve a favorable balance between different software qualities (or non-functional requirements) including: usability, reusability, flexibility, maintainability and performance, while supporting problem domain dependencies and constraints. Developers adopt different strategies in the design of versatile software including: modularization, stabilization, variation, generalization and specialization. By combining these strategies, different versatility approaches have been applied in the construction of infrastructures, for example: minimal core, one-size-fits-all, coordination languages and flexible infrastructures. Each one of have costs and benefits.

In this work, we motivate the need for versatility in the publish/subscribe domain, discuss its challenges, propose our own solution to the problem: YANCEES, a flexible publish/subscribe infrastructure, and present the results of a multi-dimensional quantitative and qualitative empirical study where we compare YANCEES with existing versatility approaches in the publish/subscribe domain. We summarize the results in the form of guiding principles, which goal is to better support application developers in choosing the best design approaches in the development of middleware; and infrastructure consumers in selecting the most appropriate infrastructure to support their applications.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Page

ACKNOWLEDGEMENTS

First and foremost, I want to thank God, revealed to us in Jesus Christ, the real author and consummator of all things, for His love and guidance, which make my life meaningful and my work possible.

I thank my advisor, Dr. David F. Redmiles, for his guidance, patience, support, friendship, for having always believed in my work and for providing insightful feedback during all the stages of the research described (and not) in this dissertation. David's encouragement made this research happen.

Cristina Lopes and André van der Hoek, both of whom served on my dissertation and candidacy committees, provided invaluable insights on this work. Their keen observations and questions have also guided me in this work.

I want to thank all current and previous members of the research group for innumerous discussions on this research and for the encouraging and supportive environment including Ban Al-Ani, Anita Sarma, Ben Pillet, Jie Ren, Erik Trainer, and Stephen Quirk, Norman Su, Steve Abrams, Patrick Shih. I owe thanks to many other professors and students in the School of Information and Computer Sciences for many invaluable discussions that contributed to this research.

Other researchers with whom I had a chance to interact along these years also deserve thanks, especially Werner Geyer for being my mentor during a wonderful summer at IBM, Cambridge. Others who deserve mentioning include Li-Te Cheng, David Millen, and John Patterson in the Collaborative User Experience Group at IBM Research.

Special thanks to Cleidson de Souza, Leila Naslavsky, and Márcio Dias, Marlon Vieira, André Nácul, Mirella Moro, Leonardo Murta, Rogério de Paula, Isabella Almeida, and Marcelo Alvim that were my colleagues at UCI, and provided invaluable insights in several occasions including practice talks, presentations, papers, and many others.

I also owe thanks to several other students in the School of Information and Computer Science for many invaluable discussions that contributed directly or indirectly to this research. I cannot list all students, so I will limit myself to just a few names: Arvind Krishna, Sundi, Adrita Bohr, Michael Kantor, and many other colleagues in the first, and difficult, years of graduate school.

I also want to thank all the brothers and Sisters from the Church in Irvine, in particular Wayne Kusumo, James Quiroga, Scott Young, Andrew Cho, Amy and Lazarus Sun, Song and Claire Chou, and Rob Egelink who nourished and cherished me during all these years.

Last, but not least, I cannot thank enough Grace, my wife, for her love, support, encouragement, understanding and patience during these years, and to my son Daniel. I also owe many thanks to my parents, Ana and Roberto for their encouragement and for always being close to me, even though living overseas.

I also acknowledge the financial support provided by the U.S. National Science Foundation under grant numbers 0534775, 0205724 and 0326105, an IBM Eclipse Technology Exchange Grant, and by the Intel Corporation.

Chapter 1. Introduction

Middleware provides a software layer between the application and the underlying network and operating systems, which goal is to relieve application software engineers from the burden of dealing with low-level distribution, communication and coordination concerns, such as network-level protocols, concurrency, transaction management, distributed object location, among others (Emmerich 2000). As such, middleware leverages on reuse (Barns and Bollinger 1991) by encapsulating network interaction and application domain expertise into APIs (Application Programming Interfaces) that facilitate the development of distributed applications.

According to a recent Gartner study (Biscotti, Jones et al. 2008), the world-wide market for middleware and application integration products has grown in 2007 to 14 billion US Dollars in annual license revenue. Figure 1 shows an overview of the market share that various middleware vendors had in 2006-2007 period. Note that these numbers are arguable higher since they do not include the use of open source software, an increasingly important class of middleware systems, which revenue comes mainly from support.

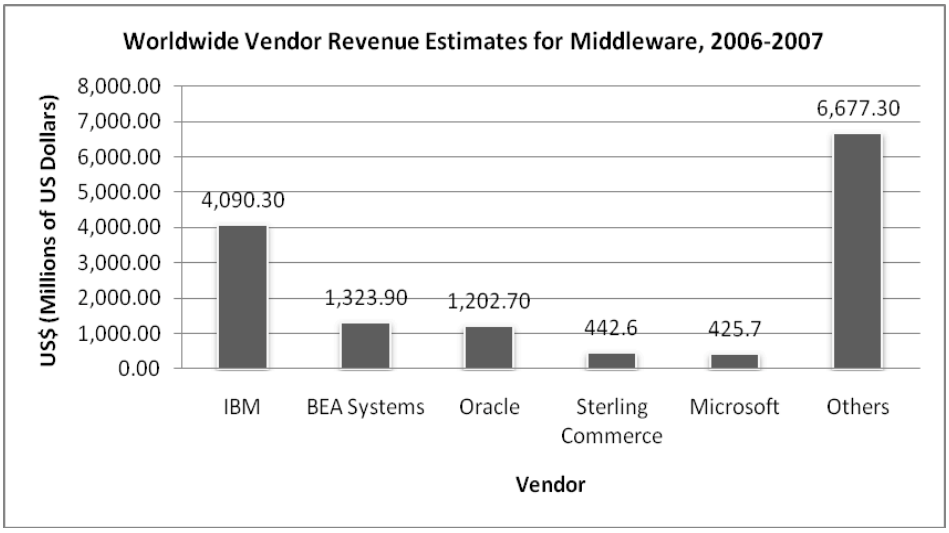


Figure 1 Worldwide vendor revenue estimates for total aim software, 2006-2007 (Millions of U.S. Dollars) source: Gartner Group 2008

The popularization of middleware has also originated different application-specific implementations, that not only frees developers from dealing with general low-level networking concerns, but also provides domain-specific support for different types of applications such as: real-time (Gore, Pyarali et al. 2004), mobility (Cugola, Nitto et al. 2001; Murphy, Picco et al. 2006) and context-aware applications (Boyer and Griswold 2004), to cite a few.

Publish/subscribe (or pub/sub) infrastructures are an important class of middleware that support the development of event-driven applications (Baldoni, Contenti et al. 2003). They are used as the basic communication and integration infrastructure on an increasing number of application domains such as: usability monitoring (Hilbert and Redmiles 1998), groupware (DePaula, Ding et al. 2005), awareness (Kantor and Redmiles 2001), residual testing (Naslavsky, Silva Filho et al.

2004), contextual collaboration (Geyer, Silva Filho et al. 2008) and many others (Gore, Pyarali et al. 2004) (Cugola, Nitto et al. 2001) (Boyer and Griswold 2004). This wide range of applications has demanded an increasingly diverse set of features such as: advanced event processing (event sequence detection, abstraction, and summarization), novel federation policies (for example: peer-to-peer), mobility support (pull notification, roaming protocols, event persistence), etc.

Thus, in order to fulfill its purpose in supporting the development of different distributed applications, middleware must be versatile. We define Versatility, in general terms, as: *the ability of software to serve different proposes in a usable and useful way*.

The development of versatile software is non-trivial. It requires a proper balance between different software qualities (or non-functional requirements) including: *efficiency*, *flexibility*, *usability*, *reusability* and *maintainability*. These qualities that many times conflict with one another. Moreover, application domain core requirements and their inter-dependencies limit the variability, configurability and extensibility of software. Finally, the technology and techniques supported in the development of software, such as programming languages, compilers and environments, also pose restrictions to the development and evolution of software. As a consequence, the development of versatile software requires a considerable amount of skill and expertise (Jingyue, Conradi et al. 2009). Hence, the understanding and documentation of these difficulties, and the derivation of principles and guidelines that support the development of these infrastructures are important for both the development and reuse of versatile software (Bosch 2004).

This brings us to our second definition of versatility, now in more specific software engineering terms. A versatile infrastructure is one that: *achieves a favorable balance between: reusability, usability, performance, flexibility and maintainability within the constraints imposed by an application domain*.

In the publish/subscribe infrastructures domain, different approaches have been developed and applied in of support versatility. In particular, in our survey of existing publish/subscribe infrastructures (Silva Filho and Redmiles 2005) we identified four major versatility approaches employed in the construction of existing industrial and research systems. They fall into a versatility spectrum that ranges from monolithic minimal core infrastructures such as Siena (Carzaniga, Rosenblum et al. 2001), to coordination languages such as JavaSpaces (Freeman, Hupfer et al. 1999), to software infrastructure that support variation in fixed points such as CORBA-NS (OMG 2004), to flexible (configurable and extensible) compositional infrastructures, such as YANCEES (Silva Filho and Redmiles 2005).

As a result, when selecting an infrastructure for their needs, application developers must choose among these existing strategies; many times, without fully understanding the consequences of their choices to important software qualities. Instead, middleware users can be mislead by common misconceptions, such as: *"more features or more flexibility are always better"* (Schwartz 2004), or *"keep it simple and general"* mottos (Raymond 2004). They are also not well aware of the trade-offs inherent to these versatility approaches. For example, the capability-usability trade-offs defined by problem domain dependencies, the complexity and lower performance of one-size-fits-all solutions, and the inflexibility and potential semantic mismatches of minimal-core infrastructures and coordination languages.

1.1 Approach

Whereas existing work discusses the benefits of each individual versatility approach (Harrison, Levine et al. 1997; Freeman, Hupfer et al. 1999; Carzaniga, Rosenblum et al. 2000; Silva Filho and Redmiles 2005), empirical studies that investigate and compare these approaches, considering both their benefits and limitations are rare in the literature (Glass 1994). In particular,

to the best of our knowledge, no comparative analysis of versatility trade-offs, involving these different approaches exist. In this paper, we present a multi-dimensional quantitative and qualitative study of the benefits and costs of different publish/subscribe infrastructures versatility approaches. For such, we analyze individual open source publish/subscribe infrastructures, built according to these approaches. Our analysis is multi-dimensional, it measures different software qualities in terms of code-level software attributes including: source code length, complexity, API size, change impact, modularity, separation of concerns, and execution delays. These measures are collected in the context of three heterogeneous and realistic case studies.

By measuring, analyzing and documenting the versatility design trade-offs of these infrastructures, our goal is to better support software infrastructure developers in applying the most adequate versatility strategy for their requirements, and infrastructure users in selecting the most appropriate strategy for the development of their applications. In other words, in this paper, we provide answers for the following questions:

- **RQ1**: Why building versatile infrastructures is so difficult? What factors impact their development and reuse? And what can be done to address these issues?

  This research question can be expressed in more specific research questions as follows:

- **RQ2:** From application developers' perspective, what major versatility approaches are available, and what's their costs and benefits with respect to API usability, reusability and performance?
- **RQ3:** From infrastructure developers' perspective, what factors should be considered in the construction of versatile infrastructures, and how do they affect important software qualities such as maintainability and flexibility?
- **RQ4:** Can we identify trade-offs between these versatility approaches, and derive principles and guidelines to support developers in building better versatile software, and users in selecting the best versatility approach implementation to their needs?

These questions are answered by means of our own experience in the development of YANCEES, a flexible publish/subscribe infrastructure, and through an analysis of versatility trade-offs, where we quantitatively and qualitatively compare YANCEES with existing industrial and research infrastructures, by means of different case studies. We summarize the results of this work in the form of a list of trade-offs and guiding principles.

1.2 Summary of contributions

The contributions of this work crosscut different research areas as follows:

1.2.1 Contributions in software engineering in general

This works contributes to software engineering research in the following manner:

- We propose the concept of versatility, together with an analytical framework that describes major operators employed in the development and reuse of versatile software (discussed in section Chapter 2), describing their main benefits and costs;
- We also perform a non-exhaustive survey of major architectural approaches adopted in the development of versatile software in general, and pub/sub infrastructures specifically (discussed at (Silva Filho and Redmiles 2005) and in section 4.1), evaluating infrastructures developed according to these approaches in our case studies;

- In order to analyze different versatility approaches, we designed comprehensive evaluation framework to compare the versatility of heterogeneous software infrastructures (Chapter 4).
- In doing so, we designed and applied a metrics suite, which quantifies software qualities as: usability, reusability, performance, flexibility, and maintainability in terms of lower-level attributes (section 4.4). In particular, we propose a new metric called development effort, which is the product of the total lines of code and cyclomatic complexity. This metric is the basis for our measurement of usability and reusability.
- The collected data was analyzed for correlations between these different software qualities, thus identifying trade-offs (Chapter 7). In particular, we provide empirical data showing that flexibility is more a consequence of design for change rather than the mere application of good software practices.
- Based on our case study, we also contribute with a set of principles and guidelines for requirements analysis, development and reuse of versatile publish/subscribe infrastructures (Chapter 8).
- Finally, we show the impossibility of the construction of an ideally versatile publish/subscribe infrastructure, one that can have its characteristics evolved independently from each other, pointing out the role of dependencies in limiting variability (as discussed in Chapter 2).

1.2.2 Contributions in the software product line engineering

In the software product line research, we contribute with:

- A deeper understanding of the impact of dependencies in limiting software flexibility, and an analysis of different feature interference problems in YANCEES (as discussed at (Silva Filho and Redmiles 2007) and in Chapter 2 of this document)
- An analysis of the role of dependencies in limiting variability, and a notation to express dependencies (as discussed at (Silva Filho and Redmiles 2006) and in Chapter 2 of this document).
- A comparative study of the versatility trade-offs in publish/subscribe infrastructures which compares flexible software product line approach (as YANCEES) with more traditional alternatives as: coordination languages (JavaSpaces), one-size-fits-all (CORBA-NS), and minimal core (Siena), as discussed in Chapter 7.

1.2.3 Contributions to middleware research

With respect to middleware research, we contribute with:

- YANCEES, a flexible pub/sub infrastructure (Silva Filho, de Souza et al. 2003; Silva Filho and Redmiles 2005), and a set of design principles supporting its development, showing how to achieve a favorable balance between different versatility software qualities in this domain;
- The extended Rosenblum and Wolf (Rosenblum and Wolf 1997) design model for publish/subscribe infrastructures, showing the importance to support protocols (discussed at (Silva Filho, de Souza et al. 2003) and in section Chapter 3 of this document);
- We also contribute with a quantitative and qualitative study of publish/subscribe middleware, where we show the complexity of using, extending and reusing different infrastructures.

1.3 Structure of this Dissertation

This paper is organized in the following manner.

Chapter 2 – Software Versatility. This chapter discusses the main challenges involved in the development of versatile software, in particular the role of problem domain, configuration-specific dependencies. It also presents a theoretical framework, based on fundamental set of operators captures the main development and reuse approaches to address these issues. We present these operators discussing their costs and benefits.

Chapter 3 – Building a Versatile Publish/subscribe Infrastructure. This chapter describes the main characteristics and versatility requirements of publish/subscribe infrastructures. It also shows how YANCEES, our solution to the problem, addresses these requirements through the application of different versatility operators.

Chapter 4 – Case Study Design. This chapter describes the design of a case study, that compares YANCEES with existing versatility approaches. As such, it describes the study setting, with its selected publish/subscribe infrastructures, applications domains, and the metrics suite used to evaluate and compare the case studies.

Chapter 5 – Case Study Implementation. This chapter describes, in more detail, the different implementations resulting from the application of existing versatility approaches in support of three application domains used in our study.

Chapter 6 – Study Results. This chapter analyzes the versatility of representative publish/subscribe infrastructures, in terms of their maintainability, flexibility, reusability, usability, and performance, when supporting our case studies. We analyze the results in a qualitative and quantitative way.

Chapter 7 – Versatility Trade-offs. In this chapter, we analyze the correlations (or their lack thereof) between the different software qualities that we analyzed in Chapter 6. The results of this chapter are used to support the design principles and guidelines for Chapter 8.

Chapter 8 – Principles and Guidelines. Through the lessons learned in our case studies and experience with YANCEES, this chapter discusses a set of design principles and guidelines to be used in the development of versatile software in general and publish/subscribe infrastructures specifically.

Chapter 9 – Study Limitations. This section discusses the limitations of the results we obtained through our case studies.

Chapter 10 – Related work. This section discusses related research contributions in the areas of software engineering, software product lines and middleware.

Chapter 11 – Conclusions. We conclude by summarizing our contributions and discussing potential implications of this work.

Chapter 2. Software Versatility

Software development is an iterative process that searches for satisficing solutions: i.e. the best solution given the available options and problem constraints (Simon 1996). In this search, it relies on the application of verified design and implementation principles and heuristics, with the help of appropriate software tools, processes and measurements. This process is, therefore, complex, and needs to deal with issues such as: software essential difficulties (Brooks 1987), problem domain and configuration-specific dependencies (Silva Filho and Redmiles 2006), the complexity of implementation approaches (Svahnberg, Gurp et al. 2005), the trade-offs that exist between different software qualities, as well as different needs from infrastructure developers and users. Moreover, it is usually the case that no single solution to a software development problem exist. Instead, a set of possible solutions can be produced, each one having specific benefits, as well as costs. As a consequence, during the process of design and implementation of software, different decisions are many times made in an ad-hoc manner, relying on expertise few developers and software designers (Larman and Basili 2003). The result is a set of implicit assumptions and trade-offs between different software qualities that usually become hidden (or implicit) in the system architecture and implementation (Roeller, Lago et al. 2006). These assumptions and trade-offs may lead to problems such as architecture mismatch (Garlan, Allen et al. 1995), inadequate performance, and poor reusability, flexibility and maintainability.

In this section we express development process of versatile software in terms of basic versatility strategies (or operations) that, when applied together, produce software with different degrees of versatility. We also discuss the problems and difficulties the development of versatile software must overcome in the domain of publish/subscribe infrastructures.

2.1 Versatility strategies

The design of a versatile infrastructure can be described as an interactive process in which different strategies are successively, and alternatively, applied in the construction of infrastructures that can fulfill different purposes. They represent “things” that designers can do in order to produce satisficing versatile infrastructures. These strategies (or operations) are: *specialization*, *generalization*, *stabilization*, *variation*, and *modularization*.

Likewise, when reusing existing infrastructures, users (application developers) successively, and alternatively, apply a set of different strategies (or operations) such as: *selection*, *adaptation*, *extension*, *configuration*, *composition* and *source code evolution*. Figure 2 summarizes the process of developing and reusing versatile infrastructures.

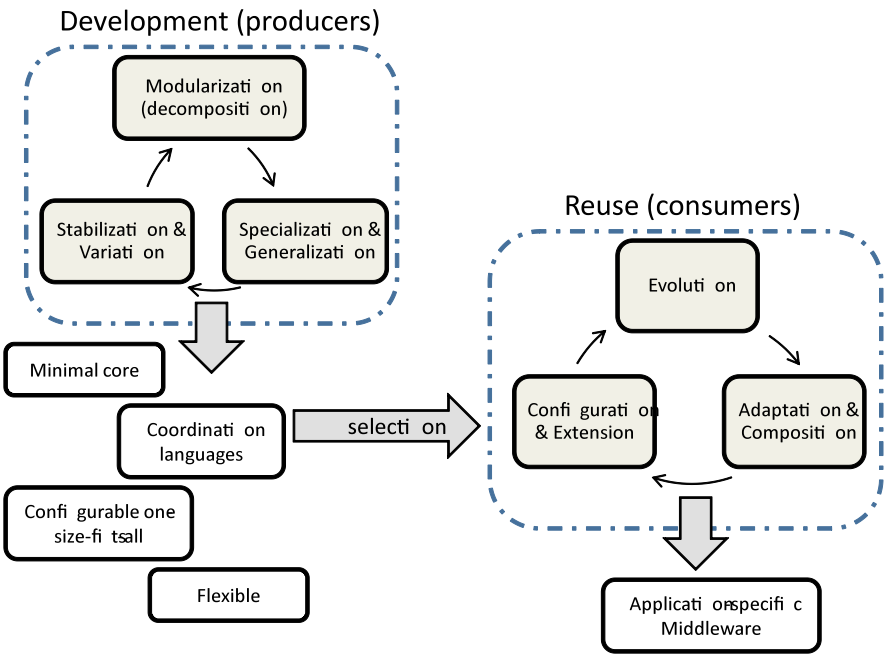


Figure 2 Development and reuse operators

As seen in Figure 2, different versatility approaches, such as minimal core infrastructure, coordination languages, one-size-fits-all and flexible approaches are a consequence of the successive application of the versatility operations. These versatility approaches fall into the spectrum of architectural patterns described in Figure 3, i.e. they originate infrastructures implemented as component-based systems, frameworks and a set of hybrid systems in between. Note that in Figure 3, grey areas imply fixed of code, whereas white areas represent variable or user-defined points in the software.

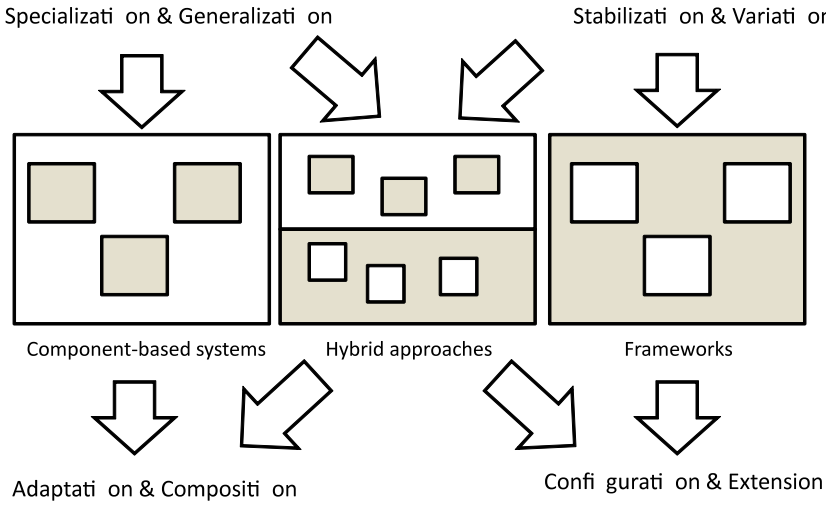


Figure 3 Correspondence between development and reusability strategies

Also note that the reuse operations of Figure 2, more or less, match the strategies adopted in the development of versatile software. Figure 3 shows the relation between development opera-

tors, the types of software infrastructures they produce, and the reuse operators utilized in the development of applications.

Modularization is the overall strategy that minimizes the impact of direct source code evolution, allowing the decomposition of software into different highly independent parts. *Configuration* and *extension* are usually applied in the reuse of software predominantly developed according to the *stabilization* and *variation* operators; whereas *adaptation* and *composition* are more suitable for the reuse of infrastructures predominantly implemented according to *specialization* and *generalization*. Hybrid approaches (shown at the center of Figure 3), come as a consequence of the joint application of different design operators. They are reused through a combination of *adaptation* & *composition* and *configuration* & *extension*. Example of hybrid approaches include software that can be extended through plug-ins (Birsan 2005), and component frameworks that provide a basic set of configurable services in support of the development of component-based software (Szyperski 2002).

The versatile software development and reuse strategies are discussed in more detail in the next sections.

2.1.1 Versatility development strategies

As shown in Figure 2, software engineers can apply the following operators in the development of versatile software.

2.1.1.1 Modularization

Modularization's goal is to decompose software into highly independent and interchangeable parts. This decomposition must be guided by the separation of concerns principle (Parnas 1972). As a result of modularization, different software parts can be replaced and evolved, with minimum impact to the software as a whole. Hence, modularity is key to software versatility as a way to simplify and add flexibility to a design (Parnas, Clements et al. 1984). As described by Baldwin and Clark (Baldwin and Clark 2000), modularization is achieved by the successive application of set of operations (*splitting, substituting, augmenting, excluding, inverting and porting*), as well as *reversion*, a new operation proposed by (Lopes and Bajracharya 2006), that captures the modularization achieved by "aspectizing" a piece of software.

The adequate modularization of software results in sets of stable public interfaces (or APIs) that work as implementation and reuse contracts. When combined with variation, modularization can support change in the form of alternative implementations or extensions to existing behavior. Examples of modularization techniques include decomposition based on objects, aspects (Kiczales, Lamping et al. 1997), features (Batory, Sarvela et al. 2003), and components (Szyperski 2002). For example, in Figure 5, each routing strategy is a module, implementing a standard interface. This allows the routing algorithm to change, while keeping a static interface.

The benefits of modularization, however, come at some costs. A side effect of modularization is the increase of the number of software parts that need to be integrated in the production of software. Moreover, this integration must obey configuration rules that are specific to the application domain. For example, in a publish/subscribe domain, different subscription, routing and event algorithms must be composed in a way that is coherent with the process of publication, routing and notification of events based on subscriptions. Hence, modularization not only results in more components, but also extra costs of composition and configuration management, as will be further discusses in this work.

2.1.1.2 Specialization

A piece of software is specialized if it perfectly fits the problem domain requirements. Specialization results in efficient and simpler implementations. In fact, it is a very common approach adopted in both research and industrial infrastructures. In our survey of versatility in publish/subscribe infrastructures (Silva Filho and Redmiles 2005), we identified many systems that were developed from scratch, providing novel specialized features. For example: CASSIUS (Kantor and Redmiles 2001), and JEDI (Cugola, Nitto et al. 2001). The application of specialization as the main development strategy have the benefits of performance simplicity. Moreover, when combined with other versatility approaches, for example variation, specialization can help in the design of software that balances different software qualities as performance, and reusability.

Specialization, however, has its cots. The excess of specialization usually results in infrastructures that are very low in versatility, i.e. that are difficult to evolve, configure, and reuse in different contexts. For example, in Figure 4, the specialized event abstractions in the left are perfect matches to the specific application requirements of IMPROMPTU (DePaula, Ding et al. 2005), a peer-to-peer file sharing tool; whereas the *GenericEvent* abstraction on the right-hand-side represents events in a generalized form: attribute/value pairs. This general representation can represent the specific events in the left, as well as novel event formats to come, at the cost of adaptation. The same cannot be said for the specialized events, which applicability (or scope) is restricted to IMPROMPTU application.

2.1.1.3 Generalization

A software is general if it can be used without change for different purposes (Parnas 1978). Generalization is a strategy which goal is to support a broad set of requirements through the use of generic data and control abstractions (for example, the *GenericEvent* in Figure 4). These abstractions provide a common vocabulary with which specialized features can be implemented. Generalization results in fixed design characteristics that fit different application requirements at the price of adaptation. As such, generalization avoids the ripple effect of changes in software, preventing unnecessary software evolution triggered by dependencies (Lehman and Parr 1976). Examples of generalization include virtual machines, and generalized data structures such as parameterized classes, templates (Czarnecki, Helsen et al. 2005) and the attribute/value event representation of Figure 4.

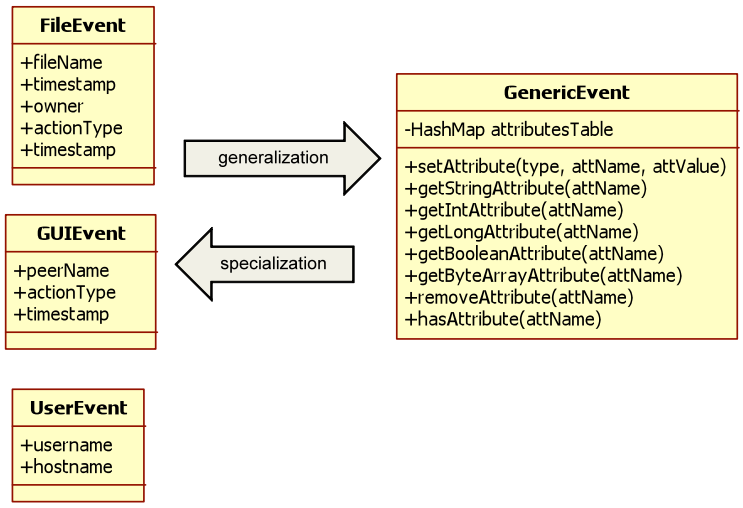


Figure 4 Specialization (left) versus Generalization (right)

The benefits of generalization, however, come at a cost. It requires the expression of existing control and data structures in terms of the generalized abstraction, which may lead to performance and software complexity penalties. Generalizations also have limitations in their expressiveness: many times they cannot represent all the variability in a domain. For example, attribute/value generalizations as the one shown in Figure 4 cannot represent objects in a programming language sense, a feature required, for example, by the mobile agents paradigm (Rus, Gray et al. 1997), or by the EDEM usability monitoring application (Hilbert and Redmiles 1998), that directly listens to GUI events form Java.

2.1.1.4 Variation

Variation allows software to support different application requirements by providing a pool of optional features. Variability allows the customization of software, based on the selection among existing optional features, installed in different variation points (van Gurp, Bosch et al. 2001) of the software. For example, by using conditional compilation (*#ifdefs* in C); by providing optional parameters in APIs; or by relying on design patterns such as Strategy (Gamma, Helm et al. 1995) and open implementation guidelines (Kiczales, Lamping et al. 1997), a system can be designed in a way that allow its users to select individual software characteristics to better match their needs. An illustration of variation is provided in Figure 5, where the Strategy design pattern is used to support different routing algorithms in a publish/subscribe infrastructure (i.e. topic-, content- and channel-based routing).

The costs of variation include the need for configuration management since some optional features may be incompatible with existing feature in other variation points. Usability costs of selecting among different approaches (Schwartz 2004) must also be considered when adopting this approach.

2.1.1.5 Stabilization

Stabilization represents the act of fixing certain parts of software, usually those that are less likely to change over time (Mahdy and Fayad 2002). That's the case, for example, with the separation between commonality and variability in software product lines (Coplien, Hoffman et al. 1998), or the separation between policy and mechanisms proposed by Wulf (Wulf, Cohen et al.

1974) where, certain core software mechanisms are fixed; while policies are variable. As seen by these two examples, stabilization is usually not applied alone, but it is combined with other strategies. An example of stabilization is the *PubSubFaçade* of Figure 5, which implements common basic publish/subscribe workflow, while supports variability in the router implementation strategy.

Stabilization has many benefits. It supports reuse by capturing the commonality in a domain, thus saving developers from the repetitive development of common software behavior. It can also be applied, together with generalization, to scope down software variability, thus simplifying the software design and reducing the costs of configuration and change impact management. For example, in the early design of YANCEES, features such as the event representation were variable. An analysis of dependencies in the publish/subscribe domain soon revealed the impact that changes in the event format would have in other parts of the system such as routing algorithms and subscription language. In order to prevent this problem, we opted to stabilize and generalize the event format, by representing events as attribute/value pairs, similar to that as shown in Figure 4.

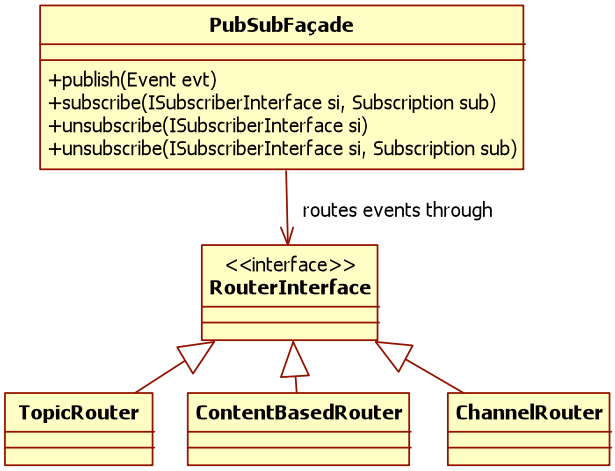


Figure 5 Stabilization and Variation in support of different routing strategies

When designing versatile software, stabilization must be applied to parts of software that are less likely to change (Mahdy and Fayad 2002), whereas variation should be used in support of variability in the domain. The stabilization of parts of software that are likely to change may excessively constraint its versatility. For example, in many publish/subscribe infrastructures, the subscription language and the routing mechanisms are stable. They define a common filtering vocabulary that cannot be easily evolved or configured. Consequences of excessive stabilization include mismatches between the provided features of the infrastructure and the required problem domain capabilities.

The combined application of these operators, i.e.: *modularization, specialization, generalization, stabilization* and *variation* results in implementations with different characteristics and degrees of versatility (as illustrated in Figure 3 and further discussed in section 4.1). These infrastructures must be reused according to different strategies discussed as follows.

2.1.2 Reuse strategies

When reusing versatile infrastructures, built according to different versatility strategies, application developers must apply a set of different operators as: *selection*, *extension*, *configuration*, *adaptation* and *composition (or integration)* (Krueger 1992). In this section, we discuss these operations in more detail.

2.1.2.1 Selection

It is usually the case that not one, but many solutions to a given problem exist. This is also true for publish/subscribe infrastructures. Hence, developers must first select the most appropriate infrastructure for their needs before extending, configuring, adapting or integrating it in support of their application domains. Selection must be supported by the proper understanding of different software infrastructure qualities, which vary according to the development strategies adopted in the software. Selection may also be impacted by organizational rules and constraints. For example, the whole organization may adhere to a standardized implementation, such as OMG CORBA-NS or Sun JMS, in spite of its lack of flexibility or usability.

2.1.2.2 Extension

Extension allows software to accommodate novel requirements, not initially foreseen in its design by supporting new features. The extensibility of an infrastructure must be planned beforehand, being supported as part of software architecture (Eden and Mens 2006). Examples of architecture-based extensibility strategies include the implementation of plug-ins (Birsan 2005), and specific components for existing component frameworks (Szyperski 2002). APIs can also be extended with new commands through the use of wrappers and façades (Gamma, Helm et al. 1995). Extension is also supported by lower-level programming language mechanisms such as the "extends" clause in Java. Aspect-oriented programming (Kiczales, Lamping et al. 1997) can also be used as an extension mechanism.

In spite of existing mechanisms to support extensibility, as listed above, the process of extension can be costly and time consuming. It requires the learning of the system API, either through documentation or existing examples, and the understanding of underlying system assumptions and dependencies (Bosch, Florijn et al. 2002).

2.1.2.3 Configuration

Configuration is the process of tuning a piece of software to a certain purpose by selecting sub-sets of features among a set of options it supports. Configuration presumes a design that supports variability and/or extensibility, and mechanisms that allow the easy selection of these options. Examples of configuration implementation strategies include: optional parameters in APIs, configuration files, configuration interfaces in open implementation (Kiczales, Lamping et al. 1997) or mechanisms such as Builder and Factory design patterns (Gamma, Helm et al. 1995). In a framework, configuration may involve the selection among different classes, components or aspects in the construction of a customized infrastructures.

The examples of Figure 16, Figure 17, Figure 19, and Figure 20 show the application of the extensibility and configurability strategies in the implementation of YANCEES.

Configurability also has costs. In particular, the need for selection mechanisms can introduce extra complexity in the software. For example, it is common to use optional methods and parameters in APIs. These extra methods may raise extra exceptions, or may require default parameters (or "don't care" values) when the options are not necessary. As a consequence, the process of

configuration may result in higher software complexity and length, which may lead to errors or performance penalties in the program. In order to address this problem, some guidelines such as separation of control and regular use interfaces must be adopted (Kiczales, Lamping et al. 1997).

2.1.2.4 Adaptation

Adaption allows software to be reused, without change, in slightly different situations. In other words, it supports the building of new functionality around existing sets of features. Adaptation also implies data and control transformations in order to fit or express existing required features in terms of to slightly different provided functionality.

Adaptation can be used in combination with generalization in support of application domain variability, allowing users to express specialized features in terms of provided generalizations. Examples of adaptation approaches include the Adapter design pattern (Gamma, Helm et al. 1995), wrappers and composition filters (Bergmans and Aksit 2001). For example, Figure 6 illustrates the case where an ideal publish/subscribe interface, required by a software monitoring tool, is expressed in terms of an existing publish/subscribe infrastructure. Extra feature, not supported by the infrastructure, as event persistency, is provided by a tuple space component. These components (the router and the tuple space) are integrated through adaptation and composition.

The process of adaptation can be costly both in terms of performance and development effort. In particular, adaptation costs are high when there are semantic mismatches between the application requirements and the provided features, requiring the implementation of extra functionality. For example, the implementation of pull notification on push notification servers requires the development of extra polling protocols and the temporary storage of events for later retrieval. As a consequence, in the worst case scenarios, adaptation costs can become prohibitive.

2.1.2.5 Composition (or integration)

The decomposition of software into modules (or modularization), presumes the recombination of these modules into the construction of useful pieces of software (Parnas, Clements et al. 1984). Composition, therefore, supports reuse by allowing the construction of application-specific software out of existing parts. Composition usually requires some degree of adaptation. Examples of composition mechanisms supported by programming languages include the weaving mechanisms supported by mixings (Cardone, Brown et al. 2002) and aspect-oriented languages (Kiczales, Lamping et al. 1997); provided and required interfaces used in component frameworks (Szyperski 2002); associations, aggregations and composition relations in object-oriented programming; and the use of simple method calls in structured programming languages. In the example of Figure 6, a complex interface is implemented by combining (or composing) the features of a tuple component and a publish/subscribe core and different adapters.

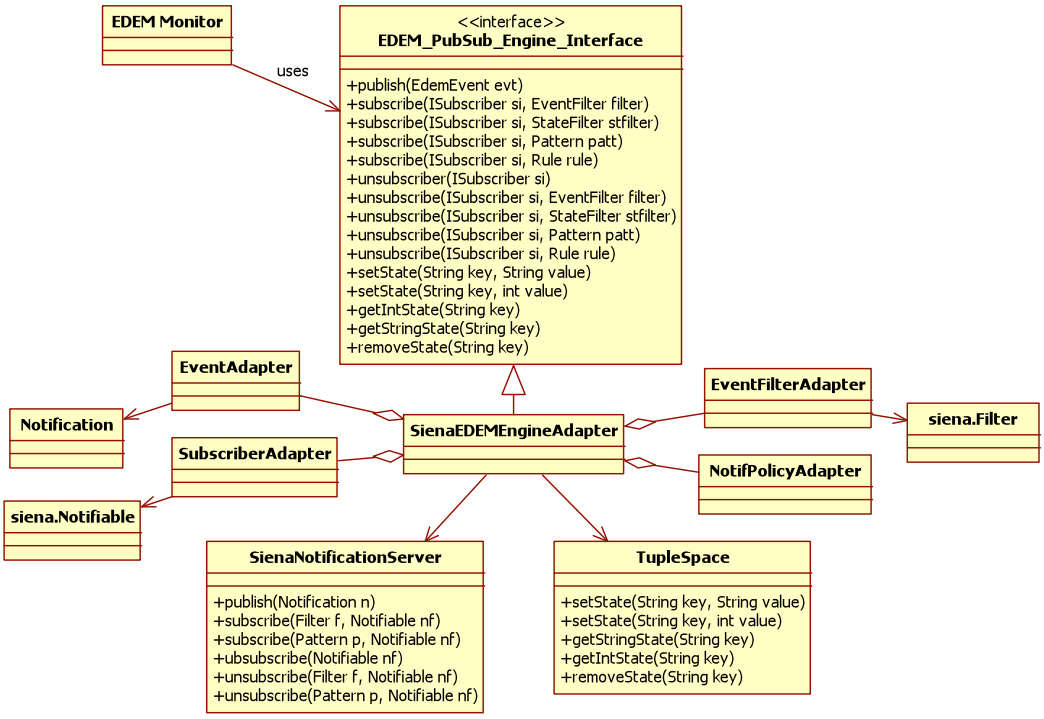


Figure 6 Expressing EDEM required functionality in terms of existing infrastructures through Adaptation and Composition

Composition mechanisms have their own costs. For example, the separation between base code and aspect code may lead to problems of over- and under- matching of aspects when base or aspect codes are evolved (Ruengmee, Silva Filho et al. 2008), which may lead to errors and costly development cycles. In the publish/subscribe domain, one-size-fits-all approaches as CORBA-NS require users (application developers) to define different configurations by composing existing proxy components, which increases the API size and the complexity of activities as posting a subscription. Dynamic architectures as YANCEES publish/subscribe infrastructure (discussed in section Chapter 3) use runtime parsers to compose subscriptions, on demand, according to the composition rules of valid subscriptions. This process increases the core infrastructure complexity, which needs to combine plug-ins into event processing hierarchies after checking for possible inconsistencies in the provided subscriptions.

2.1.2.6 Code Evolution

If the source code of software is available, developers can customize the software to their needs by direct code evolution. However, due to the high costs of software comprehension, evolution, and management of forked branches, changes in the source code are usually avoided (Parnas 1994) (Lehman and Parr 1976) (Lehman, Ramil et al. 1997). In fact, the different versatility strategies we just discussed have the goal of minimizing the need for direct changes in software code. Code evolution is therefore a least resource option that must be used only when the versatility strategies we just mentioned are not applicable.

As previously mentioned, both versatility and reuse strategies are usually not applied in isolation. Instead, infrastructure producers must apply these strategies in different forms and degrees in the production and reuse of versatile software. Moreover, these strategies not only have benefits, but also costs. For example: The decomposition of software into modules requires extra costs

of composition and coordination (De Souza 2005). Specialized features, while efficient, cannot be easily ported to different application domains. The use of general data structures requires the extra costs of adaptation, and can result in inefficient implementations. Whereas the use of variation and extension requires additional care for configuration management. These costs and benefits, therefore, must be managed in order to achieve a favorable balance between versatility and other important software qualities. In the next section, we discuss, in more detail, the main factors that hinder the development of versatile software.

2.2 Versatility challenges

As made evident in the previous section, software versatility is easy to idealize but difficult to achieve. Contrary to common wisdom, versatility is not a mere consequence of the application of good software engineering techniques. Instead, it depends on a complex set of factors, requiring developers to overcome a set of difficulties. For example, the development of versatile software must respect a set of factors such as: the fundamental characteristic of the **application domain** (Silva Filho and Redmiles 2006), the selected **architecture style** constraints (Sangwan, Li-Ping et al. 2008), the specific configuration, adaptation, composition and extension rules from the adopted **implementation strategies** (Mens and Eden 2005), and the conflicting needs of different **stakeholders**.

In this section, we discuss the fundamental software problems that make the development of versatile software difficult. These factors come from our experience in the development of YANCEES, a flexible publish/subscribe infrastructure, and from our survey of the literature (Silva Filho and Redmiles 2005). These problems are: software quality trade-offs, fundamental and configuration-specific dependencies, and technological constraints.

2.2.1 Software quality trade-offs

At the source code level, software qualities such as *flexibility*, *maintainability*, *reusability*, and *usability* are consequence of a common set of software attributes such as: code size, modularity and complexity (IEEE 1993; Bandi, Vaishnavi et al. 2003). Due to the inter-dependency between these factors, these software qualities are not orthogonal.

At the organizational level, different stakeholders are interested in distinct software qualities. For example, as illustrated in Figure 7, infrastructure producers are concerned with building software that is easy to maintain, extend and configure, thus, minimizing the work of evolving the infrastructure to support shifting requirements in a domain. Infrastructure consumers, on the other hand, are more concerned with the usability, efficiency and reliability of software.

The different stakeholders' needs, together with the inter-dependencies between different software attributes, define different trade-offs between important software qualities. For example, as shown in Figure 7, the development of flexible software requires the careful application of the design for change principle (Parnas 1978) through the use of modularization, separation of concerns and different variability implementation approaches (van Gurp, Bosch et al. 2001). These approaches can potentially increase code size and complexity, leading to a higher density of program defects (Eaddy, Zimmermann et al. 2008), thus reducing software reliability. Moreover, variability implementation approaches (for example, the factory design pattern) can negatively affect the infrastructure API usability (Ellis, Stylos et al. 2007), increasing the application developers development effort.

As a consequence, in order to balance both users' and developers' needs, the design of versatile software need to achieve a favorable balance between different and possibly conflicting software qualities. This balance is achieved through the judicious application of a combination of

versatility and reuse strategies discussed in 2.1, according to a set of principles, guidelines and ultimately, the designer's own expertise.

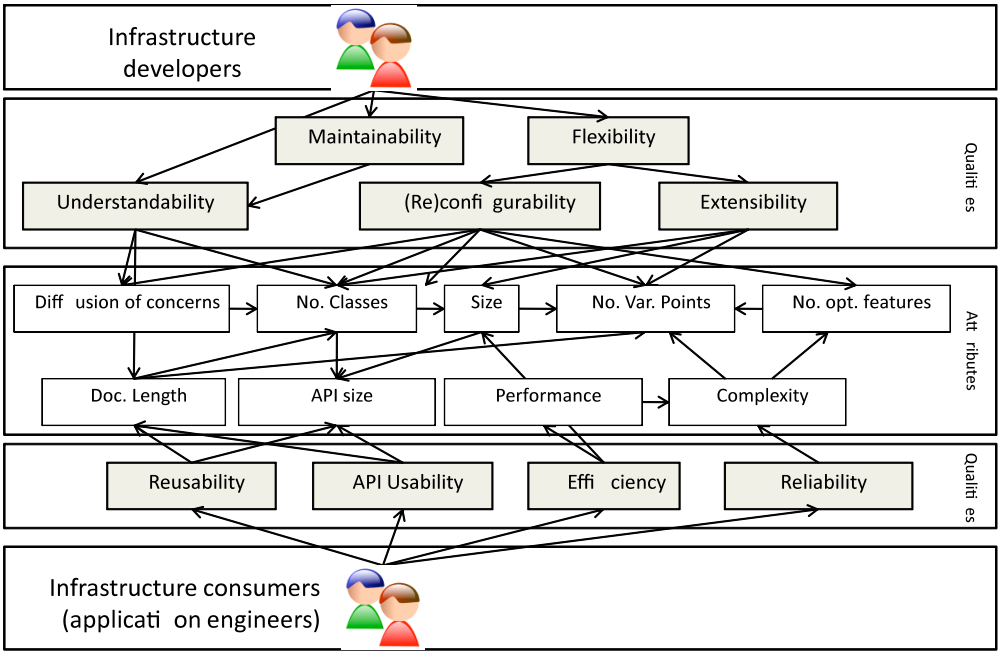


Figure 7 Different stakeholders requirements, quality dependencies and trade-offs

2.2.2 Fundamental domain dependencies

Fundamental problem domain dependencies integrate the main concerns of a problem domain through control and data dependencies. These dependencies restrict variability.

As discussed in section 2.1.1, the development of versatile software, is many times supported by the analysis of commonality and variability, which allows the separation between essential and optional concerns in a domain (Coplien, Hoffman et al. 1998). This separation supports the development of versatile software through stabilization & variation, or through specialization & generalization. In particular, modular software units can represent both common and specialized behavior that are composed in the production of domain-specific infrastructures.

For example, in the publish/subscribe domain, the main concerns of a publish/subscribe infrastructure have their variability centered on the design dimensions discussed in section 3.3, and illustrated in Figure 8, i.e. *event*, *publication*, *routing*, *subscription*, *notification* and *protocol* variability dimensions.

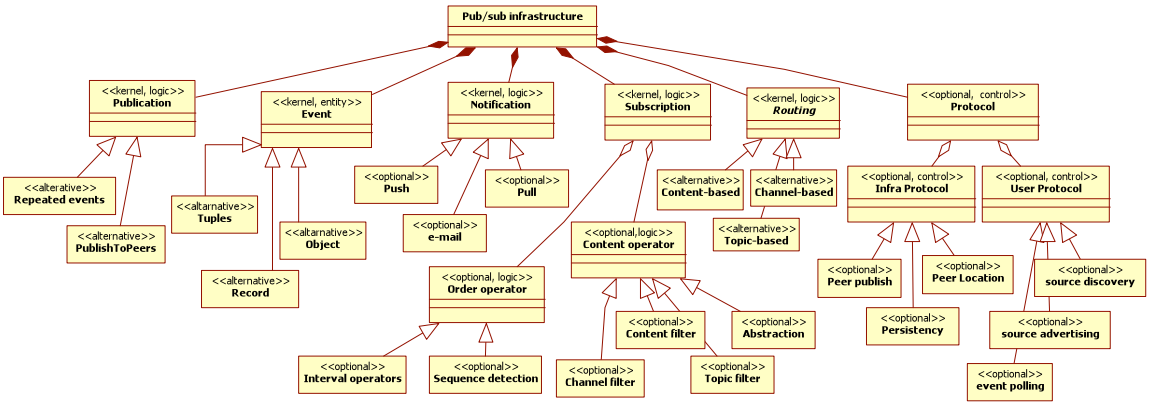


Figure 8 Publish/subscribe feature model showing design dimensions and their variability (source (Silva Filho and Redmiles 2006))

Note that the diagram Figure 8, of uses a UML notation. Stereotypes (inside << and >>) express optionally (OR relation) and exclusivity (XOR relation). An optional feature can be selected together with other optional features in the same level, for the same super feature. Abstract features appear as the first level under the pub/sub infrastructure concept, and are not marked with stereotypes. Aggregation indicates containment and composition implies a part-role relation of the pub/sub concept. When no stereotype is used, the features or concepts are mandatory.

These main publish/subscribe concerns sown in Figure 8 are not orthogonal. Instead, a closer analysis reveals a series of data and control dependencies that interconnect these fundamental concerns. These dependencies limit the set of valid combinations of features a publish/subscribe infrastructure must support at any given time. These are called **fundamental problem domain dependencies.** Figure 9 illustrates the problem domain dependencies for publish/subscribe infrastructures.

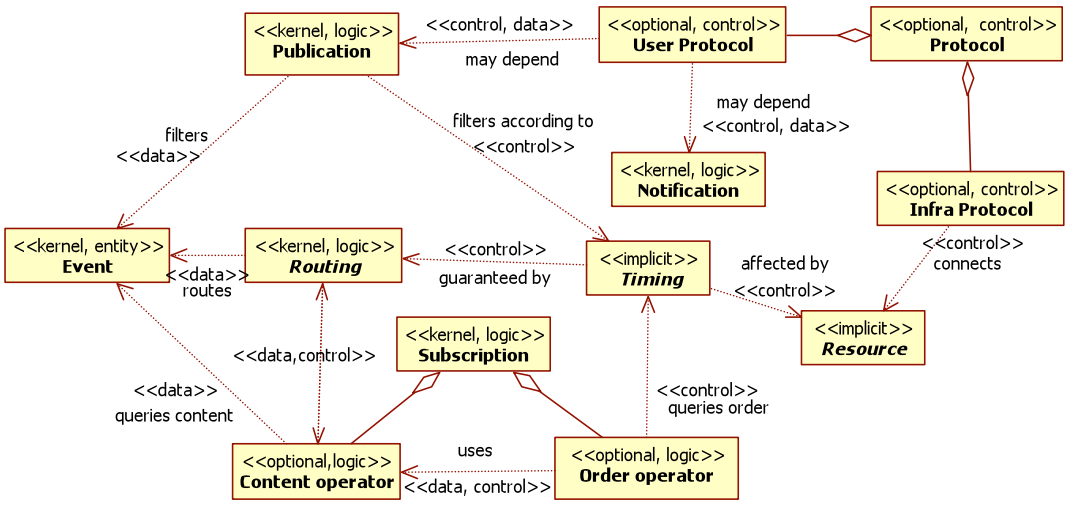


Figure 9 Publish/subscribe domain fundamental and derivative dependencies

Generally speaking, fundamental problem domain dependencies (represented as arrows in Figure 9) can either define data or control coupling between the different concerns of the system.

Control coupling usually limits the activation order of the different pieces of software, whereas data coupling can limit the variability and reuse of those components (Parnas 1978; Stevens, Myers et al. 1999). For example, as shown in Figure 9, the coupling that exists between event representation and different publish/subscribe concerns makes possible to changes in the event representation to impact concerns such as: routing, subscription, and publication; whereas other concerns, such as higher-level event processing operators (for example, event sequence detection) can more or less evolve independently from the event representation.

Note that, in the diagram of Figure 9, we also introduce new dimensions (written in italic) to represent emerging concerns as: *timing* and *resource*, that are consequence of control dependencies between different variation points. For example, the timing guarantees provided by the infrastructure are dependent on the resource model (the way the different components of the infrastructure are distributed) and the routing algorithms supported. Modifications in these parameters may affect the outcome of existing subscription operators, such as *pattern detection*, as well as *publication filters* that may combine or remove repeated events coming within a certain time interval.

In short, the fundamental problem domain dependencies define the main pillars of a problem domain. Changes in these concerns, especially those with high fan-in, will affect the problem domain in a fundamental way, preventing the independent evolution of these dependent concerns. As a consequence, designers need to adopt different strategies in the balance of versatility with the constraints posed by these dependencies, for example: generalization.

2.2.3 Configuration-specific dependencies

The variability of features in the domain also defines **configuration-specific dependencies** between compatible features that must be installed, together, in the production of valid software configurations. They also define implicit incompatibilities with features that cannot co-exist.

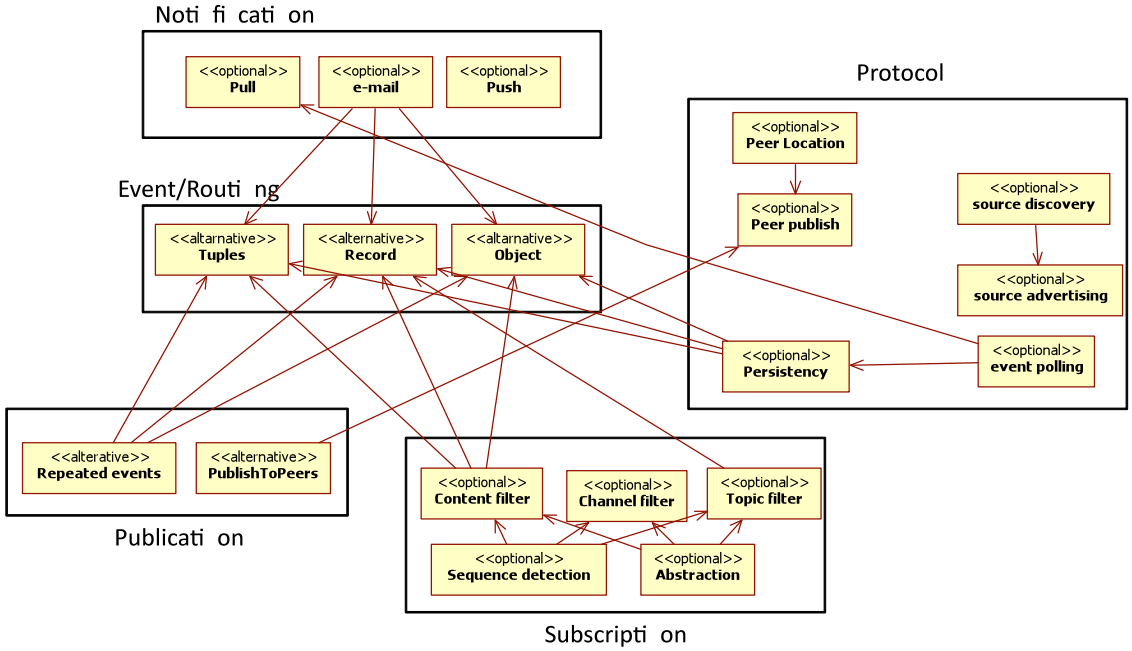


Figure 10 Configuration-specific dependencies between features

It is also the case that features in different design dimensions cannot exist in isolation; instead, they must co-exist in the context of valid configurations. For example, as shown in Figure 10, in the publish/subscribe domain, event persistency usually requires pull notification capability, allowing users to retrieve saved events at a later time, whereas content-based routing requires a combination of event representation (usually attribute/value pairs), and subscription commands that allow users to express content-based queries. Hence, configuration-specific dependencies must be supported and enforced in the construction of versatile infrastructures, in particular in versatility approaches that provide configurability and/or extensibility such as one-size-fits all and flexible approaches.

2.2.4 Technological constraints

Different implementation strategies can be applied in support of *generality*, *variability*, *configurability* and *extensibility* requirements of versatile software. Examples of such approaches include: design patterns, parameterized classes, aspects, mixings and others discussed at (Svahnberg, Gurp et al. 2005). These techniques, however, not only have versatility benefits, but also have implicit costs. They define **technological constraints** in the form of extension, configuration, adaptation and composition rules. For example, the use of software patterns such as *Strategy* (Gamma, Helm et al. 1995), require developers to interact with the selection capabilities provided by its interface. And the composition of systems into modules, either through the use of objects, aspects or more complex components, require their later integration in the construction of complex systems through the use of composition strategies and protocols (Parnas, Clements et al. 1984). In particular, the use of aspect-oriented programming, for example, requires the proper management of point cut descriptors, that must be consistent with the base code evolution. This management can result in high usability costs (Ruengmee, Silva Filho et al. 2008).

These dependencies may also affect other software qualities such as *complexity*, *usability*, and *understandability*. For example, when applied in combination, design patterns many times introduce indirections in the code that may hinder its legibility and extension ((Czarnecki and Eisenecker 2000) pp. 295). Moreover, the composition of design patterns have shown to increase the diffusion of concerns and complexity of software (Cacho, Sant'Anna et al. 2006).

Finally, the mechanisms of composition can lead to feature interaction (Bowen, Dworack et al. 1989). Feature interaction occurs when the combination of apparently unrelated features modify each other's behavior in an unforeseen way. For example, the use of patterns such as *Chain of Responsibility* (Gamma, Helm et al. 1995) as an extensibility mechanism , can lead to feature interaction if the proper order of components, that belong to different features, is not respected (Silva Filho and Redmiles 2007).

2.3 Summary

Factors such as software attributes and quality dependencies, fundamental, configuration-specific dependencies, and technological constraints can hinder the development of versatile software. Different versatility strategies can be applied in the construction of versatile software, subject to these constraints. The proper application of these operators, however, require the observation of different trade-offs. In many cases, infrastructure developers manage these trade-offs in ad-hoc ways, based on their own expertise; whereas infrastructure consumers (application developers) face the dilemma: to build new infrastructures from scratch or to reuse existing infrastructures, build according to undocumented characteristics and different assumptions (Garlan, Allen et al. 1995). Whereas some of these trade-offs are general to software engineering, others are specific of the application domain at hand.

Hence, an analysis of these trade-offs, and the derivation of guiding principles to achieve a favorable balance between conflicting software qualities becomes necessary. In this work, we analyze these trade-offs in the publish/subscribe domain by first discussing our experience in the development of YANCEES, a versatile publish/subscribe infrastructure, and then by comparing YANCEES with different versatility strategies with the help of a three case studies.

Chapter 3. Building a Versatile Publish/Subscribe Infrastructure

In this chapter, we describe the design and implementation of YANCEES, a versatile publish/subscribe infrastructure. We first discuss YANCEES' motivation, its versatility requirements, and the principles adopted in its design. We follow by discussing how YANCEES design addresses the problems induced by problem domain, configuration-specific and technological dependencies, through the application of the different development strategies, including those discussed in section 2.1.

3.1 YANCEES motivation

The development of YANCEES (Yet ANother Configurable and Extensible Event Service) (Silva Filho, de Souza et al. 2003; Silva Filho and Redmiles 2005) was first motivated by the need of a single infrastructure that could support the development of event-driven applications in different problem domains. In other words, the goal in the development of YANCEES was to leverage on reuse, configurability and extensibility in order to reduce the costs of building application-specific publish/subscribe infrastructures, thus preventing the development of different infrastructures, from scratch, each time a new event-driven application was developed. In particular, our target application domains included: software monitoring (Hilbert and Redmiles 1998), awareness (Kantor and Redmiles 2001), groupware (Silva Filho, Geyer et al. 2005), usable security (DePaula, Ding et al. 2005), as well as new event-driven software engineering applications to come.

By the time we started the development of YANCEES, on the school year of 2002-2003, we surveyed both industrial and research infrastructures for a single publish/subscribe infrastructure that could support the heterogeneous requirements of these application domains (Silva Filho and Redmiles 2005). To the best of our knowledge, no single approach existed that could be easily extended and configured to our needs. Instead, existing infrastructures were developed according to different strategies, having different degrees of versatility. These were:

- Build for a single use, to support individual application domain needs, as JEDI (Cugola, Nitto et al. 2001), and CASSIUS (Kantor and Redmiles 2001);
- Generalized minimal core systems as Siena (Carzaniga, Rosenblum et al. 2001), Scribe (Castro, Druschel et al. 2002), JMS (Sun Microsystems 2003) or Elvin (Fitzpatrick, Mansfield et al. 1999);
- Coordination languages as Linda (Gelernter 1985), IBM TSpaces (Wyckoff 1998) or SUN JavaSpaces (Freeman, Hupfer et al. 1999);
- One-size-fits-all monolithic servers as CORBA-NS (OMG 2004) and READY (Gruber, Krishnamurthy et al. 1999).

Even though these approaches are many times able to support the development of event-driven software, they have fundamental limitations: none of them supports the exact set of application-specific features required by our target application domains; and they do not provide any extensibility mechanism other than the direct modification of their source code. More specifically, both 'built for single use' and 'minimal core' infrastructures are not designed for change, being costly to evolve; one-size-fits-all approaches while support configurability, are not extensible; whereas coordination languages do not provide the necessary expressiveness to support, for example, protocols and advanced event processing required by these application domains.

In fact, as a consequence of these deficiencies, and despite the existence of standardized solutions such as CORBA-NS and generalized approaches such as Siena and JavaSpaces, new publish/subscribe infrastructures continued to be built every time novel sets of features are required. This is made evident by the large number of application-specific infrastructures discussed at (Silva Filho and Redmiles 2005).

In order to address these limitations, we designed and implemented YANCEES. YANCEES main goal was to combine the simplicity and efficiency of minimal core approaches, with the configurability of one-size-fits-all approaches in the construction of a flexible (extensible and configurable) infrastructure that can be easily customized to support the needs of different application domains.

Before discussing the implementation of YANCEES, in the next sections we first introduce the publish/subscribe communication style and its requirements; in section 3.3 we present a design framework that captures the main commonality and variability of the publish/subscribe domain; and in section 3.4 we discuss the versatility requirements of publish/subscribe infrastructures.

3.2 Publish/subscribe communication style characteristics

Events represent temporal facts in the world or state transitions in computational systems. Event-driven applications are those that operate in response to events. They are usually built according to the publish/subscribe communication style, a distributed version of the *Observer* design pattern (Gamma, Helm et al. 1995). Publish/subscribe infrastructures implement this style in support of event-driven applications development. The publish/subscribe infrastructures (or services) must support different requirements that make their design particularly challenging. These are:

Interactivity. Publishers and subscribers interact with the service by publishing events and submitting and removing subscriptions at different rates, times and formats.

Expressiveness. Subscriptions are usually expressed in the form of query languages (textual expressions in the content or order of evens), or by a combination of programming-level objects that represent commands and filters in the language. The expressiveness of the subscription language must match the routing and filtering capability of the infrastructure. In other words, there must be a correspondence between language and infrastructure functionality.

Dynamism. A publish/subscribe infrastructure must support the runtime arrival and departure of publishers and subscribers of events. Each subscriber provides an event processing and filtering expression that exercises different capabilities of the infrastructure. Subscriptions are posted and removed dynamically, requiring the routing mechanism to adapt to these changes, thus servicing multiple publishers and subscribers at the same time.

Data and control coupling. Publish/subscribe infrastructures operate over a common data flow of information (events) from publishers to subscribers. For such, they define strong data and control coupling between the different phases of the publish/subscribe process. This makes the process of publication, routing and notification of events based on subscription to be strongly dependent on the event format, and timing constrains, creating a sequential dependency among these steps, as previously illustrated in Figure 9. This coupling many times leads to designs based on dataflow-oriented decomposition, that go against the separation of concerns principle (Parnas 1972). It also makes changes in different parts of software affect other parts of the infrastructure leading to feature interference as discussed in section 2.2.

For such characteristics, pub/sub infrastructures represent an application domain where different factors must coexist, and where the different versatility strategies discussed in section 2.1 can be put to test in their fullness.

3.3 Publish/subscribe infrastructures commonality and variability

A versatile software must be able to support not only the common characteristics of a domain but also its variability (Coplien, Hoffman et al. 1998). In this section, we analyze the diversity of features that the publish/subscribe infrastructures must support, and the dimensions on which these features exist.

All publish/subscribe infrastructures share the common process of: *publication*, *routing* and *notification* of *events* based on *subscriptions*. This process, however, can be supported in ways that are specific to each application. In particular, the publish/subscribe domain variability can be modeled along the design dimensions discussed in Table I.

Table I Publish/subscribe domain variability

MODEL	DESCRIPTION	EXAMPLE
Event model	Specifies how events are represented	Tuple-based; Object-based; Record-based, others.
Publication model	Permits the interception and filtering of events as soon as they are published, supporting the implementation of different features and global infrastructure policies.	Elimination of repeated events, persistency, publication to peers (through protocol plug-ins).
Routing model	Defines the mechanism that matches events to subscriptions, resulting in the delivery of events to the appropriate subscribers.	Topic-based, Content-based, channel-based
Subscription model	Allow end-users to express their interest on sub-sets of events and the way they are combined and processed.	Filtering: content-based, topic-based, channel-based; Advanced event correlation capabilities
Notification model	Specifies how subscribers are notified when subscriptions match published events.	Push; pull; both, others

Protocol model	Deals with other necessary infrastructure interactions other than publish/subscribe. They are subdivided in **interaction protocols** (that mediate end-user interaction), and **infrastructure protocols** (that mediate the communication between infrastructure components)	**Interaction protocols**: Mobility; Security; Authentication; Advanced notification policies. **Infrastructure protocols**: federation, replication, Peer-to-peer integration.

These variability dimensions represent an extended version of the Rosenblum and Wolf's publish/subscribe dimensions (Rosenblum and Wolf 1997). In particular, we extended this model to include an extra protocol dimension, that captures the different kinds of interaction supported by the infrastructure other than the publication and notification of events. Note that the variants in each variability dimension, shown in Table I, came from a variety of application domains such as: awareness (CASSIUS (Kantor and Redmiles 2001)), groupware (IMPROMPTU (DePaula, Ding et al. 2005)), and software usability monitoring (EDEM (Hilbert and Redmiles 1998)).

In the design of YANCEES, we use the dimensions of Table I as the basic variation points, whereas the features in each dimension are mapped into components that extend these dimensions.

3.4 Versatility requirements

Besides supporting the essential middleware requirements of *performance*, *scalability*, *interoperability*, *heterogeneity*, network *communication* and *coordination* (Emmerich 2000), versatile infrastructures must: first support application-specific features discussed in section 3.2; and second, achieve a favorable balance between *maintainability, flexibility, usability, reusability* and *performance*. In this section, we further describe requirements.

3.4.1 API usability

From the point of view of application developers, middleware provides an API (Application Programming Interface) that supports the construction of distributed applications. A usable API is one that is easy to understand and operate, and which abstraction matches the users' application needs and usage scenarios. According to (Henning 2009), an API should be efficient, minimal, designed according to the perspective of the users. It should also hide unnecessary implementation details and be well documented.

Hence, in the publish/subscribe domain, a usable API must consider the individual needs of different types of users. For example, the publish/subscribe domain distinguishes between information producers (publishers) and information consumers (subscribers). While publishers are concerned with how to represent and publish events; subscribers are concerned with the way notifications are delivered and subscriptions are made. Details about the underlying communication protocols, how events are routed or how to extend and configure the infrastructure, are not relevant to these types of users, instead, these are concerns of infrastructure developers' interest. The infrastructure must, therefore, support these types of users with minimal impact on the API usability.

Hence, another important characteristic of usable APIs is separation of concerns. A well modularized API presents only the necessary information for each type of user, thus shielding them from concerns that are not relevant to their tasks.

In sum, APIs must reflect the tasks of different developers will perform with the infrastructure, matching the abstraction level with that demanded by the tasks and user roles it supports (Clarke 2004).

3.4.2 Flexibility (extensibility & configurability)

Parnas (Parnas 1978) defines flexibility as the ability of software to expand and contract in responses to changes in the application domain. Thus, flexibility implies both extensibility and configurability. While configurability allows an infrastructure to be tailored to the exact set of features demanded by an application domain, extensibility supports the addition of new features.

An ideal publish/subscribe infrastructure is one that can be extended and configured to match the shifting requirements different application domains, thus producing slim and efficient implementations, at a fraction of the cost of developing a new infrastructure from scratch.

3.4.3 Maintainability

The maintainability of an infrastructure is a function of its modularity, architecture, documentation, as well as any extra mechanisms (such as configuration management) that support the process of evolution, correction and configuration of its features (Li and Henry 1993; Kim and Bae 2006).

Hence, highly maintainable software infrastructures are those that support developers in the tasks of correcting, improving, customizing and extending these infrastructures to support shifting application requirements.

3.4.4 Reusability

Reusability is a software quality which goal is to minimize the development effort required to apply the provided features of an infrastructure in support of the required features of the application domain (Krueger 1992). In other words, a piece of software is easy to reuse if its characteristics can minimize the costs of *selection*, *extension*, *configuration*, *adaptation* and *integration*, as discussed in section 2.1.2, while supporting the requirements of the application domain.

3.4.5 Performance

Finally, an infrastructure must not only be *usable*, *flexible*, *maintainable* and *reusable*. It must also support these software qualities without penalizing the performance of the system. It is usually the case that the number of choices and features provided by the infrastructure can jeopardize the performance of the whole system. A typical example of this trade-off is the one observed in hardware industry, between RISC and CISC computer architectures (Jamil 1995). The large amount of features provided by CISC chips penalize simple operations such as addition and subtraction. It originates different design restrictions such as those involved in registers usage rules (certain registers are used for memory access, others for simple arithmetic operations, and others for more advanced operations such as division). RISC design, on the other hand, strives for simplicity, orthogonality and minimalism in its instruction set, making it possible to optimize simple operations, thus achieving better performance (even though more complex operations need to be expressed in terms of primitive operations). The same problem happens in software. A software design must balance the complexity and feature set of the infrastructure in order to improve the performance of the system.

3.5 YANCEES design

In this section, we describe the strategies applied in the design of YANCEES to support of the software qualities we just discussed. YANCEES supports *maintainability*, *flexibility*, *reusability*, *usability*, and *performance* by through the modeling of dependencies as first class entities and the application of different design decisions in the management of dependencies. In doing so, it can reap the benefits of existing approaches to versatility without inheriting their costs. As such, YANCEES applies the following design principles.

First, it supports different interfaces, around the major publish/subscribe concerns, together with a configuration API. Second, it supports variability along the main publish/subscribe dimensions as discussed in Table I, in the form of a micro kernel architecture. This variability is implemented through the use of dynamic and static plug-ins, and extensible languages (Birsan 2005). Third, it manages fundamental and configuration-specific dependencies through the use of dynamic parsers, that handle subscription and notification commands, and through a configuration manager that handles static plug-ins installation and their inter-dependencies. Additional services as reflection are also supported, allowing plug-ins to find each other at runtime. A summary of these design decisions is shown in Figure 11.

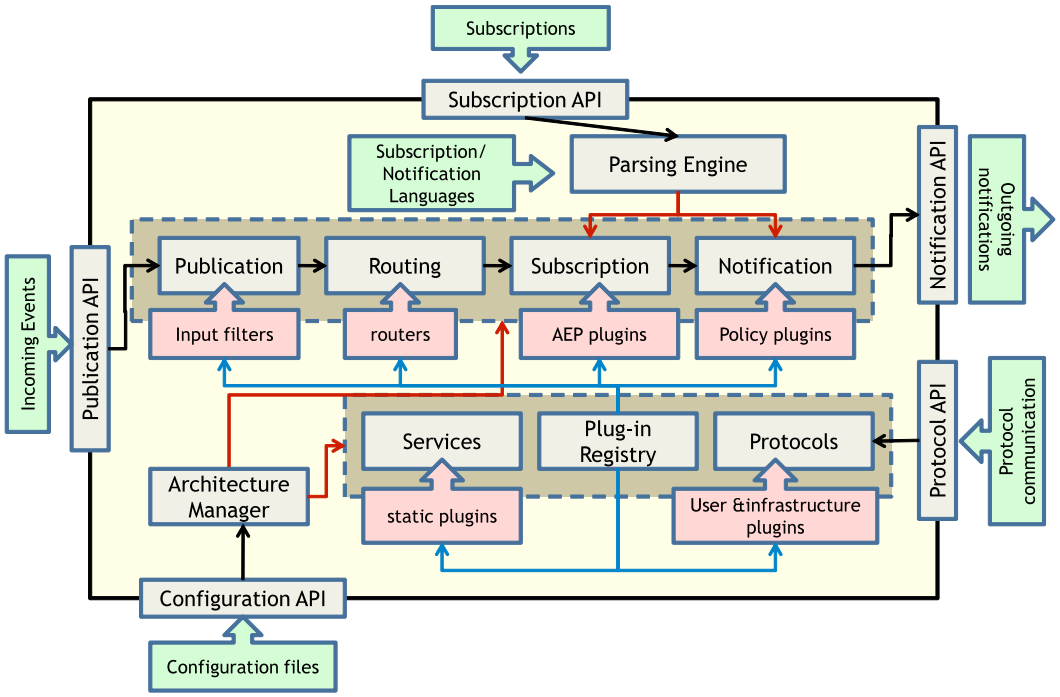


Figure 11 YANCEES high-level architecture

Through the combination of reusable plug-ins and extensible languages, YANCEES supports the development of publish/subscribe infrastructures software product lines (or SPLs) (Clements and L. Northrop 2002). The goal of SPL engineering is *"to capitalize on commonality and manage variability in order to reduce the time, effort, cost and complexity of creating and maintaining a product line of similar software systems"* (Krueger 2006). In SPLs, reuse of commonality allows the reduction of the costs of producing similar software systems, while variability permits the customization of software assets to fit different requirements of the problem domain (Coplien, Hoffman et al. 1998).

As illustrated in Figure 12, YANCEES allows the combination of exiting assets (YANCEE core, existing or custom-made plug-ins, filters, adapters, and application-specific subscription languages) in to the production of domain-specific publish/subscribe infrastructures.

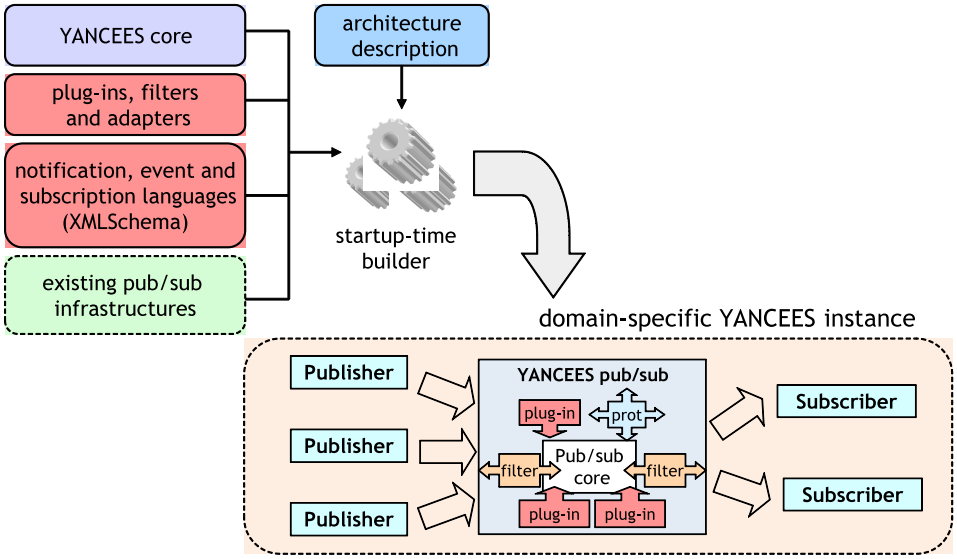


Figure 12 YANCEES approach summary

In the following sections we describe, in more detail, how these different design strategies support *usability, flexibility, reusability, maintainability* and *performance*.

3.5.1 Usability

YANCEES usability is achieved by a combination of different design decisions as follows.

Separation of API concerns. YANCEES separates publication and subscription from the extensibility and configuration APIs (also illustrated in Figure 11). This separation of concerns reduce the development effort of publishers, subscribers and developers by hiding *configuration & extension* concerns from the regular publish/subscribe users, at the same time that it still supports developers in their *extension & configuration* tasks.

Simplicity & Specificity. From application developers' perspective, YANCEES provides a very simple publish/subscribe API, similar to that available in minimal core infrastructures as Siena. In these infrastructures, the only available commands are those concerned with publication and subscription of events. Moreover, YANCEES relies on extensible text-based subscriptions, supporting the development of application-specific subscription languages. This customizability allows the infrastructure to provide the exact amount of features required by each application, thus decreasing the signal-to-noise ratio of the subscription language with respect to the application domain.

Automatic subscription parsing & composition. The dynamism and interactivity of publish/subscribe infrastructures requires special attention to subscription language usability. YANCEES performs the automatic and dynamic allocation of subscription and notification plug-ins, relieving application developers from the tasks of programmatically creating and composing subscription filters (as is the case of Siena, for example), every time a new subscription is created. Moreover, the use of textual subscription also provide automatic syntactic checking, better supporting users in the detection of common subscription errors.

3.5.2 Flexibility

YANCEES supports flexibility through the use of a plug-in oriented architecture (Birsan 2005) supporting extensible subscription and notification languages. In this approach, *modularization*, *stabilization* and *variation* strategies are applied in the production of a common publish/subscribe core that can be extended and configured with user-defined plug-ins.

In YANCEES, plug-ins, implement specific commands in the subscription and notification languages, as well as different publish/subscribe features along the main variation points (represented as rectangles inside the infrastructure shown in Figure 11). The correct composition of plug-ins into valid configurations is supported by the *Architecture Manager* component, at load time; and by the *Subscription Parsers* at runtime.

3.5.3 Reusability

YANCEES achieves a high degree of reusability by adopting a compositional approach that combines feature-specific components with an extensible publish/subscribe framework in the production of application-specific infrastructures. This modularization is performed along the dimensions of Table I. This approach reduces the abstraction distance between required application and provided infrastructure features, and supports the reusability of plug-ins in the construction of different infrastructures.

Reducing abstraction distance. By abstraction distance, we mean the effort necessary to express domain concerns in terms of the provided API as described by (Krueger 1992). YANCEES' ability to produce application-specific infrastructures allows subscription language and infrastructure features to closely match the different application domain requirements. As a result, the abstraction distance between the provided infrastructure and required domain features are reduced, relieving application developers from additional adaptation and mismatch costs.

Reuse of plug-ins. From the point of view of the developers, plug-ins are not only units of extension but also important units of reuse. They modularize individual concerns into reusable components that operate over generalized event representations. Plug-ins can also be reused in the implementation of more complex features. For example, pull notification feature can be implemented by composing pull notification plug-in with a persistence service and a polling protocol plug-in into the same configuration; whereas advanced event processing commands such as sequence detectors and rules can be expressed in terms of lower-level filter plug-ins.

Automation. Finally, the process of runtime composition of subscription and notification commands is automated. Subscription and notification plug-ins are composed based on the syntax of the subscription language. The parsing of subscriptions also support syntactic checking, thus preventing simple subscription errors.

Combined, these characteristics relieve developers from the task of re-implementing existing features from scratch and from enforcing dependencies between different plug-ins, thus improving the development process of application-specific infrastructures.

3.5.4 Maintainability

YANCEES design supports maintainability by modularizing the main publish/subscribe concerns and features in the form of reusable and extensible plug-ins, and by providing automatic configuration management mechanisms. Through the modularization of the different publish/subscribe concerns in the form of plug-ins, existing features can be more easily corrected, updated and extended with little impact to the core system components. Configuration manage-

ment automation allows dependencies between different plug-ins to be automatically enforced, thus relieving developers from manually checking for dependencies and compatibility.

3.5.5 Performance

YANCEES can achieve high levels of performance by supporting specialized features that match specific commands of the subscription language. For example, YANCEES allows the co-existence of different routing cores and specialized subscription commands. By posting specialized commands, users can implicitly choose the best routing algorithm for their needs. For example, topic-based subscriptions are routed through a specialized topic-based core; while more complex content-based queries are handled by a more capable content-based filtering core.

Moreover, by supporting static and dynamic configuration of plug-ins, only the necessary features for the application at hand are loaded at a given time. This approach reduces the runtime footprint and the total size of the server, supporting the development of applications on more restricted devices.

3.5.6 Additional benefit: interoperability

YANCEES ability to support multiple cores also supports the interoperability of the infrastructure with existing publish/subscribe networks, for example: Siena, Elvin or CORBA-NS. Events can be published to or subscribed from these different infrastructures, allowing YANCEES to be used as an advanced event processing layer on top of these systems in an approach similar to that described at (Heimbigner 2003), but with the additional configurability and extensibility provided by the infrastructure.

3.5.7 Versatility supporting concerns

As briefly discussed in the previous sections, the *usability*, *flexibility*, *reusability* and *performance* benefits of YANCEES come at some costs that must be adequately managed in order to reap the benefits previously discussed. This section makes these costs evident, discussing their role in the support of YANCEES versatility.

First, there is a need for **configuration management**. As previously discussed, fundamental and configuration-specific dependencies limit the reusability of existing plug-ins, creating incompatibilities and invalid configurations. In YANCEES, an architecture manager component is defined in order to enforce these dependencies, preventing the creation of invalid configurations. YANCEES configuration manager assures the proper installation of plug-ins based on information provided in configuration files. Upon start, the architecture manager builds a valid YANCEES instance, enforcing dependencies between plug-ins. In case of broken dependencies or invalid configurations, error messages are generated and the server startup is interrupted.

Second, there is a need for mechanisms that support the **dynamic composition** of plug-ins. Subscription and notification plug-ins must be composed, at runtime, in response to different subscription and notification commands. Each subscription specifies an expression on the content or order of events that may rely on different commands, for example: content-based filtering, sequence detection or rules. It also specifies notification policies such as push or pull. In YANCEES, this process is supported by *Notification* and *Subscription* managers (shown in Figure 14), that first validate the subscription commands, based on grammar rules; and then build dynamic event processing hierarchies using the installed plug-ins.

Third, there is a need for mechanisms that support **static plug-ins**. While subscription and notification plug-ins are allocated per-subscription basis, certain features need to be constantly available in the infrastructure, for example, protocol plug-ins that must listen to certain ports in

the network, or persistency services that stores events for later retrieval. Therefore, YANCEES supports both static and dynamic plug-ins. Static plug-ins are also know as services. They are loaded at startup.

Fourth, it is usually the case that complex features are implemented not by a single plug-in but by a combination of plug-ins. For example, pull notification requires a plug-in to implement the event notification queue, and another plug-in to implement the polling protocol. Hence, plug-ins require **architectural reflection mechanisms** that support the communication and location of plug-ins in the system. In YANCEES, this service is provided by a plug-in registry that supports plug-in registration and location by name.

3.5.7.1 Generalized event representation

Due to fundamental problem dependencies, the event format representation has an important role in the overall software reusability and maintainability.

In YANCEES original design, events were initially designed as XML messages. Whereas the ability to represent events in this format supported custom-made messages, an analysis of dependencies (shown in Figure 9) revealed the high change impact that this design decision would have. Hence, we simplified YANCEES original design to support fixed, but generalized, event representations. As a result, YANCEES events are represented as attribute/value pairs, that through adaptation, can be used to represent different event formats without impacting other components of the infrastructure such as the routing algorithm and the existing filter subscription plug-ins. This simplification improved software maintainability, through reduced change impact, as well as the overall reusability of plug-ins, that do not need to change every time a new event format needs to be supported.

3.6 YANCEES implementation

In this section, we describe the detailed design and implementation of YANCEES, highlighting the use of the versatility strategies we discussed in section 3.3.

The original publish/subscribe pattern, shown in Figure 13, is relatively simple. It defines a common interface that allows the publication and subscription of events. The infrastructure routes and delivers events based on the supported notification policy. In spite of this simplicity, this original design is inflexible. Due to fundamental and configuration-specific dependencies, changes in different design characteristics are usually difficult, and require the co-evolution of different parts of the software. The challenge in the development of YANCEES is to redesign this simple pattern in a way that supports extensibility and configurability along the main publish/subscribe variability dimensions.

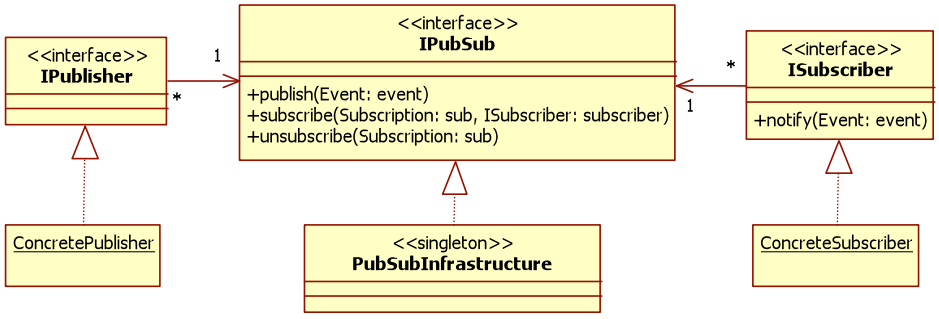


Figure 13 Publish/subscribe pattern

Through the application of different versatility strategies, YANCEES augment this initial design introducing different variation and extension points in the software as follows.

3.6.1 Applying stabilization & variation

We first applied stabilization and variation strategies in the separation between the common publish/subscribe process (that was stabilized into YANCEES core) and the different publish-subscribe concerns around the common publish/subscribe process, i.e, its publication, subscription, routing, notification and protocol models (implemented as plug-ins). As shown in Figure 14, in YANCEES these concerns are implemented with the help of individual components (or façades). Note that, in Figure 14, we only show the main classes, suppressing the methods and attributes of these classes for simplicity. Each one of these façades define extensibility interfaces and configuration rules that support the implementation of features in these variation points (or models). In the following paragraphs, we discuss each one of these models in more detail.

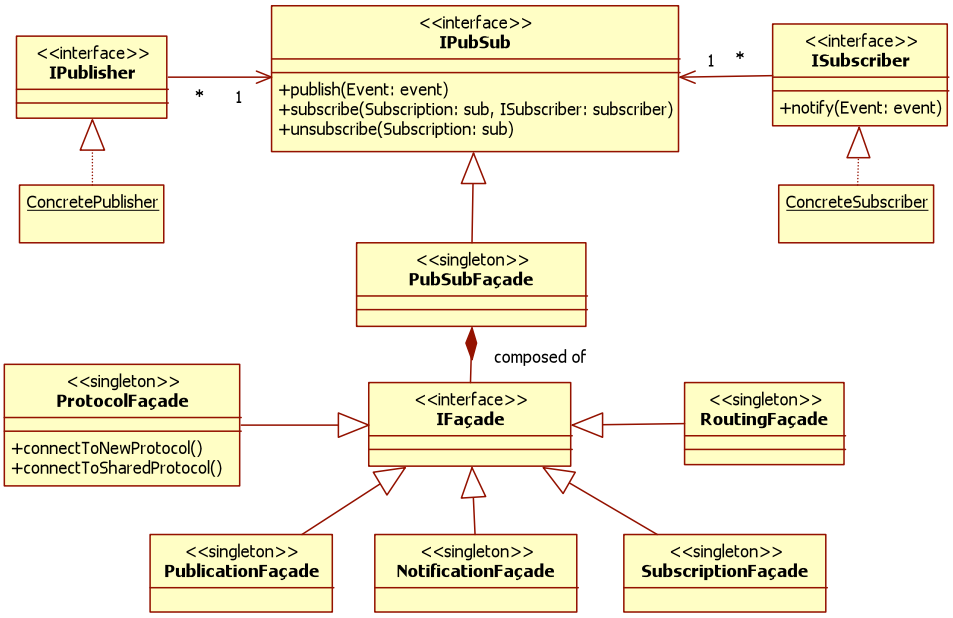


Figure 14 YANCEES main components (façades)

3.6.2 Routing model

The *RoutingFaçade* can support different routing strategies simultaneously, for example: content-based, topic-based or channel-based. Adapters intermediate the communication between the *RoutingFaçade* and different routing strategies implementations. The routing model also supports the interoperability of YANCEES with existing publish/subscribe infrastructures (such as Elvin (Fitzpatrick, Mansfield et al. 1999) and Siena (Carzaniga, Rosenblum et al. 2001)) through adapters that implement the *DispatcherAdapterInterface* as shown in Figure 15. Moreover, existing infrastructures and custom routers can co-exist, in the same infrastructure, allowing the selection of the best routing strategy for each subscription command. This approach also copes with performance, supporting the use of specialized routers for different commands.

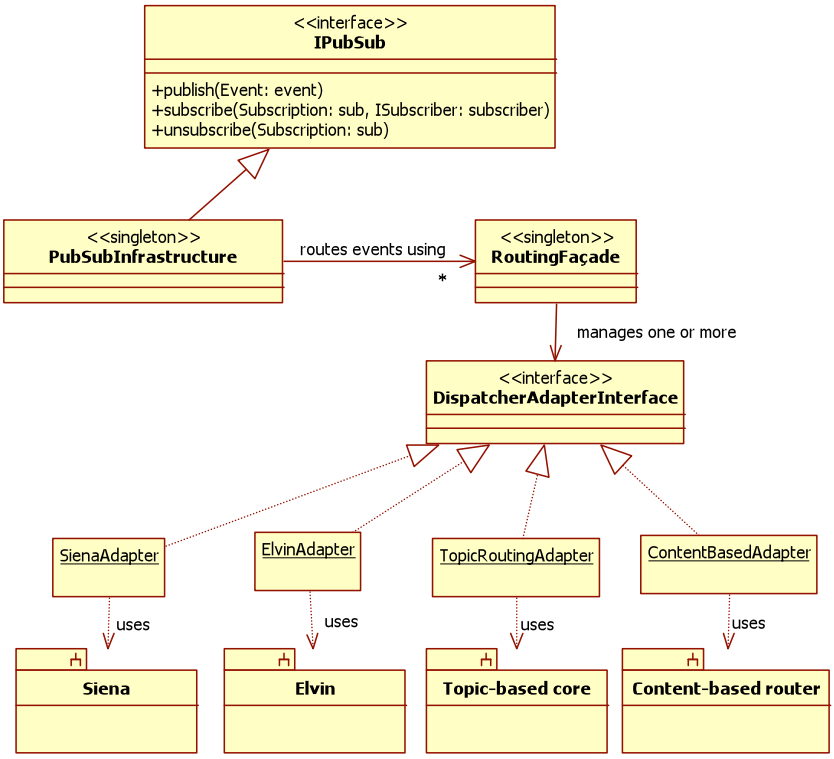


Figure 15 Support for multiple routing strategies and interoperability with different routers

3.6.3 Publication Model

The publication model is supported by the *PublicationFaçade,* which is implemented by the composition of different filters that extend this model using the *Chain of Responsibility* design pattern (Gamma, Helm et al. 1995), as illustrated in Figure 16. These filters can be extended and configured to implement global policies such as repeated events filtering, or work together in the implementation of more complex features, for example, the creation of peer-to-peer publish/subscribe networks. In this last example, further discussed in 5.3, publication plug-ins intercept and route selected events to all peers in the network with the help of a protocol plug-in.

The use of publication filters, however, creates the need for the enforcement of proper order of these filters. Changes in their order, for example, can lead to undesirable conditions or feature interference (Silva Filho and Redmiles 2007). For example, a publish-to-peers filter redirector should be installed after a repeated events filter. The change of this order will result in duplicate events being published to all routers in a peer-to-peer network, instead of only a subset of these events. In order to remedy this situation, YANCEES allows the definition publication filters priorities, described in its configuration file.

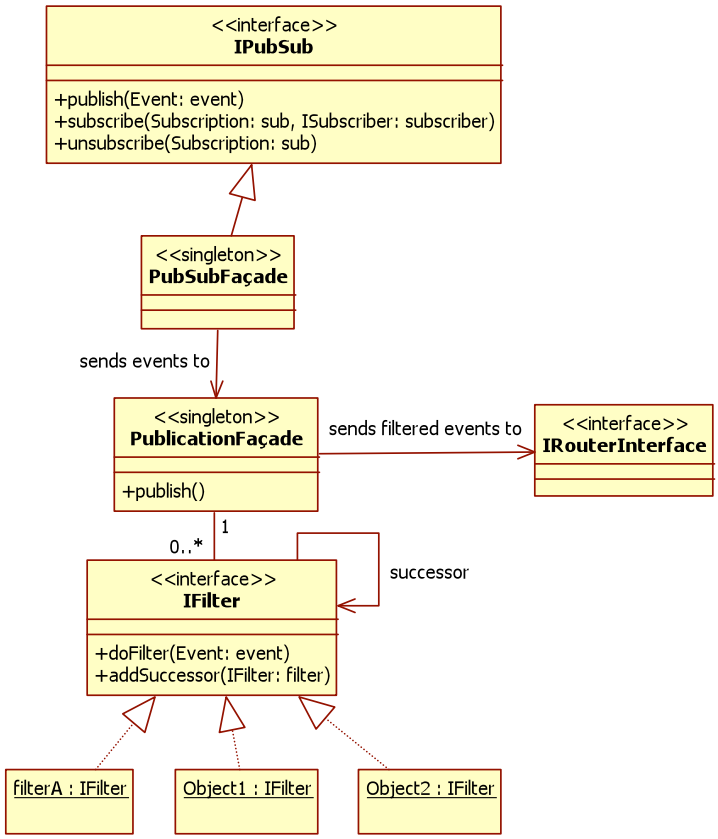


Figure 16 YANCEES Publication Model

3.6.4 Subscription Model

As illustrated in Figure 17, the *SubscriptionFaçade* is responsible for parsing the subscriptions posted by the users, and for assembling individual event processing trees based on these subscriptions. The subscription language is expressed in XML, having their grammars specified in *XMLSchema*[1], an IETF standard that supports XML extensibility.

Whenever a new subscription is posted in the system, plug-ins are automatically allocated and composed. This automatic allocation of plug-ins is facilitated by the plug-in registry, that does the translation between plug-in names (that can be commands in the subscription language) to the appropriate plug-in implementation. Plug-ins are implemented by extending the *ISubscriptionPlugin* interface, being installed in the subscription model at load-time, based on the information provided in a configuration file. A simple configuration file defines a set of unique plug-in tags and the main Java class that implements it. It can also include dependencies with other plug-ins, allowing the load-time checking for broken dependencies.

[1] http://www.w3.org/XML/Schema

Note that, in the subscription model, plug-ins can depend on one another to implement more sophisticated features. In Figure 17, the *Abstraction*, *PatternMatching* and *Sequence* plug-ins depend on the *Filter* plug-in to implement their features; whereas the Filter plug-in directly interacts with the existing routing capability of the infrastructure. Besides supporting the runtime allocation of plug-ins, the *PluginRegistry* provides a central point of access that allow plug-ins to locate each other at runtime. It works as a look-up table, translating plug-in names into runtime references. As previously mentioned, these dependencies are also declared in the configuration file of the infrastructure, where they are used for configuration management.

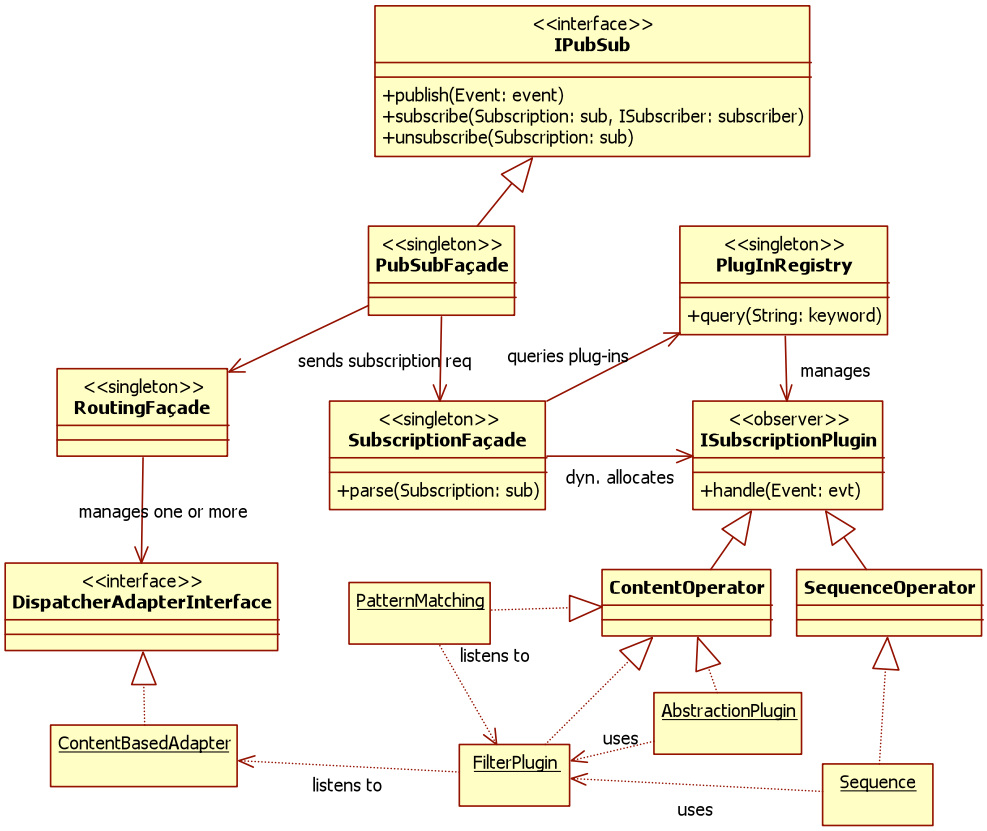


Figure 17 YANCEES Subscription Model

3.6.5 Event Model

Differently from the other models, YANCEES event model is fixed, being implemented by the *YanceesEvent* class depicted in Figure 18. Its design is a consequence of the application of the *generalization* operator in the representation of events as user-defined typed attribute/value pairs. These typed attribute/value pairs support the basic language types (*boolean, float, double, int, long*), *Strings*, and byte array (or *byte[]*). Byte arrays can be used to hold serialized objects or different data types. The *YanceesEvent* class also provides convenience set/get methods supporting the *Object* Java type. These methods can automatically serialize/de-serialize objects for transmission between publishers and subscribers.

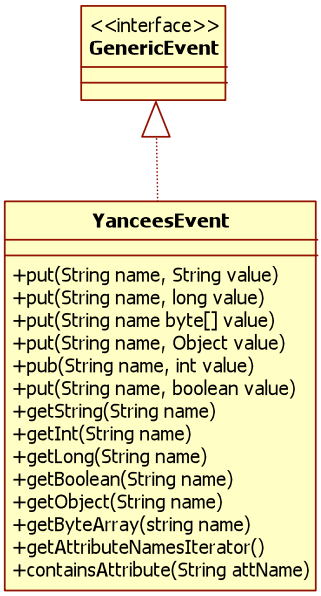


Figure 18 YANCEES Event Model

3.6.6 Notification Model

Similar to the Subscription Model, the Notification model supports the dynamic parsing and allocation of notification plug-ins, implementing different notification policies such as pull or push.

In the example of Figure 19, we present the notification model. We also show the interaction between notification, protocol and service (or static) plug-ins. In this example, the pull notification stores notifications for further retrieval with the help of a *Persistence* Plug-in. These events are later collected by the users through the polling service provided by the *Poll* protocol plug-in. Similarly to subscription plug-ins, notification policies are allocated, at runtime, by command name, with the help of *PluginRegistry*, which also supports the runtime location of other components in more complex features such as persistency.

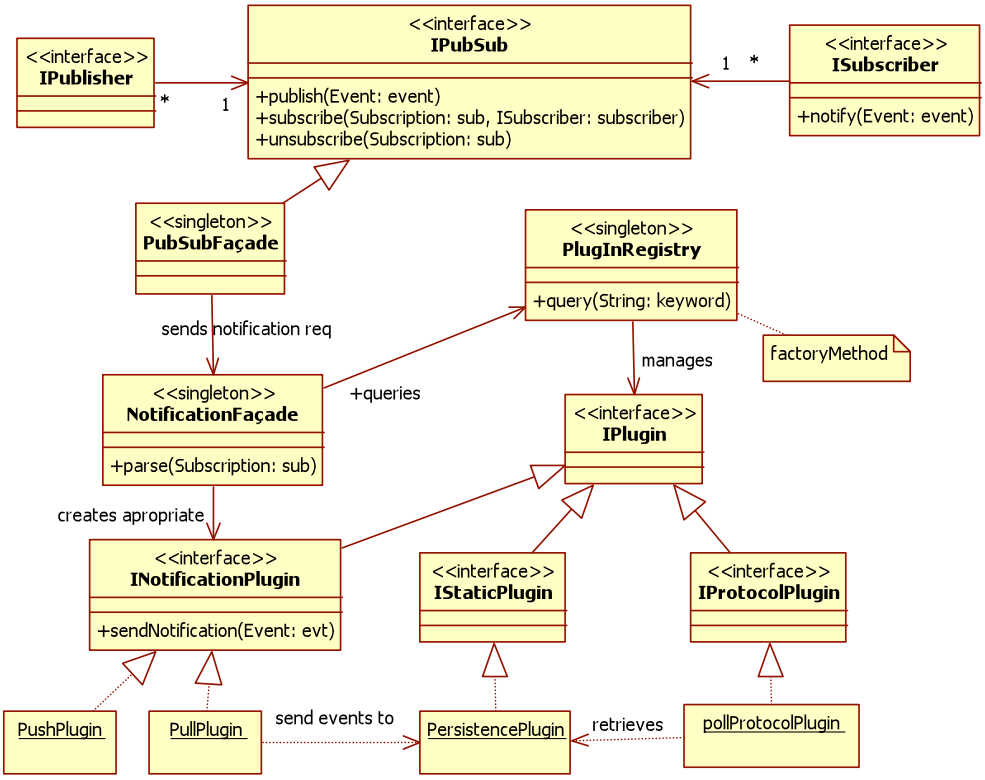


Figure 19 YANCEES Notification Model

3.6.7 Protocol Model

The protocol model, illustrated in Figure 20, supports both the interaction with end-uses and with other infrastructures. The protocol model is very general. Protocol plug-ins must only implement a simple interface that allows its location in the infrastructure. Protocol plug-ins can be either static or dynamic. Static plug-ins work in the same way as services. They are allocated at load time. The communication between clients and server through the protocol plug-ins is, by default, supported by Java Remote Method Invocation (or RMI). However, developers are free to use other communication protocols as needed. In the example of Figure 20, two protocol plug-ins are available: *PeerLocator* and *PublishToPeers*, that together support the Peer-to-peer federation of routers in YANCEES.

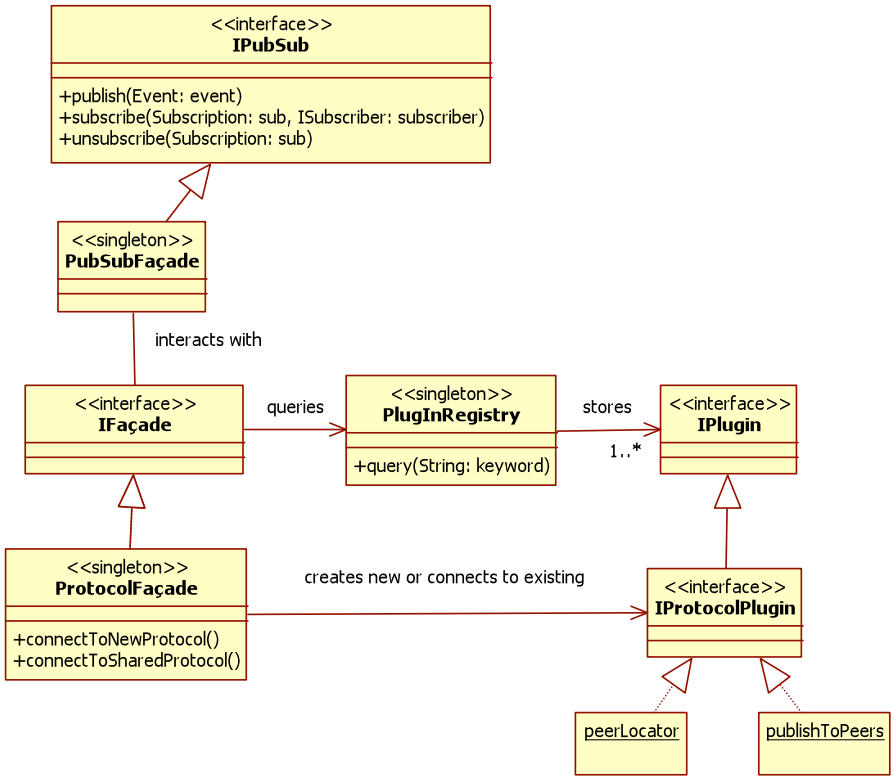


Figure 20 YANCEES Protocol Model

3.6.8 Overall Architecture

A summary of YANCEES main components is shown in Figure 21. As summarized in this figure, YANCEES employs stabilization and variation in support of different publish/subscribe concerns. These concerns are extended by different types of plug-ins, composed both statically and at runtime, according to different strategies. The process of subscription and notification composition is automated by runtime parsers; while the architecture manager handles static composition, guaranteeing the overall compatibility of plug-ins installed in the system. The plug-in registry supports the location and activation of plug-ins at runtime. Finally, YANCEES design employs generalization in the definition of its event representation.

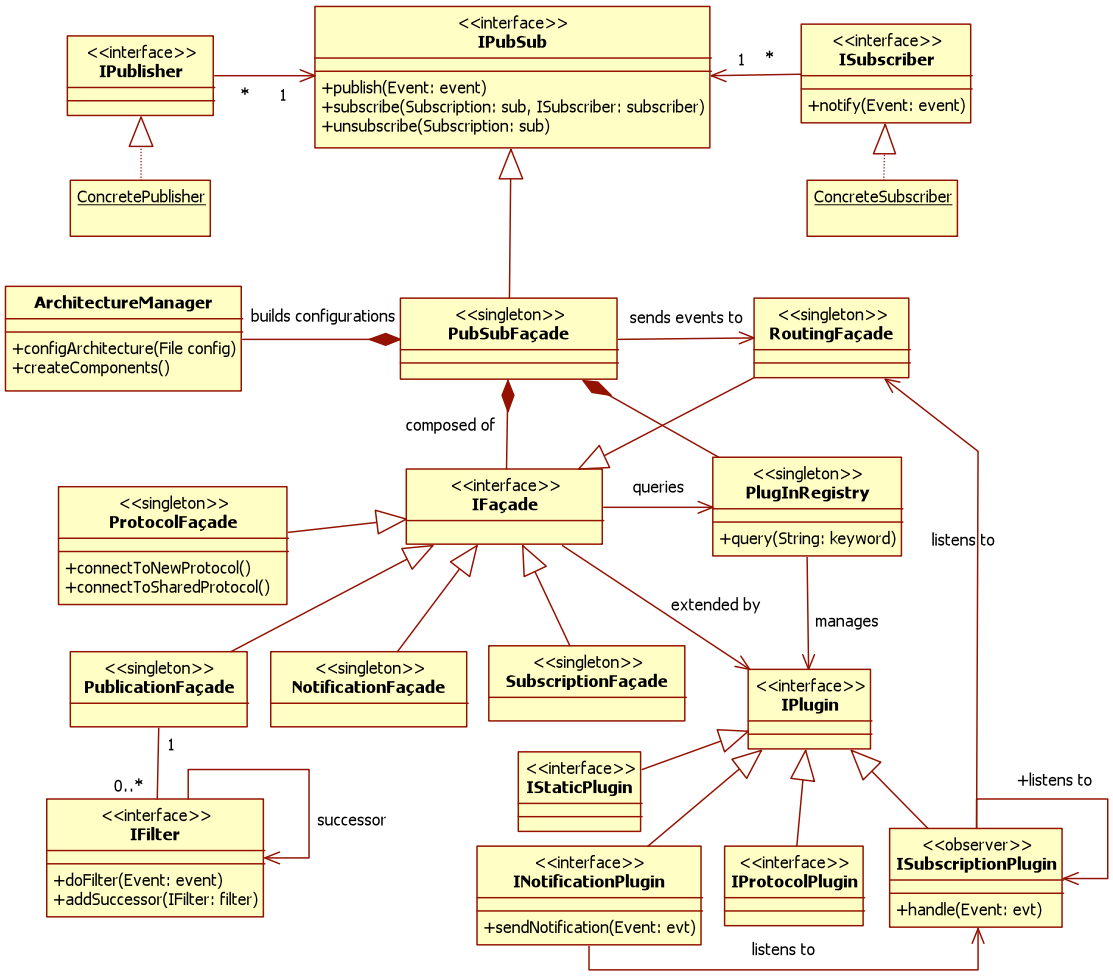


Figure 21 YANCEES general approach

3.7 Applications supported by YANCEES

YANCEES has been used as the basic publish/subscribe infrastructure in different applications including peer-to-peer file sharing (DePaula, Ding et al. 2005), contextual collaboration services (Geyer, Silva Filho et al. 2008), pocket-size devices (Silva Filho and Redmiles 2006) and collaborative software engineering (Redmiles, van der Hoek et al. 2007), besides of being evaluated in the case study described in this paper. The details on how YANCEES was customized and extended to support some of these applications are discussed in section 4.3, when we discuss the implementation of our case studies.

3.8 Summary

In this section, we discussed YANCEES design, showing how it achieves a favorable balance between *usability*, *reusability*, *performance*, *flexibility* and *maintainability* requirements of versatile publish/subscribe infrastructures. We also showed how the versatility operators discussed in section 2.1.1 were used in its design and implementation.

As such, YANCEES provides as an architectural style that addresses the problem of configuration management and reflection induced by the need of flexibility. By modeling software dependencies as first class entities, it provides solutions to the problems of static and dynamic configuration management induced by problem domain and configuration-specific dependencies, at the same time that reduces these costs by supporting generalized event representations in the form of attribute/value pairs. Finally, YANCEES support for automation and separation of API concerns improves application developers usability, allowing them to reap the benefits of flexibility without its costs.

In the next sections, we discuss the design and implementation of a set of case studies where we quantitatively and qualitatively evaluate YANCEES, comparing it with existing versatility approaches.

Chapter 4. Case Studies Design

YANCEES flexible approach is not the only way to support the variability and evolution of application domain requirements. In fact, different academic and research infrastructures have been developed that support event-driven application domain variability. As will be later analyzed, they employ different versatility strategies, which have their own benefits and costs.

This chapter describes the design of three case studies with which we compared major versatility approaches in the publish/subscribe domain. These case studies were designed according the following steps.

- First, we conducted a survey of publish/subscribe versatility approaches (Silva Filho and Redmiles 2005). In this survey, we identified four major approaches employed in the construction of versatile publish/subscribe infrastructures. These were: minimal core, coordination languages, one-size-fits-all and flexible compositional approaches.
- Second, we selected a set of open source infrastructures, one for each versatility approach above, to be compared and analyzed. These infrastructures were: Siena (Carzaniga, Rosenblum et al. 2001) representing minimal core infrastructures; Sun JavaSpaces (Freeman, Hupfer et al. 1999) representing coordination languages; CORBA Notification Service (or CORBA-NS) (OMG 2004) representing one-size-fits-all infrastructures, and YANCEES (Silva Filho and Redmiles 2005) representing flexible compositional infrastructures.
- Third, we selected three feature-rich event-driven application domains as sources of requirements for our case studies. These were: usability monitoring represented by EDEM (Hilbert and Redmiles 1998), awareness represented by CASSIUS (Kantor and Redmiles 2001) and groupware represented by IMPROMTU (DePaula, Ding et al. 2005). These infrastructures were selected first for their diversity of requirements, and second, for the previous experience of the authors in their development, which provides us with both access to the source code, and expertise in their set of requirements and algorithms.
- Forth, the requirements of each application domain were abstracted into individual reference APIs, representing ideal sets of features that publish/subscribe infrastructure must support for each domain. These APIs provide a common ground for comparing the different metrics of our study.
- Fifth, we implemented each one of these tree reference APIs using the four selected infrastructures. We also implemented each API from scratch, as base line comparisons.
- Sixth, we performed a quantitative and qualitative analysis of the resulting implementations, measuring different software qualities.
- Seventh, we identified trade-offs and derived a set of guiding principles to inform both developers and users.

4.1 Publish/subscribe versatility approaches

In a previous survey of versatility strategies (Silva Filho and Redmiles 2005), we identified four major versatility approaches employed in the construction of both industrial and research publish/subscribe infrastructures. We describe them in more detail in the following sections.

4.1.1 Minimal core infrastructures

Minimal core infrastructures such as Siena (Carzaniga, Rosenblum et al. 2001), Herald (Cabrera, Jones et al. 2001), Scribe (Castro, Druschel et al. 2002), and to a certain extend, Sun JMS (Sun Microsystems 2003) provide simple but optimized services that support the efficient routing of events in distributed publish/subscribe networks. As such, they support the most common and essential publish/subscribe features, provided in the form of simple and generalized APIs. In this approach, application-specific requirements such as advanced event processing, alterative notification policies and protocols are not directly supported. Instead, they must be implemented by the application developers themselves, based on the sets of primitive features provided by the core functionality of each infrastructure.

Moreover, generalization is widely adopted: event representations, such as attribute/value pairs, and content-based filtering capabilities are applied in the implementation of the most common publish/subscribe features. These infrastructures are therefore designed to be reused as black box routing components on top of which different event-driven applications and their application-specific features are built.

4.1.2 Coordination languages

Coordination languages such as SUN JavaSpaces (Freeman, Hupfer et al. 1999), IBM TSpaces (Wyckoff 1998) and LIME (Murphy, Picco et al. 2006) are based on the Linda (Gelernter 1985) coordination model. As such they implement a "virtual machine" approach as proposed by (Parnas, Clements et al. 1984), in the form of a persistent space of entities (or tuple space), that can be accessed through a minimal and fixed set of operations. These operations are: *read(), take(), write()* and *notify()*. Which respectively support the reading, removal, addition and notification of shared persistent objects known as tuples. These operations also support the concept of anti-tuples (or templates), that work as content-based filters, allowing these commands to be applied to a range of tuples in the space. In this approach, application-specific features can be built by composing and successively applying these primitive commands. For example, allowing tuple spaces to act as a full-fledged content/based publish/subscribe routers, as described by (Zavattaro and Busi 2001).

In commercial systems, tuple spaces are usually augmented with additional (optional) features such as transactions, leasing and authentication that are easily available as parameters of these space basic operations. Infrastructures such as LIME support additional commands for mobility.

4.1.3 Configurable one-size-fits-all

Configurable one-size-fits-all infrastructures such as CORBA Notification Server (or CORBA-NS in short) (OMG 2004) and READY (Gruber, Krishnamurthy et al. 1999) support a broad set of application domain requirements by integrating different features and qualities of service (or QoS) into a single, configurable infrastructure. They are built on the premise that the way to support domain variability is to provide variation, maximizing the number of options and features supported by the infrastructure. In this approach, users can select among different combinations of event representations, notification policies, routing strategies and qualities of service, out of an existing (and specialized) pool of options along most publish/subscribe variability dimensions discussed in section 3.3. These options can then be combined into valid configurations in support of specific application domain requirements.

4.1.4 Flexible publish/subscribe infrastructures

Flexible (configurable and extensible) publish/subscribe infrastructures such as YANCEES (Silva Filho and Redmiles 2005), FACET (Pratap, Hunleth et al. 2004) and DREAM (Leclercq, Quema et al. 2005) strive to combine the simplicity, generality and efficiency of minimal core infrastructures with the configurability and variability of one-size-fits-all systems. They do so by separating policy and mechanism (Wulf, Cohen et al. 1974) in the development of infrastructures that can be expanded or contracted to address the specific requirements of different applications (Parnas 1978).

For example, FACET separates common publish/subscribe behavior and variable features into a fixed base code and variable aspects (in AOP sense), that implement the different publish/subscribe features. YANCEES separates the common publish/subscribe process into a component framework extensible through plug-ins and extensible languages, whereas DREAM provides a component framework with different feature-specific components that are programmatically combined in the implementation of application-specific infrastructures. Since not all possible combinations of features are feasible, flexible infrastructures rely on different mechanisms to automate the process of (re-)combining common publish/subscribe code with feature-specific components in the production of different, and coherent, application-specific publish/subscribe infrastructures (as previously discussed in Chapter 3).

4.1.5 Comparing the versatility of different strategies

Each one of these approaches employ a specific set of design strategies. In Figure 22, we lay these infrastructures with respect to their degrees of generalization and flexibility. Note that a solution to a problem is general if it can be applied, without change, in as many situations as possible; whereas it is flexible if it can be tailored (configured and extended) to better match the problem at hand (Parnas 1978).

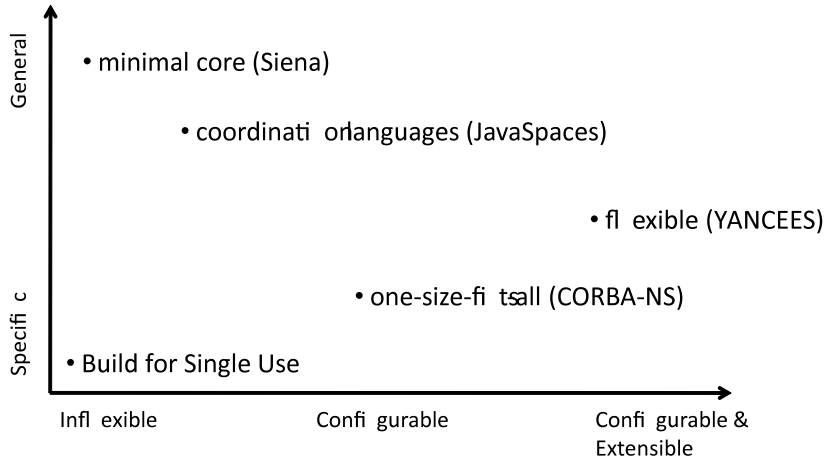


Figure 22 Comparative analysis of different versatility design considering their generality, specificity and flexibility

As illustrated in Figure 22, minimal core infrastructures adopt generalization in their event and subscription representations, supporting general but fixed APIs. Coordination Languages also employ generalization in tuple representation and filtering, in the construction of a simple tuple manipulation API. Most tuple space systems support optional features, which increases their ranking in terms of configurability. One-size-fits-all systems support configurability and different

qualities of services around a fixed set of specialized features; whereas flexible infrastructures support extensibility and configurability (with footprint management) of major pub/sub concerns. In doing so, they adopt design strategies that end up compromising important software qualities such as *usability*, *reusability*, *performance*, *flexibility* and *maintainability*.

In order to practically analyze and compare these approaches, we selected individual infrastructures to use in our case studies. These infrastructures are described in more detail in the next section.

4.2 Selected publish/subscribe infrastructures

The set of infrastructures used in our case studies were chosen to represent each one of the publish/subscribe versatility approaches discussed in section 4.1. We sought to analyze a set of infrastructures that were implemented using the same programming language (Java in this case), that were mature enough for our case study, and that provided free source code access. We further discuss the selected infrastructures, and their characteristics as follows.

4.2.1 Siena

Siena (Carzaniga, Rosenblum et al. 2001) is an Internet-scale publish/subscribe router. Siena's subscription model supports content-based filtering and event sequence detection (conjunction-style pattern matching). The event model is tuple-based and the notification model is push. Siena's protocol model applies advanced subscription advertisement and event routing algorithms to adequately route events published in one side of the network to subscribers in nodes that are routers away from the event source. The version utilized in our benchmark (version 1.5.5) guarantees partial event ordering with best-effort routing, implying no event delivery or order guarantee. Siena's basic features and components are depicted in Figure 23 as follows.

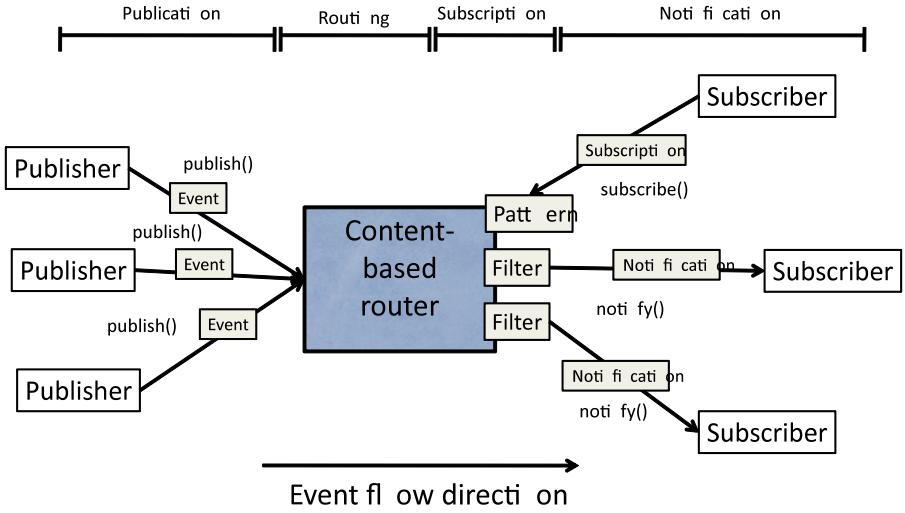


Figure 23 Siena architecture

Figure 23 shows a single router representing the logically centralized architecture of Siena. In Siena, the content-based router responds to different subscriptions (represented as either a content filter or pattern – a set of filters). Events produced by publishers are routed to selected subscribers whenever the event content matches their respective filter expressions. Routers can be federated in the construction of arbitrarily complex routing networks.

4.2.2 CORBA-NS

The CORBA Notification Service (CORBA-NS in short) (OMG 2004) is an Object Management Group (OMG) standard specification. It extends the existing CORBA Event Service (or CORBA-ES) (OMG 2001) to support a broader set of qualities of service (or QoS) such as: event notification reliability, priority, ordering, and timeliness. CORBA-NS is backward compatible with CORBA-ES. Both the original Event Service interfaces, and the new CORBA-NS interfaces, are available. In our study, we used version 1.4.0 of an open-source implementation of CORBA-NS called Community OpenORB[2].

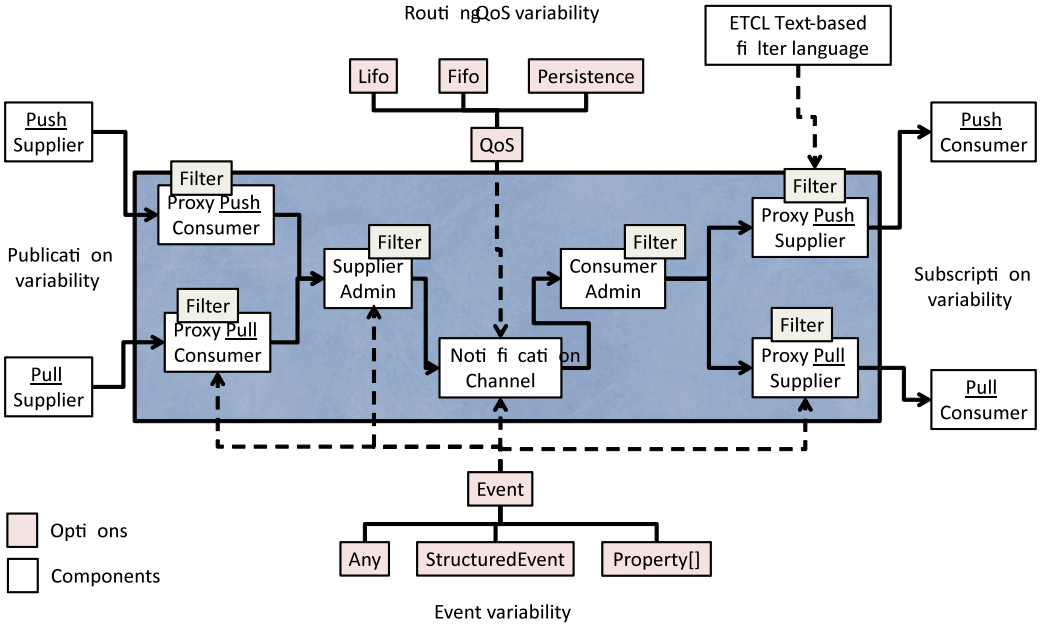


Figure 24 CORBA-NS main components

The CORBA-NS routing model supports both topic and channel-based routing, as well as content-based filtering of events. Events can be typed, un-typed (CORBA::Any) or structured (a mix of both). The interaction with the server is mediated by a hierarchy of proxies, administrative interfaces and filters. Administrative interfaces allow the specification of different channel QoS such as: event guaranteed delivery, persistency and time to live. Secure channels can also be established between publishers and subscribers. Subscriptions are supported through the use of filters, attached to proxy suppliers or consumers. Filters are programmed using the ETCL constraint language, an extension to the TCL (Trader Control Language). The event delivery can be performed using either pull or push notification policies.

An architectural representation of CORBA-NS basic components is shown in Figure 24. In this diagram, different consumer proxies are used for different consumer configurations (either pull or push); whereas the equivalent variety of proxies are supported in the event supplier side. *Admin* objects (shown in Figure 24) are used to create instances of proxies and to select among existing notification channels, configured according to different QoS.

[2] http://openorb.sourceforge.net/

4.2.3 JavaSpaces

The tuple space model, as implemented by Sun JavaSpaces (Freeman, Hupfer et al. 1999), extends the traditional Linda API with Database Management Systems (or DBMS) features such as transactional semantics, supporting, for example roll-back of operations. It also supports event notification (through the *notify()* command), allowing applications to be notified when new tuples matching the provided anti-tuple are posted to the space. The basic primitive operations supported by Sun JavaSpaces are: *notify(), read(), readifExist(), take(), takeIfExist()* and *write()*. All of them have different parameters including anti-tuples, that work as simple content filters for the tuple in the space.

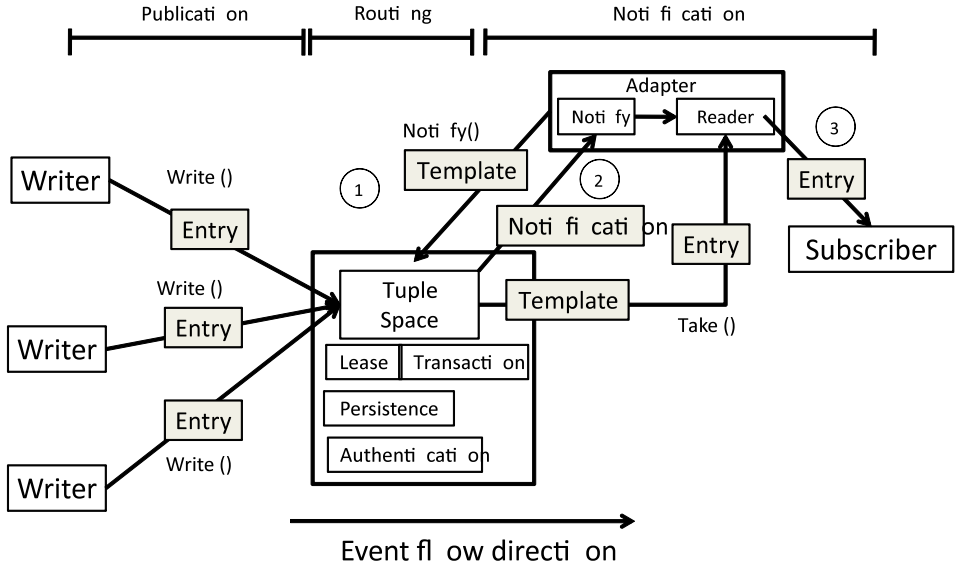


Figure 25 JavaSpaces architecture (with client-side adaptation)

Differently from the *subscribe()* command commonly found in existing publish/subscribe infrastructures, the JavaSpaces *notify()* command does not include a copy of the entries (tuples) that triggered the notification. It also does not automatically remove the entries from the space in response to a notification. Hence, as illustrated in Figure 25, in order to implement a push notification policy compatible with existing publish/subscribe infrastructures semantics, a set of extra steps are necessary. First, one should subscribe to selected types of tuples using the *notify()* command, passing an anti-tuple as a parameter; then, whenever new tuples are written in the space matching the provided template, the new tuples notification should be handled (1). After that, a *take()* command, matching tuples out of the space should be performed (2), followed by the notification of the subscribers with the new tuple (3).

4.2.4 YANCEES

YANCEES (Silva Filho and Redmiles 2005) is a flexible publish/subscribe infrastructure that supports extension and configuration of features along the main publish/subscribe design dimensions shown in Table I. YANCEES design and implementation were extensively discussed in Chapter 3.

4.2.5 Summary of selected infrastructures design decisions

In this section, we summarize the differences between the selected infrastructures, comparing their main characteristics. We chose to classify these infrastructures with respect to: 1) the amount of features they support; 2) the way they represent features; 3) the way they support feature selection; 4) the way feature extension is supported, and the underlying communication technology adopted. These characteristics will be useful in our case study evaluation. These results are summarized in Table II.

Table II Comparison of the characteristics of the selected infrastructures

	Siena	Java Spaces	CORBA-NS	YANCEES
Feature set	Fixed, minimal	Fixed, with optional	Configurable, specific and optional	Configurable, extensible
Decomposition approach	Monolithic	Monolithic	Methods and proxies	Plug-ins
Configuration approach	None	Manual: method parameters	Manual: factories, proxies	Automatic
Reusability approach	Black box	Black box	Black box	Grey box
Underlying communication	Sockets	RMI	ORB	RMI

As shown in Table II, with respect to the variability of features supported, both Siena and JavaSpaces are fixed. JavaSpaces, however, supports different optional features that can be selected through the use of valid parameters in its API. CORBA-NS supports configurability, allowing existing features to be selected and combined in support of different application domains. Finally, YANCEES is flexible, supporting configurability and extensibility.

With respect to the configuration mechanism adopted, CORBA-NS relies on proxies, factories and configuration methods; whereas YANCEES configurability is supported both statically and dynamically through automatic configuration managers. YANCEES also supports extensibility, allowing new plug-ins to be implemented.

With respect to the reusability approach, YANCEES is different from the other approaches. While the other infrastructures are reused as black boxes, supporting a layered extension mechanism, YANCEES supports extensions in the form of plug-ins, installed in predetermined variation points of the infrastructure itself.

Finally, with respect to underlying communication protocol, JavaSpaces, CORBA-NS and YANCEES rely on remote method invocation mechanisms, whereas Siena is implemented using Sockets.

In the next section, we discuss the selected event-driven applications used in our evaluation.

4.3 Selected event-driven applications

Our case studies were based on existing event-driven applications i.e. CASSIUS, EDEM and IMPROMPTU. In this section, we describe the publish/subscribe requirements of these applications, abstracting them in the form of ideal APIs.

4.3.1 CASSIUS

CASSIUS (Kantor and Redmiles 2001) is a notification server designed to support the development of awareness-based applications. A distinctive feature of CASSIUS is its protocol model. It supports the ability to manage information source hierarchies, allowing end-users to advertise, browse, and subscribe to events from different sources using those hierarchies. CASSIUS uses a fixed record-based event model, with its own set of fields. CASSIUS subscription model is content-based, i.e. it supports logical expressions on the entire content of the event attributes (with the exception of some binary fields). Valid subscription operators include: 'and', 'or' and 'not', '<', '>', '<=', '>=','==', and the wild card '*'. Subscriptions are expressed in a textual way. The notification model is pull, supporting the retrieval of events received before or after a certain time stamp.

CASSIUS reference API is shown in Table III. Note that it supports the concept of accounts that manage sets of event sources (objects) and lists of typed events they produce.

Table III CASSIUS reference API

```
public interface ICassiusNotificationServerAPI {

// Account management
public void createAccount(String accountName,
       String description) throws CassiusNSException;
public void deleteAccount(String accountName)
       throws CassiusNSException;
public ICassiusAccount[] listAllAccounts()
       throws CassiusNSException;
public String addObjectToAccount(String accountName,
              String objName, String objType,
              String parentID, String description)
              throws CassiusNSException;
public void removeObjectFromAccount(String accountName,
            String objectID) throws CassiusNSException;
public ICassiusObject[] listAccountObjects
       (String accountName, String parentId)
       throws CassiusNSException;
public void addObjectType(String accountName,
            String typeName, String[] eventNames,
            String description) throws CassiusNSException;
public ICassiusObjectType getObjectType(String accountName,
       String typeName) throws CassiusNSException;
public void deleteObjectType( String accountName,
       String typeName) throws CassiusNSException;
public String[] listObjectTypeEvents( String accountName,
       String typeName) throws CassiusNSException;

// Account listener management
public void addAccountEventsListener(ICassiusAccountListener
```

```
al, String account) throws CassiusNSException;
    public void removeAccountEventsListener (
            ICassiusAccountListener al, String account)
            throws CassiusNSException;
    public void addModelEventsListener(ICassiusModelListener ml)
            throws CassiusNSException;
    public void removeModelEventsListener (
            ICassiusModelListener ml) throws CassiusNSException;

    // Publish/subscribe API
    public void publish(ICassiusEvent event, String accountName)
            throws CassiusNSException;
    public void subscribe(ICassiusSubscriberInterface si,
            ICassiusSubscription subscription)
            throws CassiusNSException;
    public void unsubscribe ( ICassiusSubscriberInterface si,
            ICassiusSubscription subscription)
            throws CassiusNSException;
    public void unsubscribe (ICassiusSubscriberInterface si)
            throws CassiusNSException;

    // Pull notification
    public void pullNotifications (
            ICassiusSubscriberInterface si,
            ICassiusSubscription subscription, boolean delete)
            throws CassiusNSException;
    public void pullNotifications (
            ICassiusSubscriberInterface si,
            ICassiusSubscription subscription,
            long since, boolean delete)
            throws CassiusNSException;
    public void clearNotifications( String accountName,
            long olderThan) throws CassiusNSException;
}
```

4.3.2 EDEM

Expectation Driven Event Monitoring (or EDEM) (Hilbert and Redmiles 1998) is an approach to software usability testing based on the concept of expectations (common sequences of steps that represent user interface interactions). Through the direct monitoring of applications deployed to end-users computers, EDEM detects, summarizes and logs invalid or unexpected sequences of user interface events. This information is periodically sent to software developers and interface designers, thus helping in the resolution of common usability problems. The EDEM approach is illustrated in Figure 26.

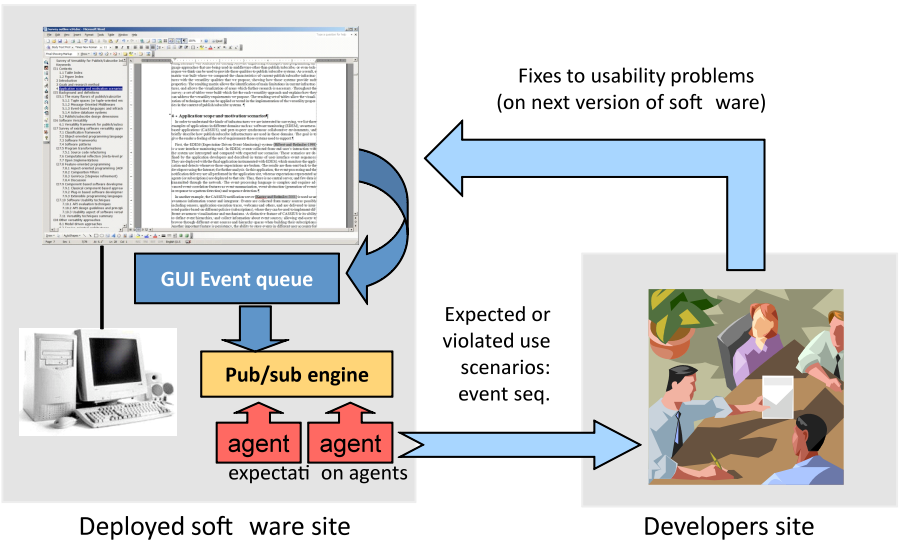


Figure 26 EDEM approach summary

EDEM relies on a core publish/subscribe component that is responsible for detecting and recording sequences of events according to a set of rules (subscriptions defined in terms of Event-Condition-Actions (or ECA rules)). From the point of view of the publish/subscribe infrastructure, EDEM requires the following features: event content-based filtering; event pattern detection supporting disjunction, conjunction and exact sequence match, and ECA rules. It also requires a way to store temporary results in the form of system properties, expressed as attribute/value tuples. Rules are special abstract data types that combine patterns, actions (e.g. event recording or counting) and a set of *begin* and *end* triggers. Triggers (or Guards) are pattern detectors based on state changes or event occurrences. They are used to control the activation and deactivation of actions within a ECA rule. EDEM event model is object-based (events are actual Java AWT events). The notification model is implemented by individual rules that can either push, summarize and/or save events for further analysis in the system state space. The routing is content-based, directed by subscription filters. Finally, the protocol model supports tuple manipulation that allows the storage, and further retrieval of event logs.

EDEM reference API is shown in Table IV. It supports both tuple manipulation for persistency and publish/subscribe based on filters, patterns and rules. Note that subscriptions are expressed as objects in the target programming language, in a way similar to that used by Siena, for example.

Table IV EDEM publish/subscribe core reference API

```
  public interface IEDEMNotificationServerAPI {

    // State manipulation operations
void setState(String key, String value);
void setState(String key, int value);
String getState(String key);
String getIntState(String key);
void removeState(String key);

    // publication and subscription primitives
void publish(IEDEMEvent event);
void subscribe(ISubscriberInterface si, IEventFilter filter);
```

```
   void subscribe(ISubscriberInterface si, IStateFilter filter);
   void subscribe(ISubscriberInterface si, IPattern condition);
   void subscribe(ISubscriberInterface si, IRule rule);
   void unsubscribe(ISubscriberInterface si);
   void unsubscribe(ISubscriberInterface si, IEventFilter filter);
   void unsubscribe(ISubscriberInterface si, IStateFilter filter);
   void unsubscribe(ISubscriberInterface si, IPattern condition);
   void unsubscribe(ISubscriberInterface si, IRule rule);
}
```

4.3.3 IMPROMPTU

IMPROMTU (DePaula, Ding et al. 2005) is an ad-hoc peer-to-peer file sharing desktop application. IMPTOMPTU allows users to share files in an ad-hoc way. It supports different types of visibility (see, read, write and persistent), and notify users of events such as file read, open, write, move and others. It is built on top of a topic-based publish/subscribe bus that connects all the peers in the network. This event bus is self configurable, i.e. it automatically locates and connects to other peers in the network, forming a virtual event bus. IMPROMPTU uses events to synchronize the user interfaces of each peer and to represent timely file manipulation notifications of changes in visibility or different read/write accesses. IMPRMPTU's architecture is shown in Figure 27.

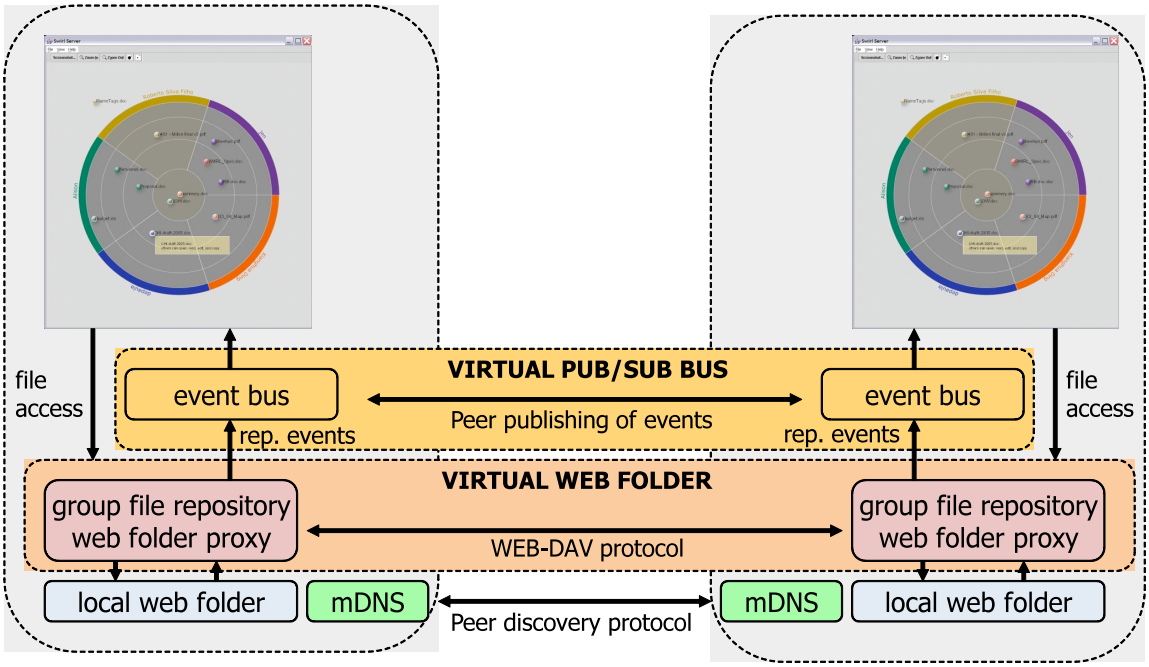


Figure 27 IMPROPTU high-level architecture

Impromptu publish/subscribe bus is topic-based. Events are record-based, representing either as *File* or *GUI* events. The protocol model supports peer location via IETF Zeroconf[3] multicast-

[3] http://www.zeroconf.org/

DNS (or mDNS), and handles the dissemination of events among all peers in the network. The publication model supports special filters that remove repeated events, within a short time interval, before they are propagated to other IMPROMPTU peers. This feature is important to reduce the traffic of events in the network. The IMPROMPTU publish/subscribe core reference API, as shown in Table V, is very simple, it supports the basic topic-based routing of events between peers.

Table V IMPROMPTU publish/subscribe infrastructure reference API

```
public interface IImpromptuNotificationServerAPI {

    public void publish (IImpromptuEvent event,
          boolean publishToPeers) throws ImpromptuNSException;
    public void subscribe (IImpromptuSubscriberInterface si,
          IImpromptuTopicSubscription subscription)
          throws ImpromptuNSException;
    public void unsubscribe (IImpromptuSubscriberInterface si,
          IImpromptuTopicSubscription subscription)
          throws ImpromptuNSException;
    public void unsubscribe (IImpromptuSubscriberInterface si)
          throws ImpromptuNSException;
}
```

Together, these three application domains and their reference APIs pose a diverse set of features that exercise every publish/subscribe design dimension of Table I. A summary of the three scenarios and their publish/subscribe infrastructure requirements are presented in Table VI.

Table VI Summary of features required by the three application domains used in our case studies

	CASSIUS	EDEM	IMPROMPTU
Event	Record-based	Objects: AWT events	Record-based
Publication	Publish to user account	none	Repeated events filter
Routing	Content-based	Content-based	Topic-based
Subscription	Content-based subscription language	Content filters, pattern matching, and rule objects	Topic filter represented as an object
Notification	Pull	push, recording, summarization	Push
Protocol	User: Account management Event source browsing	User: tuple manipulation	Infrastructure: P2P location and publishing

4.4 Metrics suite

In our study, we are interested in analyzing and comparing the versatility of different publish/subscribe infrastructures. The versatility of an infrastructure is thus defined as a combined set

of qualities including: infrastructure *maintainability* and *flexibility*, as well as the overall system *usability*, *reusability*, and *performance*. These software qualities are expressed in terms of source code attributes such as number of lines of code, McCabe's cyclomatic complexity (or CC), API size, and scattering of concerns. In our measures, we many times adopt a concern-based approach, where we analyze different aspects of the infrastructure, groping code fragments into crosscutting categories such as features, and role-based APIs.

In the following sections, we discuss the primitive metrics we adopted in the measurement of higher-level software qualities.

4.4.1 Development effort

In our study, we quantify different software qualities such as cognitive distance (4.4.2), and API usability (4.4.3) in terms of a common metric called development effort.

The development effort is measured as the product of two well known metrics: the number of lines of code (or LOC), and McCabe's cyclomatic complexity (or CC) (McCabe 1976). This product is used to balance code complexity and its length, working as an indirect indication of the developer's effort.

4.4.2 Reusability: Cognitive distance

According to Krueger (Krueger 1992), the reusability of an infrastructure can be measured by the concept of *cognitive distance*. Krueger defines cognitive distance as the work necessary to reuse the infrastructure in a different context. This work requires the successive application of the operations described in 2.1.2.

As illustrated in Figure 28, the cognitive distance represents the effort of *adapting*, *extending*, *configuring* and *composing* the features provided by each infrastructure in the implementation of the reference APIs. In our case studies, we measure the cognitive distance of each infrastructure by calculating the development effort of each reference API when reusing each infrastructure.

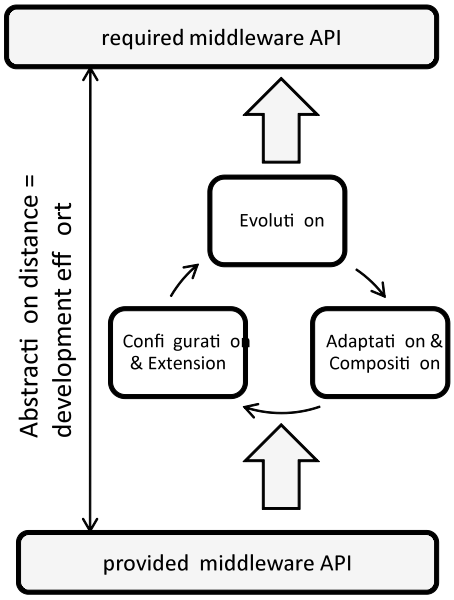


Figure 28 Cognitive distance as the total development effort to reuse a provided middleware API in the development of an (ideal) required application-specific API

4.4.3 Usability: API size and task complexity

In order to analyze and compare the usability of the selected infrastructures, we employ quantitative metrics such as API size and the development effort of performing common tasks such as publishing or subscribing to events.

We define API size as the sum of the total number of public methods (M), fields (F), parameters (P), classes (C) and interfaces(I) of each infrastructure; whereas the API usability is measured as the development effort (LOC*CC) of the most common API use cases such as publication of event, notification and subscription.

4.4.4 Modularity and scattering of concerns

Modularity and scattering of concerns are two important software attributes that correlate with software *maintainability* and *flexibility* (Li and Henry 1993; Sullivan, Griswold et al. 2001). In our analysis, we use two different metrics: CDC (Concern Diffusion over Components) originally proposed by (Garcia, Sant'Anna et al. 2005); and DOSC (Degree of Scattering over Components) originally discussed at (M. Eaddy and Murphy 2007), DOSC is a measure between 0 and 1. High DOSC (close to 1) indicates that the implementation of a concern is highly scattered (less modular); whereas low DOSC (close to 0) indicates that the concern is localized in one class (more modular).

Chapter 5. Case Studies Implementation & Data Collection

This chapter describes the implementation of the three reference APIs described in section 4.3, highlighting their main components and architecture. In doing so, this chapter's goal is to make explicit the major commonalities and differences between each publish/subscribe approach when supporting different application domain requirements. We also discuss the measures adopted in order to obtain a fair comparison between the case studies, including the data collection procedure, discussing some examples of measurements we performed.

5.1 Case study design & implementation challenges

When comparing heterogeneous software infrastructures, developed according to different original goals, it is important to strive for a fair evaluation process. Different measures were adopted in the design & implementation of our case studies to increase equitable comparison between the different approaches.

- First, we chose to implement the case studies ourselves to eliminate the variance that may come by the use of different developers, at different levels of expertise.
- Second, we adopted best of breed design practices in all implementations (Gamma, Helm et al. 1995), applying them consistently throughout the case studies.
- Third, we modularized common features into components that were reused throughout the different implementations, this approach minimizes the variance between implementations. We also adopted the same algorithms used by the original applications we supported.
- Fourth, we aligned the different implementations to follow the same task structure. This facilitates our data collection and analysis, and guarantees equality in the implementations of each case study.
- Fifth, we strived, as much as possible, to base our implementations on the features already provided by each infrastructure, thus avoiding the unnecessary implementation of features that are natively supported by each system.
- Sixth, we compared the infrastructures based on the same set of concerns, originated from the middleware (Emmerich 2000) and software engineering literature (Table I)
- Seventh, we conducted our performance benchmarks in the same set of machines (one client and one server), connected via a 100 Mbps Local Area Network, thus providing a constant environment.

These strategies collectively increase the likelihood that code style, application-specific algorithms and overall implementation approaches were similar throughout our experiments, at the same time that allows the strengths of each infrastructure to be reused as much as possible.

In the next sections, we describe, in more detail, the steps undertaken in our evaluation.

5.2 EDEM case study implementation

As discussed in section 4.3, an ideal publish/subscribe infrastructure, supporting EDEM requirements need to provide the following features: event content-based filtering; event pattern detection supporting disjunction, conjunction and exact sequence match; state change filters; and rules supporting guards and actions. EDEM also requires the ability to store state properties and variable values.

As shown in Figure 29, in our case study, we produced four distinct implementations for the EDEM reference API. Note that while JavaSpaces, CORBA-NS and Siena are reused as black boxes, YANCEES is reused as a grey box, i.e. it is extended from the inside, on its different variation points. As a consequence, when reusing black box infrastructures, developers need to provide extra threading and distribution capability to the infrastructure before wrapping the integrated components beneath a common server API. This is not the case with YANCEES that, for being extended in the server side, supports the reuse of its threading and distribution features.

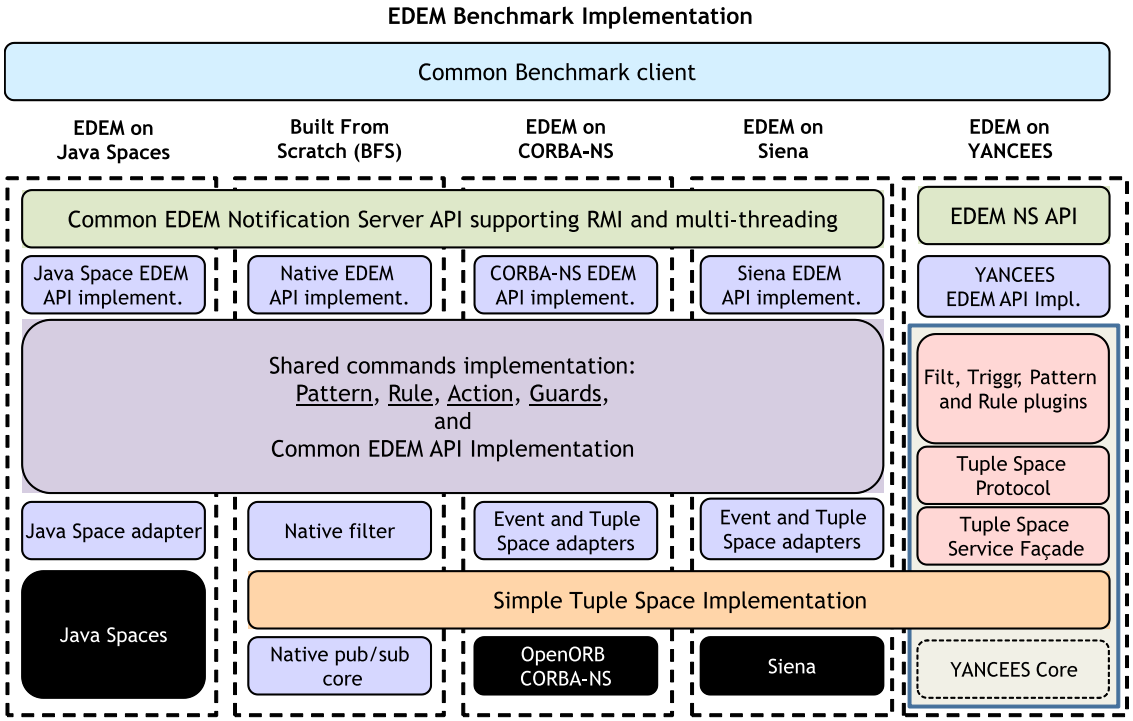


Figure 29 EDEM case study main components

In Figure 29, boxes represent components; dashed vertical boxes represent individual implementations used in our tests, for example: EDEM API implemented on top of JavaSpaces, CORBA-NS, Siena, etc. Solid rounded boxes represent major components integrated in the production of each implementation. For example, EDEM on Siena combines: the tuple space component, the Siena notification server, server-specific event and tuple space adapters (that handle the conversion between Siena and EDEM events, notifications, triggers and subscriptions), and a common set of Pattern, Rule, Action and Guard implementations. Components that are shared between two or more different implementations crosscut different dashed boxes.

5.3 IMPROMPTU case study implementation

As discussed in section 4.3.3, an ideal publish/subscribe infrastructure supporting IMPROMPTU requirements must provide a very fast publish/subscribe core that is able to automatically find other peers in the network, thus creating a topic-based routing bus. This peer-to-peer feature requires each router in the network to interact with the multicast DNS protocol, and to multicast events to each other. The peer-to-peer event bus from IMPROMPTU operates over record-based messages representing repository and GUI events. It is also responsible for removing repeated events, during publication, thus reducing the traffic in the network.

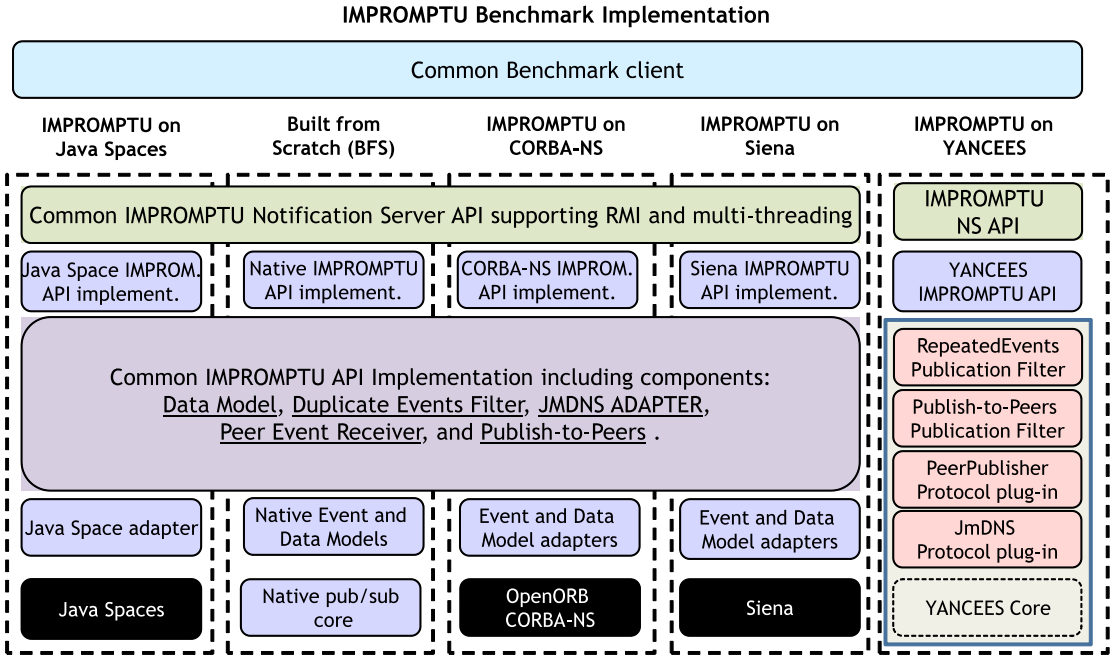


Figure 30 IMPROMPTU case study main components

These features are very unique to IMPROMTU. as a consequence, in the IMPROMPTU case study implementation relied on different components as shown in Figure 30. The *RepeatedEventsFilter* component removes repeated events as they are published; the Publish-to-Peers component intercepts events as they are published, routing them to all known peers in the network. The propagation of events between peer routers is mediated by the *PeerPublisher* component that provides a back door in each router, allowing them to receive events from other peers; the JmDNS component interacts with the mDNS protocol creating a model of all known peers in the network.

5.4 CASSIUS case study implementation

CASSIUS reference API supports event source browsing protocol, event persistence with pull notification, and a subscription language that allows sequence detection and content-based filtering. In particular, as opposed to IMPROMPTU and EDEM, CASSIUS subscriptions are not expressed as objects in the target programming language, but as text-based expressions. This requires extra parsing, represented as *CASSIUS Subscription Parser* components in Figure 31. Since YANCEES supports automatic subscription parsing, this feature comes "for free" in the infrastructure.

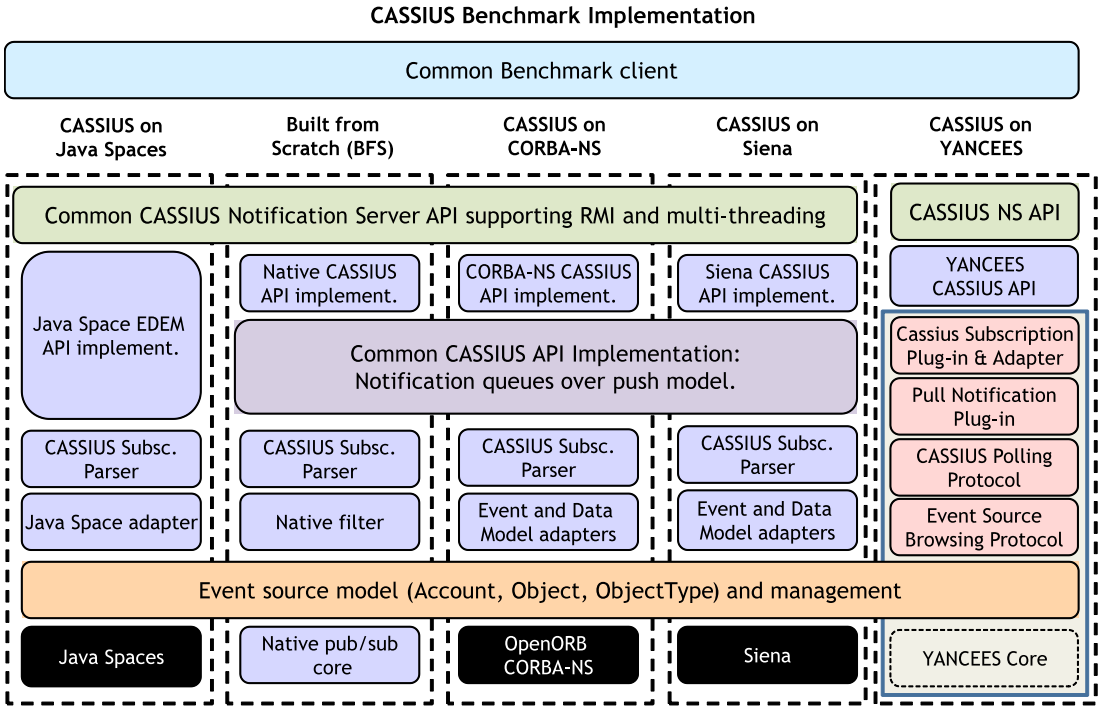


Figure 31 CASSIUS case study main components

The pull notification feature of CASSIUS is similar to that provide by an e-mail server. Notifications are stored in individual accounts defined by the different subscriptions that originated them. As shown in Figure 31, this feature is implemented by queue components (notification queues over push model) that store these notifications for further retrieval. This component is shared by CORBA-NS and Siena subscriptions. Note that CORBA-NS pull notification model was incompatible with the one used by CASSIUS. It does not provide persistency of events based on individual subscriptions.

Since JavaSpaces already supports persistency, this feature is implemented using the *read(), notify()* and *take()* commands of the tuple space in the JavaSpaces adapter. YANCEES supports CASSIUS features through a set of protocol plug-ins and the pull notification plug-in that implements the queue of notifications for each subscription.

Finally, it is worth noting that, for our case studies, YANCEES was extended with a set of domain-specific plug-ins. These plug-ins implement content-based and topic-based filtering, supporting a subscription language that is feature-compatible to Siena.

5.5 Data collection

After the case studies implementation, we collected different measurements as discussed in section 4.4. The data collection was performed in a semi-automatic fashion, as shown in Figure 32. Metrics such as Lines of Code (or LOC), and McCabe cyclomatic complexity (or CC) were collected with the help of Eclipse Metrics plug-in[4]. Whereas metrics such a CDC and DOSC were

[4] Eclipse Metrics Plugin: http://metrics.sourceforge.net/

collected using *ConcrenTagger*[5], a tool, based on *ConcernMapper* (Robillard and Weigand-Warr 2005), that allows the grouping of code fragments (methods, interfaces, and classes) into concerns. These results were combined and analyzed with the help of spreadsheets and charts.

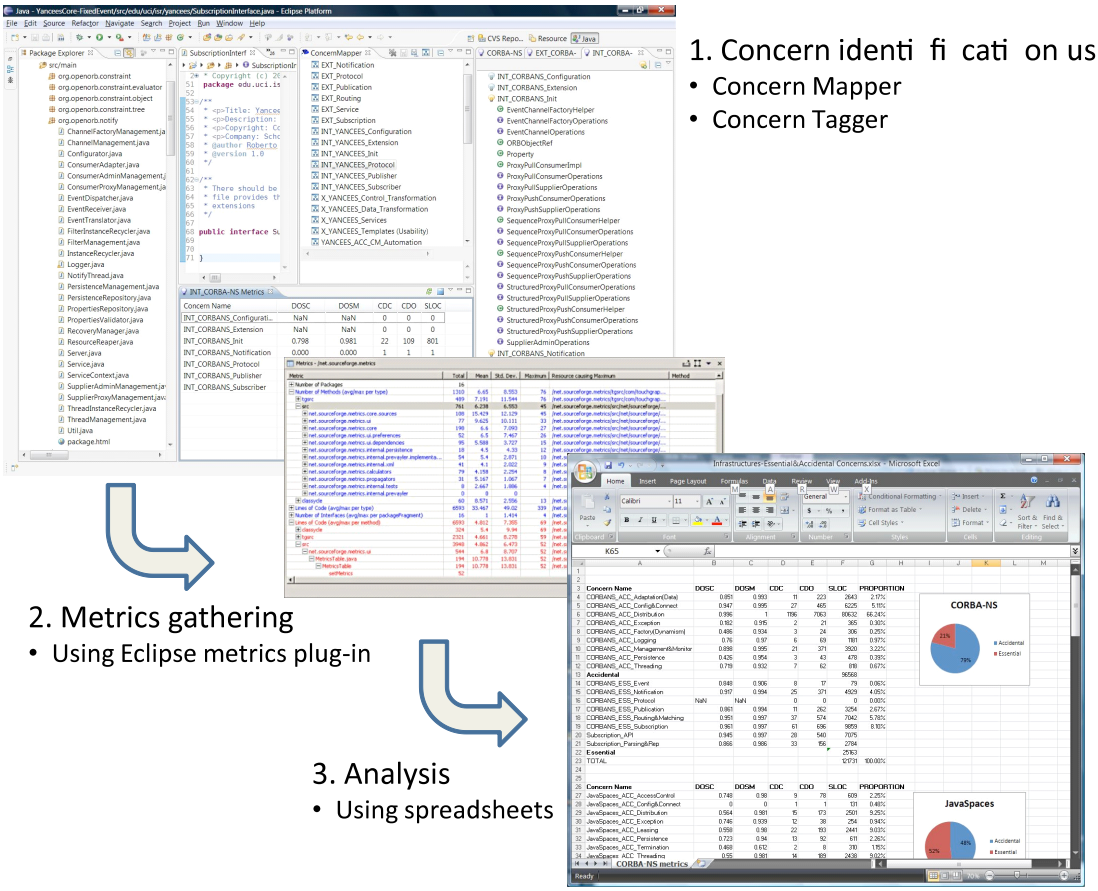


Figure 32 Metrics gathering and analysis process

The source code of the case studies ,the *ConcernTagger* and *ConcernMapper* databases, and the spreadsheets used in our analysis are available in the website: *http://www.isr.uci.edu/projects/yancees/tradeoffs*

5.5.1 Concern tagging criteria

When comparing different infrastructures through different metrics, the devil resides in the details. Divergences in the measurements criteria may favor one infrastructure or case study over another, invalidating the results. In order to achieve a fair comparison between the different infrastructures, the concern tagging procedure needs to follow a common criteria. In our evaluation, concerns are either functional requirements, as publish/subscribe domain and middleware specific features; and non-functional requirements, such as versatility software qualities discussed in Chapter 2. Table VII summarizes the sets of concerns we measured in the selected infrastructures and their implementations.

[5] http://sourceforge.net/projects/concerntagger/

Table VII List of major publish/subscribe concerns used as tagging criteria

Functional requirements			Non-functional requirements
Domain-specific concerns	**Middleware concerns**	**Optional concerns**	**Versatility concerns**
• Event representation • Subscription representation • Publication • Routing (order and content) • Notification policies • Protocols • Parsing (subsc. languages)	• Multi-Threading • Distribution • Logging • Connection	• Transactions (JavaSpaces) • Leasing (JavaSpaces) • Access control (JavaSpaces) • Session persistence (CORBA-NS) • Management (CORBA-NS)	• Usability • Maintainability • Reusability • Extensibility • Configurability • Performance

Whereas domain and middleware concerns are functional requirements, being easily identifiable in the code; versatility concerns are non-functional requirements, that indirectly depend on different software characteristics. As such, they are usually measured indirectly, through case studies and benchmarks. For example, usability is measured in terms of the API size and specific tasks development effort; flexibility is measured as the change impact of domain-specific features evolution; and reusability is given by the total development effort of each case study. The full set of concerns and the measures we adopted in our case studies are further described in Table VIII.

Table VIII Concern tagging criteria and some of their examples

		CONCERN	DESCRIPTION	EXAMPLES
Functional requirements	**Domain specific**	Event	Classes and interfaces that are used to represent events	• *Notification* class in Siena; • *YanceesEvent* class in YANCEES • *Entry* interface in JavaSpaces • *StructuredEvent, Property* and *Any* classes in CORBA-NS
		Subscription	Classes interfaces and API calls used to represent subscription expressions, as well as subscribe API calls	• *Filter, Pattern, Op, AttributeConstraint* classes plus *subscribe()* commands in Siena; • *GenericSubscription* class, that wraps XML subscriptions, plus *subscribe()* commands, and internal subscription parsing classes in YANCEES. • *Entry* interface plus *read()* and *notify()* commands in JavaSpaces • Classes representing subscription commands, and *filter* manipulation method calls in CORBA-NS proxies.
		Publication	Classes, interfaces	• *publish()* command in Siena; • *publish()* command with *GenericFilter*, *Fil-*

			and API calls that support the publication process. Includes publication filters	*terInterface* and other filter management classes. • *write()* command in JavaSpaces • CORBA-NS allows the definition of publication filters. *push()* and filter manipulation commands in CORBA-NS proxies.
		Routing	Classes, interfaces and APIs that implement the matching of subscriptions to events	• *Patternmatcher* and *Posets* in Siena; • *EventDispatcher* and auxiliary interfaces in YANCEES. • *EntryHandle* and internal queues in JavaSpaces • *EventQueueFilter* and auxiliary interfaces in CORBA-NS.
		Notification	Classes, Interfaces and APIs that handle the post-processing of events that were matching, including filtering and delivery of events to listeners	• *notify()* command in the *PatternMatcher class* and *Notifiable* interface in Siena. • *NotificationPlugin, NotificationManager* classes and *SubscriberInterface* in YANCEES. • *Watcher* and *Notifier* classes in JavaSpaces • *Puller, Pusher, Orderer and Queue Dispatcher and Receiver* classes in CORBA-NS.
		Protocol	Classes interfaces and API calls that handle user and infrastructure protocols	• *advertise()* commands in Siena. • *ProtocolManager, ProtocolFaçade* and other protocol implementation classes in YANCEES. • *contents()* command and its auxiliary classes in JavaSpaces • Management and monitoring commands in CORBA-NS API.
	Middleware	Distribution	Classes and interfaces that support communication	• Java RMI Remote interfaces and classes in JavaSpaces (*OutriggerServerImpl*) and YANCEES (*YanceesRMIClient*) and auxiliary classes • *SEMP* and auxiliary classes that interact with Sockets API on Siena • ORB IDL descriptions of its interface (not counting the automatically generated code)
		Threading	Classes and methods that support multi-threading and concurrency	• Threads inside *HierarchicalDispatcher* threading methods in Siena • Threads inside *RemoteYanceesImplementation* class in YANCEES • *OutriggerServerImpl, Wrrapper and OperationJournal* in JavaSpaces • *Puller*, *Pusher* and *EventQueue* classes in CORBA-NS

		Logging	Classes and methods that implement logging	• *Logging and Monitor* classes in Siena • YANCEES does not provide logging in its core. Instead, it logs few events to the standard output and delegates logging to plugin developers. • JavaSpaces' logging is provided by the JINI platform in which it is based • *Logger* class and log methods in different classes from CORBA-NS
	Optional	Transaction, Leasing, Access Control, Management, Session persistence, etc	Classes and interfaces that implement these features	These concerns are specific to each infrastructure, and are tagged accordingly as shown in Table VII.
Non-functional requirements		Usability	Public API size and analysis of complexity of the most common pub/sub tasks	Individual task analysis of common publish/subscribe commands plus • API size of *HierarchicalDispather, Filter, Pattern, Op, AttributeConstraint, and Notification* classes in Siena • API size of *YanceesClient, SubscriberInterface and YanceesEvent* classes in YANCEES • JavaSpaces' logging is provided by the JINI platform • *Logger* class and log methods in different classes in CORBA-NS
		Maintainability	The average modularity of basic publish/subscribe concerns	Calculated as the average modularity of each infrastructure based on the individual modularity of domain-specific concerns (measured in DOSC).
		Reusability	Total development effort to reuse an infrastructure for each case study	Total development effort (LOC*CC), for each infrastructure, for the three case studies we propose.
		Reusability (adaptation)	Costs of converting from and application-specific to infrastructure-specific data structures	The costs of converting EDEM, CASSIUS and IMPROPTU event and subscription formats into the native formats of CORBA-NS, Siena, JavaSpaces and YANCEES
		Extensibility	Number of lines of code that explicitly support extension towards the domain-specific features	YANCEES is the only infrastructure that matches this criteria. It provides extensibility towards pub/sub design dimensions, in the form of variation points such as: *PluginInterface*, *EventDispatcherInterface*, *FilterInterface* and others.

	Configurability	Classes, methods, libraries and files that allow the selection between different domain-specific features	• Siena supports the configuration of the topology of its network of routers, but provides no configuration of its provided features. • In YANCEES, domain-specific configurability is provided by its *ArchitectureManager*, *SubscriptionManager*, *PluginRegistry* and additional configuration and yancees.property files • JavaSpaces supports the configuration of its entry repository but there is no domain-specific configurability. • Factory methods, QoS properties, and builders support the creation of application-specific event channels in CORBA-NS
	Flexibility	Change impact of modifying classes and interfaces that implement domain-specific concerns	For example, for the event representation best case scenario: • *Notification, AttributeValue, AttributeConstraint* and internal classes that perform event matching classes in Siena • *GenericEvent* and *YanceesEvent* classes in YANCEES (plug-ins may need to change) • Different internal classes that, through reflection, manipulate *Entry* classes in JavaSpaces • Different event manipulation classes and methods in different CORBA-NS proxies

Chapter 6. Study Results

After representing each application domain requirement in terms of three reference APIs discussed in section 4.3, we implemented these APIs reusing the selected publish/subscribe infrastructures of section 4.2, and collected different measures as described in section 4.4 and 5.5. We also implemented each API from scratch, in order to better understand the implicit development costs of each case study.

In this chapter, we present the measurements of our case studies according to two perspectives: infrastructure developers' and infrastructure users'. Whenever possible, we also discuss the root causes of the measurements we obtained in order to better explain some of the results we present.

6.1 Infrastructure developers' perspective

From the point of view of the infrastructure developers, the different design decisions adopted in the construction of the publish/subscribe systems have positive as well as negative impacts to important software qualities such as maintainability and flexibility. They also introduce additional types of concerns (or features) to the infrastructure design. In this section, we discuss the details of the analysis of these main concerns, analyzing their maintainability and flexibility.

6.1.1 Publish/subscribe main development concerns

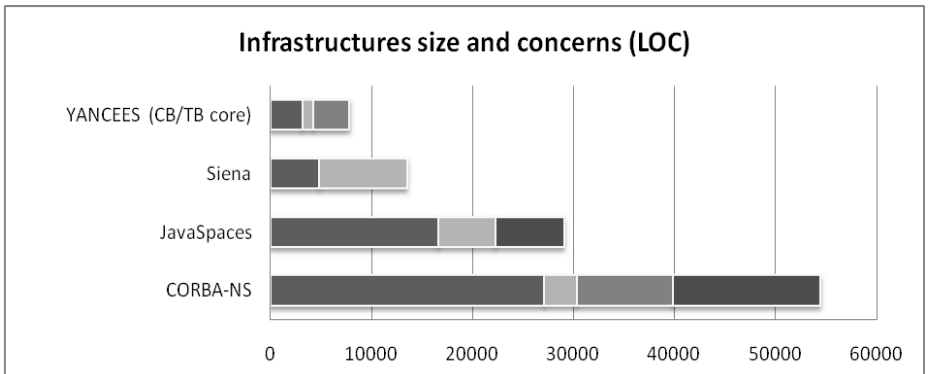
Even though apparently simple, publish/subscribe infrastructures can become very complex pieces of software, supporting an increasing number of features driven by the need to support application-specific requirements, network protocols, as well as the very generality or flexibility characteristics of each infrastructure. In order to understand the impact of these concerns in the software complexity, we first performed a concern-based analysis (Robillard and Murphy 2007) of each one of the selected infrastructures. This analysis was performed by the direction inspection of the infrastructures code. In this inspection, we categorized the infrastructure's code according to the concerns described in Table VIII of section 5.5.

6.1.2 Quantifying publish/subscribe main development concerns

The different versatility approaches analyzed in our case studies support heterogeneous sets of the concerns summarized in Table VIII. By measuring the distribution of concerns in the selected infrastructures, we obtained the chart shown in Figure 33, which shows the total size of each infrastructure based on the concerns of Table VIII. Likewise, Figure 34 shows the proportion of these concerns with respect to the total infrastructures sizes.

Note that, from all the available versatility concerns of Table VIII, Figure 33 and Figure 34 only represent *extensibility* and *configurability*. These two versatility concerns can be measured by the number of variation points and code devoted to configuration management in the code, allowing them to be quantified in terms of infrastructure' lines of code. The other versatility con-

cerns as: *reusability*, *usability*, *maintainability* and *performance*, are not directly correlated to LOC alone. They require specific measurements, and will be analyzed separately.

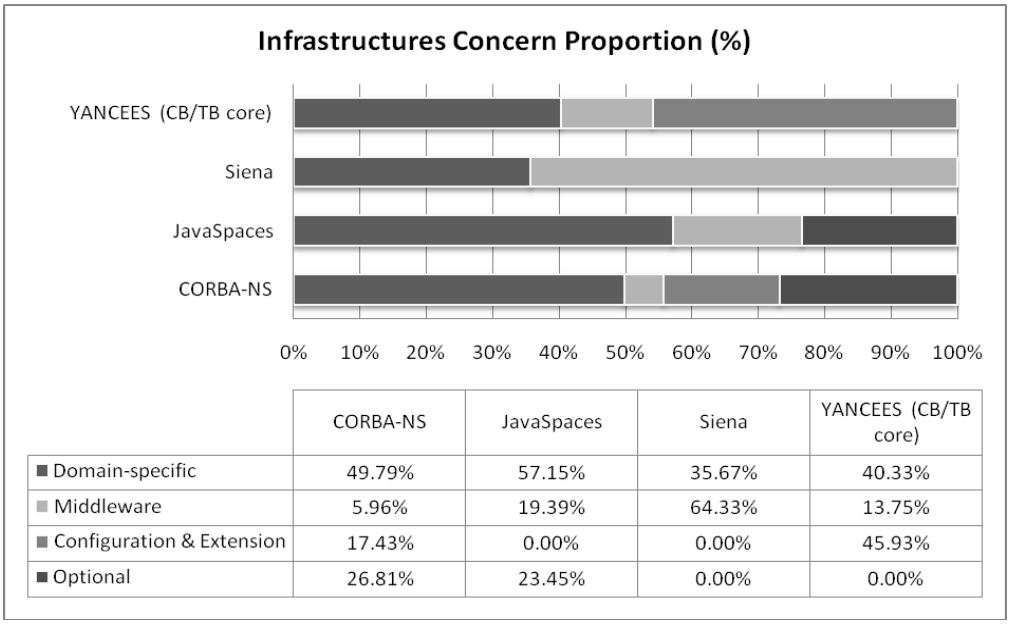


	CORBA-NS	JavaSpaces	Siena	YANCEES (CB/TB core)
Domain-specific	27095	16604	4812	3148
Middleware	3243	5634	8678	1073
Configuration & Extension	9487	0	0	3585
Optional	14589	6814	0	0

Figure 33 Infrastructures size by concerns

As shown in Figure 33, the amount of domain-specific, middleware, configuration & extension, and optional features supported by each infrastructure vary considerably. Both JavaSpaces and CORBA-NS support a considerable amount of optional features. These features can be selected by setting specific parameters on JavaSpaces API commands, or by using specialized commands in the CORBA-NS API. Both Siena and YANCEES support no optional features: while Siena strives for minimalism and generality, YANCEES supports the exact set of features required by each application domain.

Configuration and Extension concerns are only supported by YANCEES and CORBA-NS (configurability, in the case of CORBA-NS, and both extensibility and configurability in the case of YANCEES).



	CORBA-NS	JavaSpaces	Siena	YANCEES (CB/TB core)
Domain-specific	49.79%	57.15%	35.67%	40.33%
Middleware	5.96%	19.39%	64.33%	13.75%
Configuration & Extension	17.43%	0.00%	0.00%	45.93%
Optional	26.81%	23.45%	0.00%	0.00%

Figure 34 Proportional size of major infrastructure concerns

As seen on Figure 34, configuration and extension concerns represent more than 45% of the total size of YANCEES infrastructure, a consequence of its design for flexibility.

The amount of middleware concerns vary according to each approach. Compared to other infrastructures, Siena has a relatively large amount of code devoted to distribution (about 2/3 of its total size). This is a consequence of the native implementation of its communication and advertisement protocols using Sockets. This design decision, while improves the overall performance of Siena (as will be discussed in 6.2.6), results in more lengthy and less modular code (the modularity is shown in Table IX). All other infrastructures rely on remote method invocation (or RMI) mechanisms. Both YANCEES and JavaSpaces are built on top of Java RMI; whereas CORBA-NS relies on the distribution facilities of CORBA ORB (Siegel 1998), that implements RMI according to the OMG-OMA specification (Group 2003). This design decision explains their smaller amount of middleware concerns if compared to Siena.

Overall, the more features an infrastructure provides, the higher is its likelihood to support different application domain requirements. However, as will be further analyzed, this can result in larger code size, lower maintainability and poorer performance (see 6.2.6). For example, in terms of code size, both Siena and YANCEES are relatively small implementations, requiring less than 1500LOC, as seen in Figure 33. Both CORBA-NS and JavaSpaces have larger implementations, a consequence of the support for optional features.

6.1.3 Infrastructures Maintainability

The different development concerns discussed in the previous sections are not easily to modularize. Instead, they usually become scattered over many infrastructure components (Tarr, Ossher et al. 1999). The more scattered these concerns become, the more difficult it usually is to extend and maintain the infrastructure (Kim and Bae 2006).

In our use cases, we estimate the maintainability of each infrastructure, in terms of the modularity of their major concerns, as previously shown in Table VIII. We measure modularity using the DOSC metric. The results of this concern-based measure are shown in Table IX.

Table IX Infrastructure Modularity per concerns (Degree of Scattering of Concerns)

Infrastructure	Domain-specific	Middleware	Configuration & Ext.	Optional
CORBA-NS	0.93	0.95	0.87	0.94
JavaSpaces	0.66	0.59	N/A	0.71
Siena	0.66	0.84	N/A	N/A
YANCEES	0.68	0.75	0.76	N/A

As seen on Table IX, JavaSpaces and Siena present the highest modularity (lowest DOSC) with respect to publish/subscribe domain-specific concerns, being closely followed by YANCEES. Overall, YANCEES core modularity is jeopardized by the complexity of its configuration and extension concerns; whereas Siena's modular design is jeopardized by its middleware concern implementation. For example, in YANCEES, different extension points exist, one for each design dimension, which scatters the domain-specific concerns throughout individual interfaces and abstract classes; whereas in Siena, the different subscription commands need to be mapped to protocol primitives in the communication protocol layer, which scatters this concern throughout different abstraction layers and classes.

When considering the average modularity of each infrastructure, including the core plug-ins used to configure YANCEES with a Siena-compatible set of features, we obtain the chart of Figure 35 as follows.

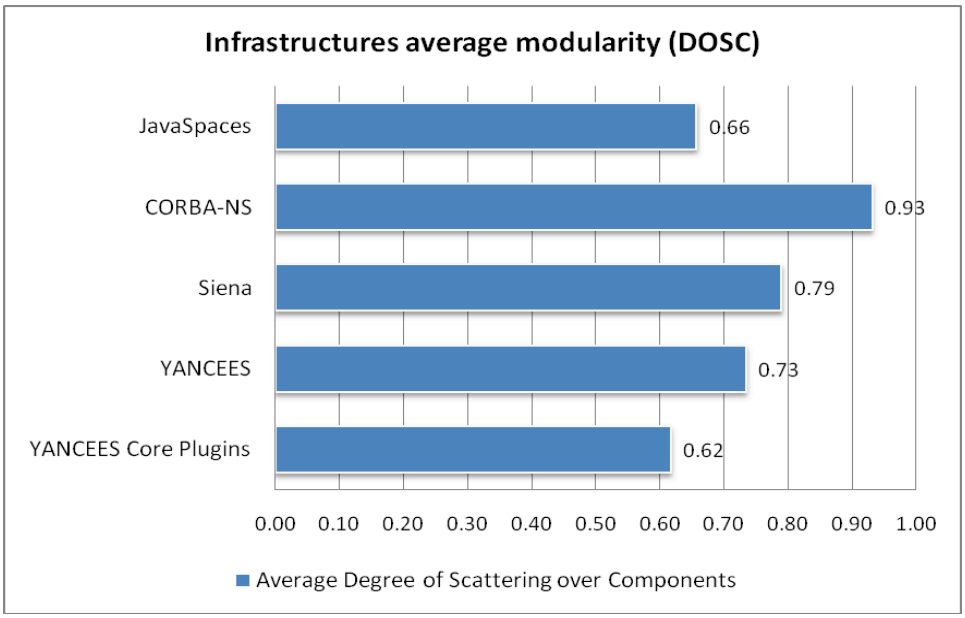


Figure 35 Average infrastructures modularity

As shown in Figure 35, among the analyzed infrastructures, CORBA-NS presents the highest average DOSC, being the least modular. This is a consequence of its large set of features, the need for configuration mechanisms, and the way features are decomposed i.e., CORBA-NS is implemented in the form of proxies and methods that provide different implementations, for each variant feature in the system. In particular, different methods and proxies are defined for each event variant (Structured, Property and Any), notification policy and role (publisher or subscriber). For example, the support for *Structured Events* is scattered throughout producer and consumer proxies, pushers and pullers, as well as different event queues in its CORBA-NS core. The result is increased maintenance costs of particular features.

Even though YANCEES was not the most modular infrastructure (loosing for JavaSpaces), the extensions developed using YANCEES are relatively modular (see YANCEES Core Plugins in Figure 35). This comes as a consequence of the design for extensibility of YANCEES, which supports extensions along the main publish/subscribe dimensions.

Even though the separation between publish/subscribe main concerns is a good indicator of maintainability, it does not guarantee that the infrastructure is flexible (extensible and configurable). The reason for that, are the fundamental, configuration-specific and technological dependencies discussed in section 2.2, which defines both data and control dependencies. Hence, we also analyzed the flexibility of each infrastructure, using a more direct change impact analysis as follows.

6.1.4 Flexibility (feature change impact)

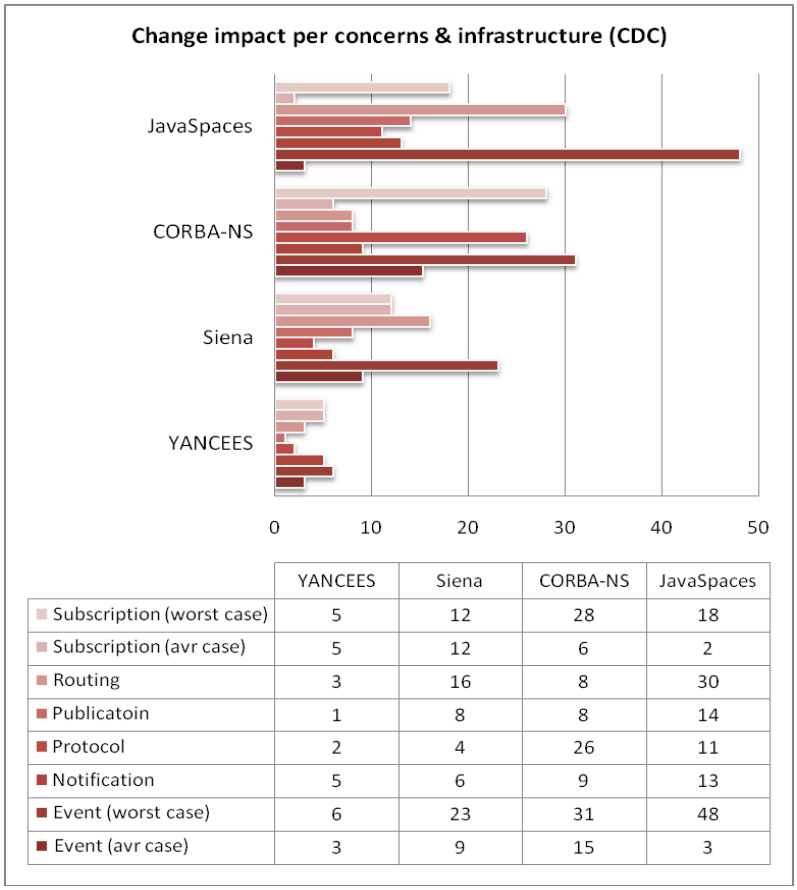
Parnas defines flexibility as the ability of software to be extended and contracted to fulfill different purposes (Parnas 1994). As noted by, Eden & Mens (Eden and Mens 2006), flexibility is not an absolute software quality. Instead, software is more or less robust toward particular (usually planned) classes of changes. In particular, Eden & Mens define flexibility as the complexity of the task required to adapt or evolve a system from an initial stage (or implementation), to a new implementation stage, that satisfies a set of new requirements triggered by shifts in the problem domain. For such, an *evolution step*, or adjustment, needs to be applied to software. The flexibility of software with respect to that *evolution step* is defined as the computational complexity of the meta-program that adjusts the original code to meet a new requirement. In other words, it is a direct function of the impact of an evolution step in the software main components.

In this section, we investigate the ability of each infrastructure to support changes along the main variability dimensions of publish/subscribe infrastructures (described in Table I). For such, we measure the change impact of modifying or adding concerns to each infrastructure using the CDC metric. For this particular case, this metric represents the number of classes potentially affected (changed, added, removed) by the implementation of a feature in that particular publish/subscribe design dimension.

When measuring a particular CDC for a feature, we first represent these concerns as possibly overlapping sets of classes and interfaces, using *ConcernTagger*. The identification of each concern was performed manually, by inspecting the code and tagging the methods that belong to each concern. We also utilized references to important objects, for example event representations, to identify classes and interfaces that would be potentially impacted by changes in key data representations such as event format and subscriptions.

The result of our analysis is shown in Figure 36, which calculates the impact, in terms of number of classes and interfaces affected, in the worst case scenarios, when a new feature is added, removed or modified in each major publish/subscribe variability dimension. In particular, for the subscription and event models, we also calculate the average scenarios, which are less severe classes of changes, made possible by the adoption of generalized data representations.

In Figure 36, the subscription change impact worst case scenario is calculated by counting all the classes and references that depend on the standard event or subscription representations; whereas the best case scenario assumes changes in the event and subscription representation keeping existing generalized interfaces fixed.



	YANCEES	Siena	CORBA-NS	JavaSpaces
Subscription (worst case)	5	12	28	18
Subscription (avr case)	5	12	6	2
Routing	3	16	8	30
Publicatoin	1	8	8	14
Protocol	2	4	26	11
Notification	5	6	9	13
Event (worst case)	6	23	31	48
Event (avr case)	3	9	15	3

Figure 36 Change impact analysis per publish/subscribe concern (measured in terms of concern diffusion over components) for each infrastructure

One of the most prominent features of Figure 36 is the impact of event change in the infrastructures, followed by changes in the subscription format and in the routing strategies. These measures reveal the impact of fundamental dependencies in the publish/subscribe domain, which makes the development of flexible infrastructures challenging. In particular, the dependencies between event representation, routing strategies and subscription filters are the most important ones in the publish/subscribe domain. They define the routing algorithm supported by the infrastructure and restrict the independent evolution of each one of these dimensions as discussed in section 2.2.2).

6.1.5 Discussion: the role of generalization, variation and configuration management in the reduction of change impacts

In order to improve the infrastructure flexibility, preventing changes driven by fundamental problem dependencies, different strategies have been adopted in the development of the versatile

infrastructures we analyzed. These are: generalization (Siena, JavaSpaces, YANCEES), variation (CORBA-NS) and configuration management (YANCEES).

For example, even though JavaSpaces event model has a high change impact, it uses generalized tuples as both event and subscription representation. It then relies on reflection to match tuples to templates (anti-tuples). This generalization permits variations in the event representation (in terms of its set of attributes), without impacting the existing routing and subscription algorithms, thus reducing the change impact in the average case (see Figure 36).

In another example, both Siena and YANCEES events are represented as attribute/value pairs that support both object and record-based events. Hence, instead of modifying the event format to represent specific Objects or Records, users can convert and encode these external representations as attribute/value pairs, rebuilding the original representations once notifications are routed, thus preventing changes in the implementation of these infrastructures.

Finally, CORBA-NS supports multiple events through variation: by simultaneously providing different event representations. However, in order to prevent the proliferation of routers, supporting same algorithms for different event formats, CORBA-NS adopts a common internal event representation. Events of different types are automatically converted into *StructuredEvents* before being routed by the infrastructure. This approach, while effective, adds more complexity to the infrastructure implementation.

As shown in Figure 36, YANCEES presents very low change impact for the main variability dimensions. This comes from the use of generalized algorithms and interfaces, and the fact that YANCEES is a component framework, a partial implementation which functionality is provided by plug-ins.

This approach, however, has some disadvantages: it implicitly delegates the management of fundamental and configuration-specific dependencies to plug-in developers. For example, plug-ins in different dimensions must be developed to be compatible with certain kinds of events and timing constraints. Hence, unannounced changes in the event representation will result in incompatibilities on existing plug-ins. Moreover, it is usually the case that plug-ins depend on other plug-ins through configuration-specific dependencies, for example, a pattern detection is dependent on filters and routing algorithms guarantees. Chances in these parameters may result in different timing or order of events, invalidating the pattern detection algorithm (see Figure 10).

Hence, in order to address these issues, and improve the usability of the infrastructure, a great amount of effort is spent in configuration management (see *Config&Extension* concerns in Figure 33), that enforces compatibility relations based on user-provided information in the plug-ins manifest. Moreover, as previously discussed, the event representation was fixed, and a general attribute/value pair representation was adopted (see Figure 4).

Finally, when comparing the routing strategy change impact, both YANCEES and CORBA-NS have relatively lower change impacts than JavaSpaces and Siena. This comes from the fact that both infrastructures support the simultaneous use of different routing strategies. These strategies are modularized into components that are selected at runtime. For example, CORBA-NS provide different event queues, supporting QoS such as LIFO, FIFO; as well as event formats as *Any* and *StructuredEvent*. YANCEES supports the installation of different routers, that can be developed to support different event formats. In particular, the ability to support different routing strategies support fast routing algorithms that match specific event representations and subscription commands. Hence, variation represents another strategy to tame the effects of change, while improving performance.

6.2 Application developers' perspective

In the previous section, we discussed the software qualities that are important from the perspective of infrastructure developers. In this section, we turn our attention to application developers.

From the application developers' perspective (infrastructure users), it is important to understand the *usability*, *reusability* and *performance* characteristics of each versatility approach. These software qualities are analyzed in the following sections.

6.2.1 API Usability

The usability of an infrastructure is a direct function of its Application Programming Interface (or API). The development of good APIs is not a trivial task and faces fundamental challenges. The first challenge is the problem discussed by Kiczales (Kiczales 1995), the trade-off between abstraction level and fitness to the problem. The higher level and closer to the application domain concerns an API becomes, the easier it is the development of software. However, in the design of specialized APIs, different decisions, simplifications and assumptions are made, which usually lead to the next problem: abstraction mismatch. Abstraction mismatch (Garlan, Allen et al. 1995) occurs when APIs, supporting features that apparently fit the problem, have subtle semantic differences that hinder their direct use by the application. The result is the need for extra adaptation, and extension, which leads to performance and development costs. In worst case scenarios, these differences can represent complete mismatches (Kiczales 1995).

In our studies, we collected three major measures: API Size, API separation of concerns, and the complexity of common application tasks when using the API. For such, we collect the data based on different use cases (or tasks).

6.2.1.1 Task-based analysis

Publish/subscribe infrastructures support different users fulfilling different roles, and performing different tasks. As a consequence, a publish/subscribe infrastructure API can be analyzed in terms of the most common use case scenarios it supports. For example, Siena's API is composed of different classes such as: *HierarchialDispatchrer*, which implements the main publish/subscribe commands, and the objects that represent events (*Notification*), subscriptions (*Filter*, *Pattern*, *Constraint*, *Op*, etc.) and listeners (*Notifiable*). Every publish/subscribe API must support two major tasks: the publication and subscription/notification of events. The publication API size can be measured by counting the number of methods and parameters that deal with the publication of events. In the Siena example, it includes the *publish()* command and the *Notification*, *AttValue* interfaces, that are used to represent the events being published. Likewise, subscriptions tasks are supported by a set of *subscribe()* and *unsubscribe()* commands, together with the *Filter*, *Pattern*, *Constraint* and *Op* parameters they require, and the *Notifiable* interface, that also must be implemented by subscribers in order to receive notifications.

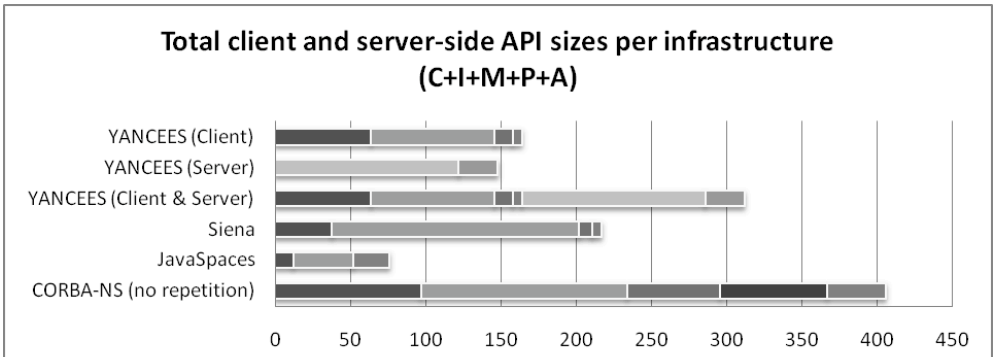
Hence, in a task-based analysis, the API concerns vary according to each typical use case scenario, and must be calculated based on the methods/objects used during these typical interactions with the infrastructure. In the following section, we analyze the selected infrastructures APIs with respect to their size, separation of concerns, and development effort according to common use cases (or tasks).

6.2.1.2 API Usability: Size

The programming effort necessary to perform the most common operations of an API, as well as the total API size are good indicators of how easy it is to learn and reuse an infrastructure. The lengthier an API is, the more difficult to learn it becomes. Likewise, the higher the effort to perform common API operations, the higher the overall application development effort (see section 7.2).

In Figure 37, we present a task-based analysis of the client API sizes. For completeness, in the case of YANCEES, we also include the server side API, YANCEES(Server), which supports extension and reflection. We also present, in a separate bar, YANCEES (Client) and YANCEES(Server) APIs tougher as YANCEES(Client &Server).

Different factors such as subscription format, the support for optional features, and the use of generalization can impact theAPI size. We further discuss the impact of different design decisions in the API sizes shown in Figure 37 as follows.



	CORBA-NS (no repetition)	JavaSpaces	Siena	YANCEES (Client & Server)	YANCEES (Server)	YANCEES (Client)
Publisher API	97	12	38	64	0	64
Subscriber API	137	40	164	82	0	82
Protocol API	62	0	9	12	0	12
Configuration API	71	0	0	0	0	0
Initialization API	39	24	6	6	0	6
Extension API	0	0	0	122	122	0
Reflection API	0	0	0	26	26	0

Figure 37 Task-based analysis of the API sizes of the infrastructures

The impact of subscription format. The way subscriptions are represented can significantly impact the size of an API. For example, from the point of view of API size and semantics, both YANCEES and Siena APIs are very similar. They both provide a simple set of *publish()* and *subscribe()* commands; and both support the same publication and subscription tasks. They, however, differ in the way subscriptions are represented. While Siena relies on objects such as *Filter* and *Pattern* to express subscription constraints, YANCEES uses subscription languages expressed in XML. As a consequence, YANCEES API does not have objects such as filters, patterns and operators. The result, as seen on Figure 37, is that YANCEES (client) has a smaller publish/subscribe API size than Siena.

The impact of optional features. Other factors also impact the API size, such as the number of optional features (either through methods or parameters). For example, CORBA-NS API size is the largest of all the analyzed infrastructures. This is a consequence of the large number of optional features it supports, and the way these options are distributed over different proxies, event representations and administration interfaces. For example, CORBA-NS publisher API has different publication methods, replicated over different proxies, one for each kind of event. A unique feature of CORBA-NS is its configuration API, that allows users to control how the proxies are connected to form customized event channels. Additionally, CORBA-NS protocol API provides management interfaces that allow the monitoring of the different components in the system. Together, these individual APIs contribute for the overall API size of CORBA-NS.

Note that differently from YANCEES, that separates client and server-side APIs, automating the configuration management, CORBA-NS requires users (application developers) to programmatically configure the infrastructure before utilizing it to their needs. This lack of automation and separation of concerns, contributes to the large size of CORBA-NS API.

The impact of generalization. JavaSpaces provides the smallest API of all infrastructures evaluated, a consequence of generalization and simplicity of design. Instead of supporting different features by means of different proxies, as in CORBA-NS, JavaSpaces API provides commands with optional parameters. In these commands, features such as leasing and transactions can be selected, by means of valid operation parameters, or ignored, by means of "don't care" values. For example, a positive leasing parameter turns on this feature, whereas negative values indicate no lease; a valid transaction ID indicates a begin or end of a transaction, whereas 0 indicates no transaction use.

Likewise, JavaSpaces does not prescribe any proprietary event or subscription format. Both subscriptions (anti-tuples) and events (tuples) are represented as regular objects, that implement a standard marker interface. As a consequence, the API size is significantly reduced.

This combined set of design decisions results in an API that supports the variability of application domains, and different optional features through a single small API. The cost to be paid for this approach, however, is the increase of the client-side code complexity, that needs to handle exceptions produced by features that are not necessarily in use. For example, in JavaSpaces, transaction and leasing exceptions must be handled for every *write(), read(), take()* and *notify()* command, even though they may not be used in a method call. This increases code cyclomatic complexity (as shown in Figure 38), which may lead to errors.

6.2.1.3 API Usability: separation of concerns

Another factor to be considered is API separation of concerns. The modularization of an API according to different user roles and tasks (for example: publishers, subscribers, user protocols, and configuration & extension) can improve the usability of an infrastructure by exposing only the necessary concerns to each task. This prevents, for example, the unnecessary handling of exceptions that are not related to the task/user role at hand, and increases the signal-to-noise ratio (Lidwell, Holden et al. 2003) of the system. The results are simpler, easier to learn, and more concise APIs.

We measured the modularity of the selected infrastructures APIs, according to these tasks as shown in Table X, using the DOSC metric. The smaller the DOSC, the more modular an API is.

Table X Infrastructure's API modularity (DOSC)

CONCERN	CORBA-NS	Siena	YANCEES	Java Spaces
Configuration	0.67	N/A	N/A	N/A
Extension	N/A	N/A	0.90	N/A
Reflection	N/A	N/A	0.58	N/A
Initialization	0.53	0	0	0.31
Protocol	0.90	0	0	N/A
Publication	0.85	0.24	0.11	0.64
Subscription	0.87	0.74	0.14	0.77

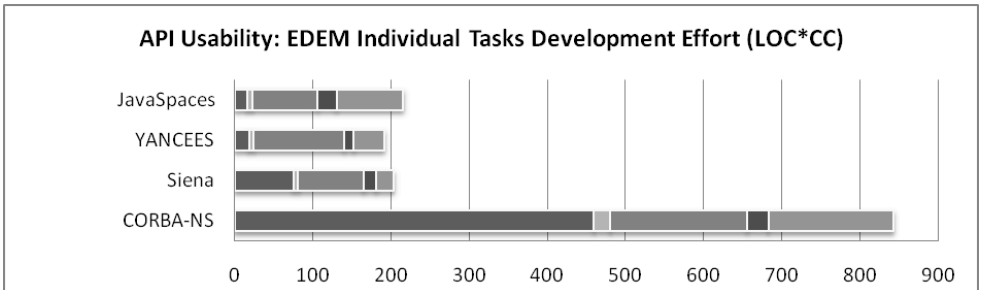
As seen in Table X, the lack of separation of between configuration and regular usage concerns in CORBA-NS, and the support for optional features in JavaSpaces, decrease their publication and subscription API modularity. This is not the case with YANCEES, which provides configuration management automation, and application-specific features. The low DOSC values for both publication and subscription tasks, achieved by Siena are a consequence of its simple API, and the non-existence of extension and configuration concerns built into the software (these concerns are delegated to the application developers).

From the point of view of the extensibility interface, YANCEES presents a high DOSC. This comes from the fact that the extensibility interface of YANCEES is a combination of individual interfaces, one for each variation point. When considered together, as a single extensibility concern, the DOSC increases.

6.2.1.4 API Usability: common task analysis

Every API subsumes different use case scenarios that prescribe the order its methods should be called in support of more complex features. The more complex these steps are, the lower the usability of the infrastructure becomes. Poor designed APIs usually leads to lengthier, more complex and difficult to understand code, leading to less efficient and bulkier programs (Henning 2009). A good API design is one that minimizes the development effort of its users, for the tasks it was designed to support, resulting in code that is simple, efficient and easy to understand.

We analyzed the selected infrastructures, measuring their ability to support the most common publish/subscribe operations as shown in Figure 38 as follows.



	CORBA-NS	Siena	YANCEES	JavaSpaces
connectToServer()	459	75	18	16
createEvent()	21	6	6	6
createSubscription()	175	84	116	84
publish()	27	16	12	24
subscribe()	160	22	39	85

Figure 38 Comparative development effort of most common publish/subscribe tasks (based on EDEM benchmark code)

Figure 38 and Figure 39 present the development efforts (measured in LOC*CC) of the most common publish/subscribe operations. We use both EDEM and CASSIUS case studies as the sources for our measures since they represent two different subscription approaches, i.e. text-based (CASSIUS) and object-based (EDEM) subscription representations. We examined the following tasks:

- *connectToServer()*: represents the programming effort required to obtain a reference to the notification service;
- *createEvent():* represents the effort required to convert EDEM or CASSIUS event representations into the formats supported by each publish/subscribe infrastructure;
- *createSubscription()*: represents the development effort required to translate the application domain subscription representation into (Objects in the case of EDEM and textual expressions in the case of CASSIUS) into the format supported by each publish/subscribe infrastructure;
- *publish()*: represents the task of publishing an event in the target infrastructure. It implicitly calls *creteEvent()* method to perform the adequate data translation.
- *subscribe()*: represents the programmatic effort of subscribing to posting a subscription in the target infrastructure. It implicitly invokes *createSubscription()* to convert the subscription before interacting with the target infrastructure.

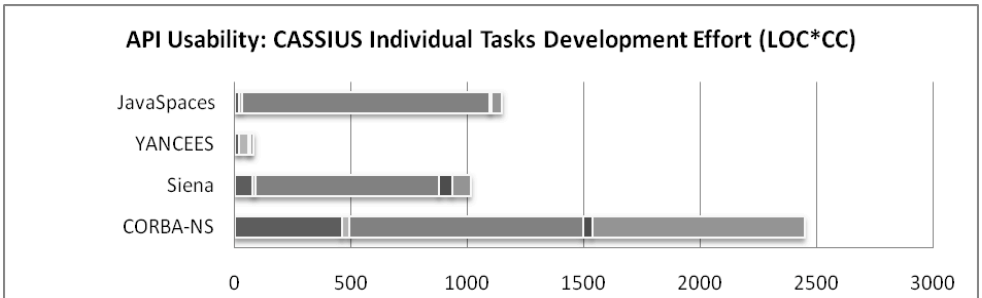
With respect to the connection effort (*connectToServer()* task in Figure 38), CORBA-NS requires users to utilize different factory objects to create event channels and filters, besides of interconnecting different proxies in the process of subscription. This additional work reflects CORBA-NS design decision of supporting fine-grained configurability, and to delegating these concerns to the end-users. This design decision also results in high *connectToServer()* and *subscribe()* efforts if compared to the other infrastructures.

JavaSpaces's additional *subscribe()* effort is a result of its extra exception handling complexity associated to the *notify()* command, which comes from the need to handle exceptions raised by optional features such as transactions, access control and leasing.

The costs of parsing textual and object-based subscriptions. With respect to *createSubscription()* task of Figure 38, both CORBA-NS and YANCEES present high task complexity; whereas Siena and JavaSpaces present similar complexity. This comes from the fact that both YANCEES and CORBA-NS represent their subscription in the form of textual expressions; whereas Siena and JavaSpaces use objects to represent filters and anti-tuples respectively.

In Figure 39 we see a more dramatic effect of the differences between object-based and text-based subscription representations. The parsing of text-based subscriptions into either object-based or other text-based approach can be costly. This comes from the fact that text-based subscriptions are difficult to parse programmatically. The only case where text-based representations of subscriptions provided an advantage was on YANCEES case. The use of XML subscriptions required by CASSIUS application was a perfect fit for YANCEES, allowing the reuse of its internal parsing mechanism, freeing users from having to parse the subscription themselves.

By comparing the CASSIUS case study with EDEM, we observed that application domains that require textual subscription representations must provide mechanisms for automatic parsing of these subscriptions into programmatic representations. The lack of automation result in higher adaptation costs. YANCEES was able to minimize these costs by supporting the automatic parsing of XML subscriptions, a format that matched CASSIUS subscription format.



	CORBA-NS	Siena	YANCEES	JavaSpaces
connectToServer()	459	75	18	16
createEvent()	33	14	38	16
createSubscription()	1003	788	0	1061
publish()	39	56	12	10
subscribe()	913	80	12	44

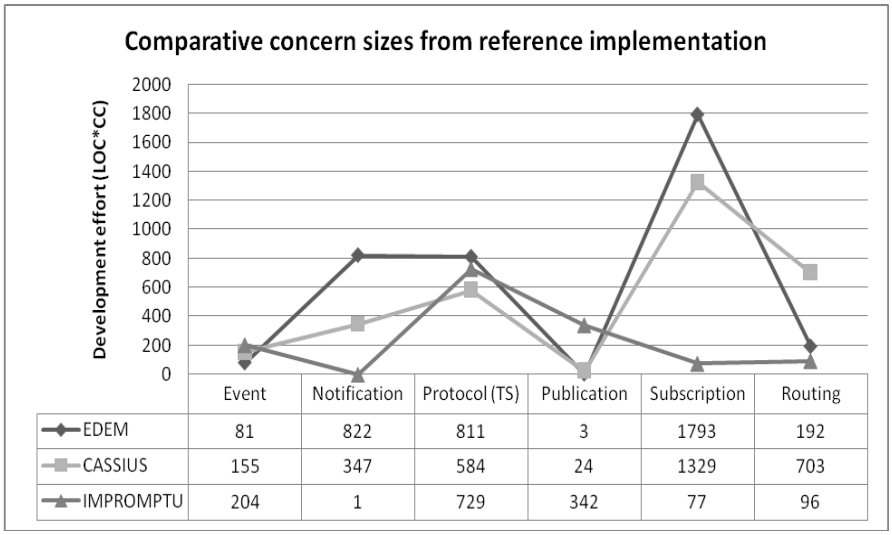
Figure 39 Comparative development effort of most common publish/subscribe tasks (based on CASSIUS benchmark code)

Note that, overall, the development efforts of CASSIUS case study tend to be higher that the ones in EDEM case study. This is not a consequence of the user of textual or object-based subscriptions, but of the number of attributes in CASSIUS event and the differences in the subscription languages of these two infrastructures.

6.2.2 Domain-specific concerns and their development effort

Before analyzing the reusability of the different infrastructures in support of the selected application domains, we turned our attention to the study of their requirements. Our goal is to understand the characteristics of each application domain, and the development effort their features

require. For such, we implemented the three reference APIs from scratch (on top of Java RMI), and measured the development effort of the major publish/subscribe concerns. The results are presented in Figure 40.



	Event	Notification	Protocol (TS)	Publication	Subscription	Routing
EDEM	81	822	811	3	1793	192
CASSIUS	155	347	584	24	1329	703
IMPROMPTU	204	1	729	342	77	96

Figure 40 Comparing concern sizes of build-for-single-use (or BFS) implementations of each reference API used in our study

The development efforts represented in Figure 40 are a direct consequence of the complexity of each domain-specific requirements. Both EDEM and CASSIUS are subscription-intensive application domains, requiring expressive filtering capability. In particular, CASSIUS subscriptions are expressed in XML, which requires extra code devoted to parsing. CASSIUS also requires more complex content-based subscription routing, whereas EDEM routing algorithm is simplified due to the use of fixed object-based events. EDEM support for rules, with different notification policies, requires a higher amount of notification code. All application domains are also protocol-intensive, requiring extra code to handle event source advertisement and browsing (CASSIUS), tuple manipulation (EDEM), and peer-to-peer communication (IMPROMPTU). IMPROMPTU also exercises the publication model with the need for repeated event filters and a publish-to-peers component that work with protocol components to multicasts published events to all the peers in the network.

In our case studies, we use these baseline development efforts to compare the impact of each versatility approach in the development effort of each one of these concerns.

6.2.3 Case studies development effort

After analyzing the selected publish/subscribe infrastructures (section 6.1) and the domain requirements of the selected case studies (section **Error! Reference source not found.**), we turn our attention to the costs of reusing these systems in the implementation of the three reference APIs we derived in section 4.3. In this section, we present the development effort (measured in LOC*CC), for each case study, and discuss the main factors that contribute to this effort.

Different factors contribute to the development effort (or reusability) of the selected infrastructures. In particular, we identified the following major factors:

The role of reuse strategy. While Siena, CORBA-NS and JavaSpaces are reused as black boxes on top of which each reference API is implemented, YANCEES is reused as a grey-box,

i.e. the YANCEES server is customized with plug-ins and language extensions to best fit the each API requirements. Then, a thinner layer of code, described as YANCEES (Client) in our evaluation, is written on top of the tailored YANCEES (Server), to implement a façade that matches the reference APIs. For comparative purposes, the combination of both client and server side development efforts is presented as YANCEES (Client+Server) results in our evaluation results.

Middleware costs. In the three case studies, since CORBA-NS, JavaSpaces and Siena are reused as black box event processing components, there is a need to provide additional threading and distribution. In other words, a new infrastructure must be built by combining the routing core with additional features in the implementation of each reference API. This cost is reflected in the higher middleware development effort when these three infrastructures are reused (see the Middleware concern in Figure 41, Figure 43 and Figure 45). YANCEES server is an exception to this rule. Since it already supports multithreading and is customized and extended "from the inside", these features are inherently reused. This explains the reduced middleware development effort observed in YANCEES. On the server side, YANCEES middleware concerns represent the implementation of remote interfaces by protocol plug-ins; whereas on the client side, it represents the effort to connect to the server.

Adaptation costs. Adaptation costs are those spent on converting data representations (subscriptions and events) from and to the application domain representation and the infrastructures. These costs vary according to the subscription format adopted. For example, in both EDEM (Figure 43) and IMPROMPTU (Figure 45) case studies, subscriptions are expressed as objects, whereas CASSIUS uses a XML-based subscription language (Figure 41). Since CORBA-NS and YANCEES rely on text-based subscriptions, IMPROMPTU and EDEM constraint objects need to be parsed into the languages of these infrastructures. This results in higher costs of adaptation, when these infrastructures are reused to implement object-based APIs, and explains why Siena adaptation costs were relatively small for IMPROMPTU and EDEM, but high for CASSIUS.

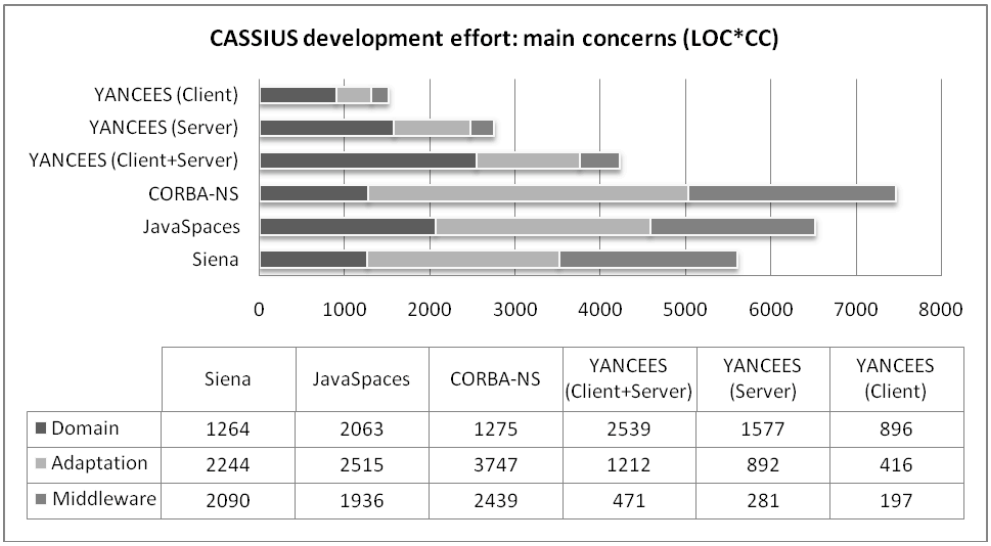
In another example, comparing CORBA-NS and Siena, both infrastructures provide compatible set of features (content-based filtering and routing), which makes the implementation of domain-specific features in these two infrastructures compatible, in terms of domain-specific effort. Their adaptation costs, however, differ, due to the use of subscription languages in CORBA-NS and subscription objects in Siena, and due to the CORBA-NS API complexity.

Domain-specific costs. As discussed in section 5.5, domain-specific development costs are those devoted to complement the features provided by each infrastructure in the implementation of each API. It includes, for example, the development of protocol features and advanced event processing capability not provided by the infrastructures, as well as the main commands of each case study API. In the particular case of YANCEES, it includes the effort to develop server-side plug-ins.

6.2.3.1 CASSIUS Case Study

In this first case study, we analyze the reusability of the infrastructures in support of CASSIUS requirements. The results are summarized in Figure 41; whereas the development effort for domain-specific concerns are broken down into individual publish/subscribe concerns as show in Figure 42.

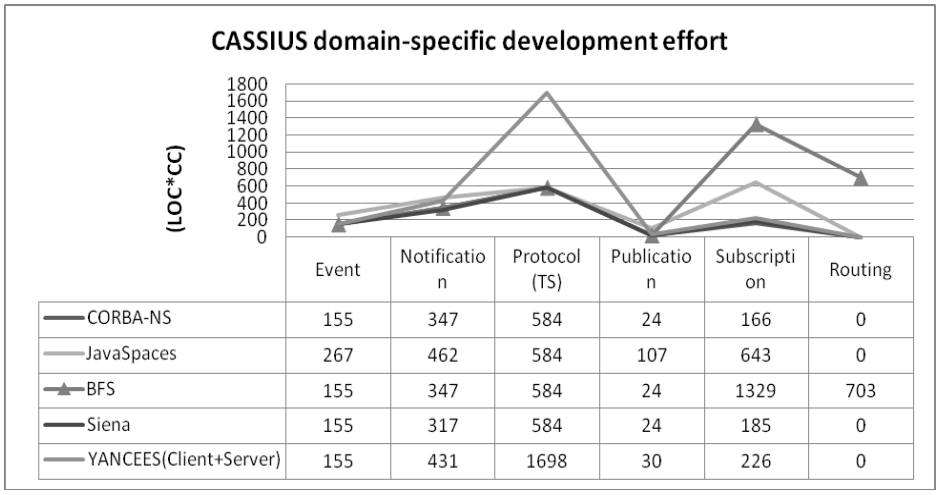
Note that in Figure 42 we compare the results with the build-for-single use (or BFS) baseline, identifying the cases where the reuse of the infrastructures improve (reduce) or worsen (increase) the development effort for a given concern. In doing so, we are able to identify extra costs associated to specific versatility approaches.



	Siena	JavaSpaces	CORBA-NS	YANCEES (Client+Server)	YANCEES (Server)	YANCEES (Client)
Domain	1264	2063	1275	2539	1577	896
Adaptation	2244	2515	3747	1212	892	416
Middleware	2090	1936	2439	471	281	197

Figure 41 CASSIUS case study development effort

As shown in Figure 41, in the CASSIUS case study, YANCEES requires the lowest overall reusability effort, being followed by Siena, JavaSpaces and CORBA-NS. Since CASSIUS subscription language is represented in XML, it was fully compatible with YANCEES subscription model, significantly reducing the adaptation costs in that case. However, considering only the domain-specific requirements effort (see Figure 41) the situation is different. Siena and CORBA-NS have the lowest domain-specific development efforts, followed by JavaSpaces and YANCEES. The reason for this is explained by analyzing the individual domain-specific costs shown in Figure 42.



	Event	Notification	Protocol (TS)	Publication	Subscription	Routing
CORBA-NS	155	347	584	24	166	0
JavaSpaces	267	462	584	107	643	0
BFS	155	347	584	24	1329	703
Siena	155	317	584	24	185	0
YANCEES(Client+Server)	155	431	1698	30	226	0

Figure 42 CASSIUS benchmark: domain-specific development effort.

The costs of generality and separation between protocol and publication interfaces. By analyzing Figure 42, we see that YANCEES' protocol development effort was relatively high. This is a consequence of the generality of the protocol model. Differently from the subscription and notification models that automatically parse XML-based subscriptions and dynamically allo-

cate plug-ins built accord to standard interfaces, YANCEES protocol model is very general, it provides no standard templates or automation aids to the developers. Moreover, its interface is independent from the main YANCEES publication façade. While this strategy allows the support of different types of protocols, and separation of interface concerns, the lack of automation can result in higher costs for infrastructure developers. For example, protocol plug-ins developers must provide their own distribution and multithreading. Moreover, in the specific case of CASSIUS benchmark, developers must handle the translation of client-side subscriber references into server-side references. This is important, since in CASSIUS, protocol users are uniquely identified by their subscriber interfaces. Client-side interfaces need to match their correspondent server-side references, and must be the same as the ones used in the publication/subscription API. Since YANCEES separates between subscription and protocol APIs, there is a need to keep track of these references by accessing YANCEES internal subscriber interface registry, which requires extra development effort.

The subscription development effort also presented variations. As observed in Figure 42, the CASSIUS case study subscription costs for all reused infrastructures are lower than the BFS reference implementation. This comes from the fact that CASSIUS subscription parsing costs are counted as part of adaptation costs of the reused infrastructures, while in the BFS, it is counted as part of the subscription development effort.

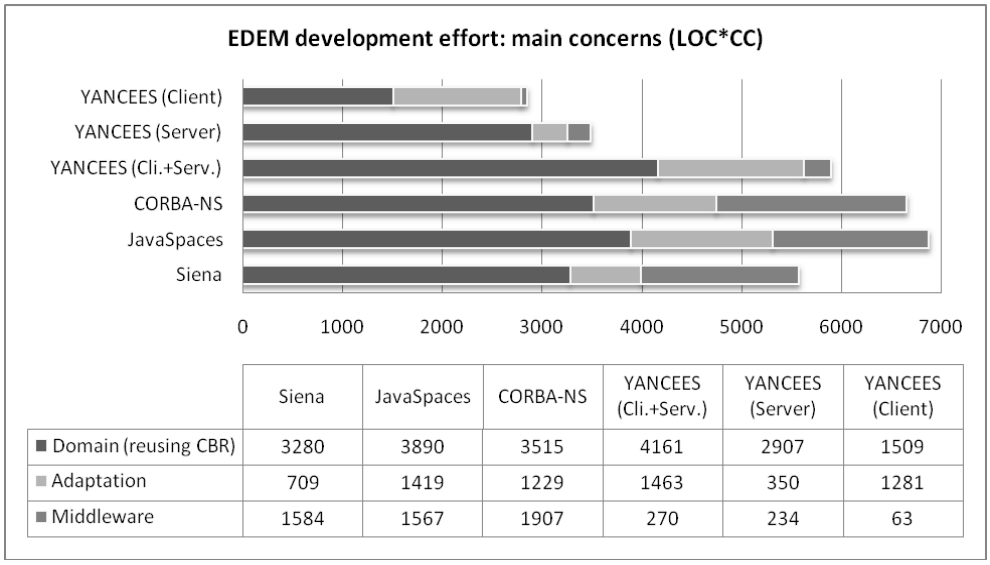
The costs of semantic mismatches. JavaSpaces subscription costs are higher than the other infrastructures. This is a consequence of a mismatch between the filtering capability of JavaSpaces and the subscription language of CASSIUS. JavaSpaces does not support content operators such as: >,<,>=,<=, and == over numerical values. This forced us to implement our own numerical matching schema, which increased the domain-specific complexity for this infrastructure.

Flexibility and generality implementation costs. Finally, YANCEES' subscription costs in Figure 42 are slightly higher than the other infrastructures. This is a consequence of two major factors. First, in YANCEES, plug-ins need to extract parameters from DOM representations of subscription language commands, a cost that comes from the reuse of YANCEES' internal XML parser. Second, YANCEES subscription and notification plug-ins operate over generalized attribute/value pair events. This generalized data structure requires the handling of exceptions such as: *AttributeNotFoundException* and *AttributeTypeMismatch*, which increases the cyclomatic complexity of the implementation, and consequently its development effort (measured as LOC*CC). The equivalent components in the other infrastructures execute in the client side and operate over application-specific event formats, resulting in simpler implementations.

6.2.3.2 EDEM Case Study

In the EDEM case study, which results are shown in Figure 43, Siena presented the lowest total development effort, being followed by YANCEES, CORBA-NS and JavaSpaces.

The costs of supporting method-based variability. Even though JavaSpaces requires less or equivalent amount of lines of code to support EDEM applications, it presented the highest development efforts in the EDEM case study. In particular, these costs are associated to the domain-specific concern implementation. The reason for that is the high complexity of the code that interacts with JavaSpaces. Due to the optional transaction, authentication and leasing features, exceptions need to be handled for every JavaSpaces API call. The handling of exceptions increases the cyclomatic complexity of the code, consequently its development effort, even though the optional feature that raises the exception is not used. In short, the extra options supported by JavaSpaces ends up increasing the client side development effort for all case studies.

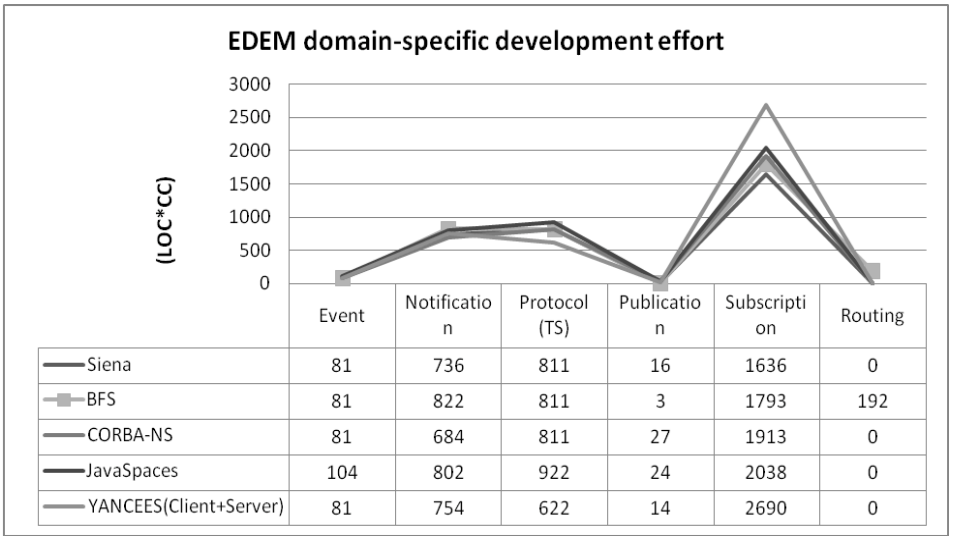


	Siena	JavaSpaces	CORBA-NS	YANCEES (Cli.+Serv.)	YANCEES (Server)	YANCEES (Client)
Domain (reusing CBR)	3280	3890	3515	4161	2907	1509
Adaptation	709	1419	1229	1463	350	1281
Middleware	1584	1567	1907	270	234	63

Figure 43 EDEM case study development effort

As shown in Figure 44, the domain-specific costs of reusing the selected infrastructures closely follows the costs of developing the system from scratch. Some few deviations, however, are observed. YANCEES subscription costs are slightly higher than the other infrastructures. As observed in the CASSIUS case study, this comes from the need of subscription plug-ins to extract parameters from DOM objects, from the handling of generalized event representations, and the modularization of YANCEES model, that originates more classes.

In particular, we measured the EDEM case study difference between the equivalent components used to implement advanced event processing in YANCEES server side (based on generalized events, with extra framework interaction overhead) and the equivalent components implemented in the client size (based on object-based application-specific events, with no extra overhead). YANCEES components were in average 20% (or 12 lines of code) larger, and 60% more complex (1.7 versus 2.81 in average) than the ones build to support application-specific events.

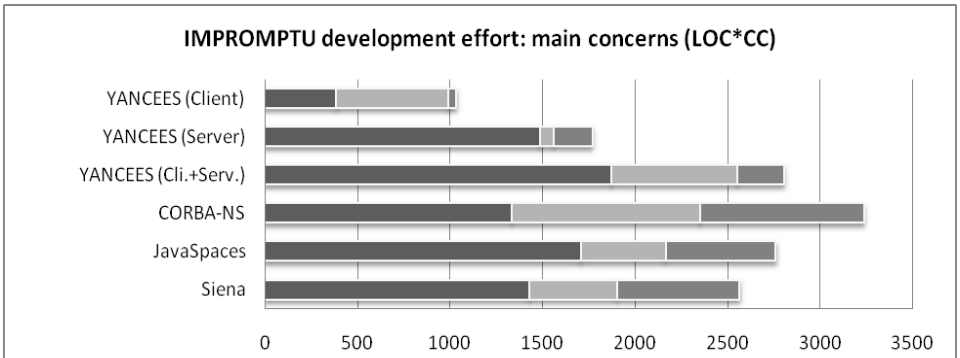


	Event	Notification	Protocol (TS)	Publication	Subscription	Routing
Siena	81	736	811	16	1636	0
BFS	81	822	811	3	1793	192
CORBA-NS	81	684	811	27	1913	0
JavaSpaces	104	802	922	24	2038	0
YANCEES(Client+Server)	81	754	622	14	2690	0

Figure 44 EDEM case study: domain-specific development effort

6.2.3.3 IMPROMPTU Case Study

The results of IMPROMPTU case study are shown in Figure 45. IMPROMPTU case study requires a simple topic-based routing and record-based event model that relies on push notification mechanisms. In this benchmark, Siena presents the lowest development effort, being followed by JavaSpaces, YANCEES and CORBA-NS.



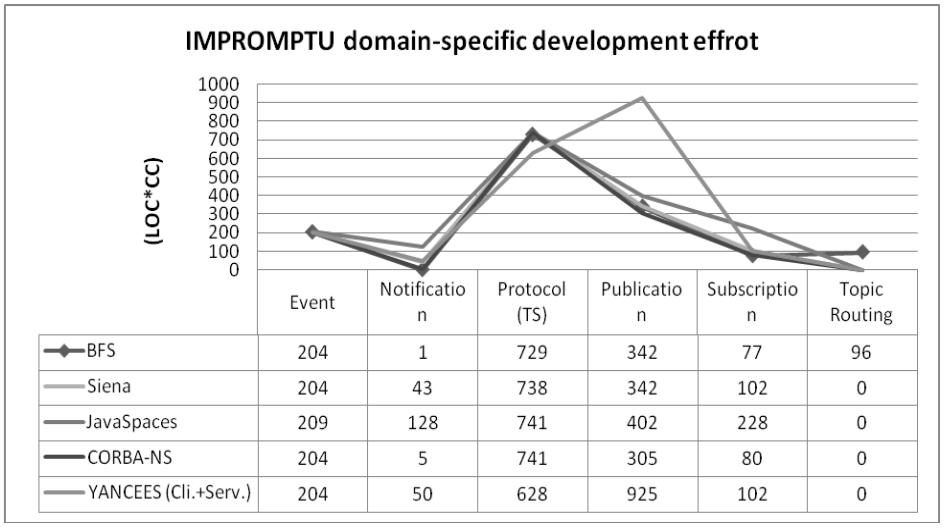
	Siena	JavaSpaces	CORBA-NS	YANCEES (Cli.+Serv.)	YANCEES (Server)	YANCEES (Client)
Domain (reusing CB core)	1429	1707	1335	1872	1486	386
Adaptation	476	458	1015	682	75	607
Middleware	659	595	885	250	208	42

Figure 45 IMPROMPTU case study development effort

Flexibility and generality overhead. As shown in Figure 46, the publication model of YANCEES requires extra client side development effort, when compared to the domain-specific implementation using the other infrastructures. Again, this is a consequence of the publication plug-in model of YANCEES, which requires the implementation of a few extra interfaces, used more classes (one for each filtering concern) and relies on generalized event formats, which require extra exception handling.

For example, In the other infrastructures, IMPROMPTU's duplicate events filter and the publish-to-peers algorithm are implemented as part of the *publish()* command, being designed to operate over two fixed event formats. In YANCEES, on the other hand, these concerns are implemented by separate filter components installed in the publication variability dimension. These filters operate over *YanceesEvent* objects, generalized attribute/value pairs that represent both File and GUI events from IMPROMPTU.

As a consequence, the components developed for YANCEES are in average 10% larger (an average of 10 LOC more), and 79% more complex (2.61 versus 3.32 in average) than these developed for a single use. Moreover, due to the need for exception handling, the cyclomatic complexity of the *RepeatedEventsFilter* in YANCEES is twice as that of the other implementations (i.e. it increases from 2 to 4).



	Event	Notification	Protocol (TS)	Publication	Subscription	Topic Routing
BFS	204	1	729	342	77	96
Siena	204	43	738	342	102	0
JavaSpaces	209	128	741	402	228	0
CORBA-NS	204	5	741	305	80	0
YANCEES (Cli.+Serv.)	204	50	628	925	102	0

Figure 46 IMPROMPTU case study: domain-specific development effort

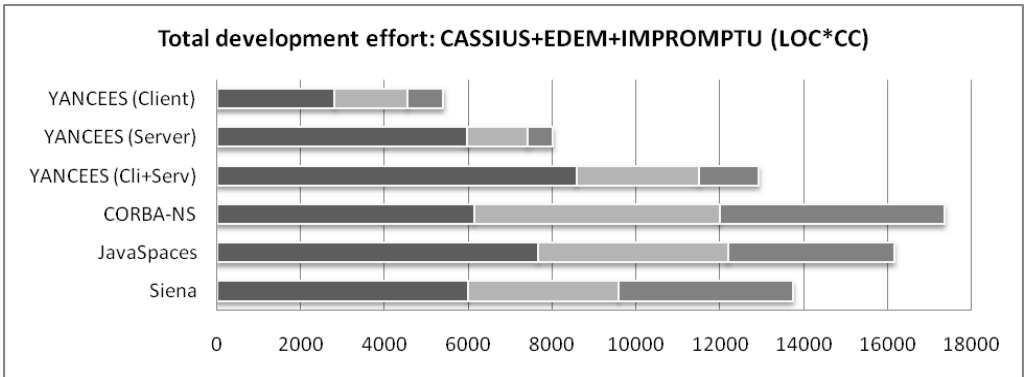
Notification model mismatch. As shown in Figure 46, JavaSpaces' notification model presents a relatively high development effort. This comes from the mismatch between IMPROMPTU push notification and JavaSpaces pull notification models (previously discussed in section 4.2.3). This mismatch also creates an overhead that jeopardizes its performance, as will be discussed in section 6.2.6.

6.2.4 Total development effort

When considering the total development effort, i.e. by combining the results of the three case studies (as shown in Figure 47), YANCEES emerges as the infrastructure that requires the smallest total development effort, being closely followed by Siena.

YANCEES fitness. YANCEES fitness to these combined problem domains is a consequence of its lower Middleware and Adaptation development efforts, together with the reusability of some of its components such as the content-based filtering, the ability of YANCEES to natively handle XML subscriptions (in the CASSUS case study), and the lower client-side development costs, resulting from the ability of YANCEES to closely meet the application domain requirements. In sum, the highest costs of developing YANCEES components start to pay-off, on the course of successive reuses.

Overall, YANCEES component model requires a higher development effort (LOC*CC) for implementing domain-specific features, with potential higher development complexity in the server side (see Figure 49). However, these higher costs come with lower maintainability costs (Figure 50), discussed in 6.2.5.



	Siena	JavaSpaces	CORBA-NS	YANCEES (Cli+Serv)	YANCEES (Server)	YANCEES (Client)
Domain	5973	7660	6125	8572	5969	2791
Adaptation	3613	4530	5861	2925	1449	1739
Middleware	4150	3960	5362	1422	590	866

Figure 47 Total development effort for the tree case studies

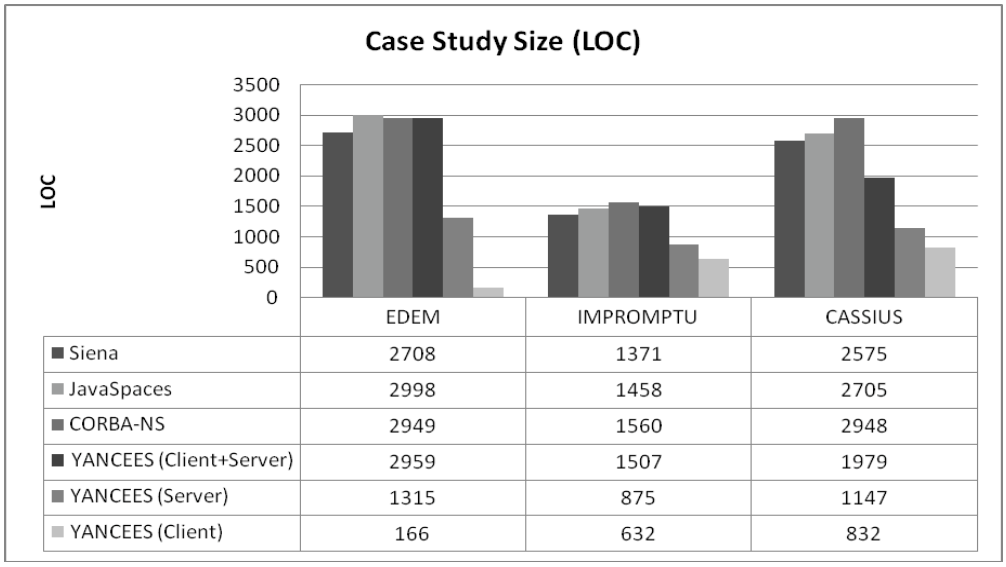
JavaSpaces functionality mismatches. JavaSpaces' higher costs in two of the case studies are mainly due to three reasons: (1) the mismatches between the *notify()* command semantics, that does not automatically deliver the matched tuples in its notification; (2) the lack of numeric filter operators (critical to EDEM case study), and (3) the lack of push notification model (required for IMPROMPTU and EDEM).

CORBA-NS configurability costs. CORBA-NS extra adaptation costs are a consequence of its extensive API size and the externalization of configuration management concerns to the end users, as previously discussed in section 6.2.1.4.

Siena balanced development costs. Siena reuse costs are balanced between middleware, adaptation and domain-specific. Its lack of explicit extensibility and configurability requires extra middleware costs to wrap-up the resulting implementation under a distributed, multi-threaded API. Its domain-specific costs are associated to the additional features required by the application domains, that can be implemented through the layering and reuse of Siena's minimal core functionality. Its adaptation costs are a result of the use of generalized event representation and content-based filtering, which requires adequate translation from the application domain specific formats.

6.2.4.1 Breaking down the development effort costs

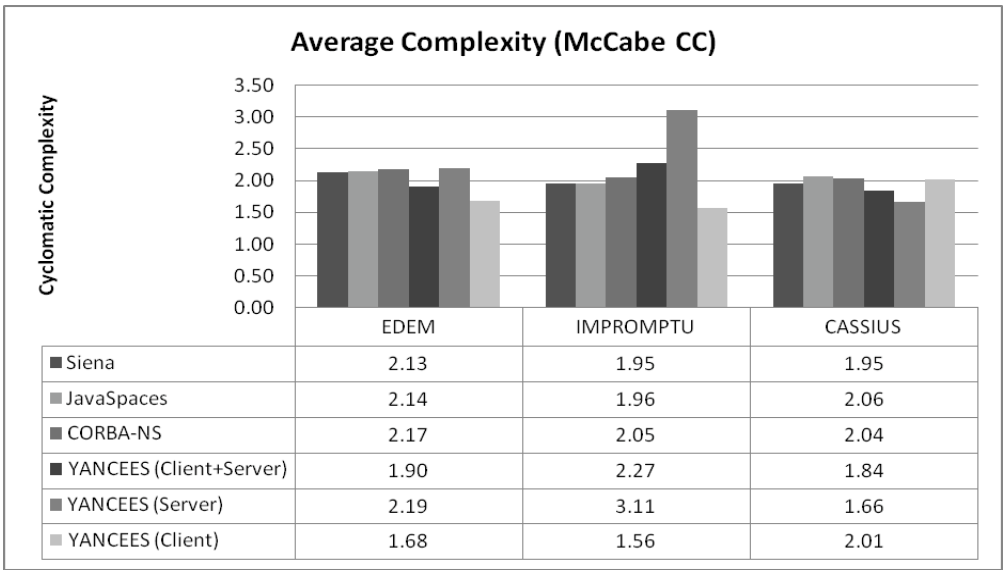
When separating the number of lines of code from code complexity for the three case studies, we obtain the graphics in Figure 48 and Figure 49.



	EDEM	IMPROMPTU	CASSIUS
Siena	2708	1371	2575
JavaSpaces	2998	1458	2705
CORBA-NS	2949	1560	2948
YANCEES (Client+Server)	2959	1507	1979
YANCEES (Server)	1315	875	1147
YANCEES (Client)	166	632	832

Figure 48 Total lines of code per case study and infrastructure

As shown in Figure 48, in terms of LOC, YANCEES required the same or less lines of code in order to support the three case studies requirements, however, as seen in Figure 49, YANCEES internal framework overhead, increasing number of modules, and the use of generalized event representations, increase the complexity of the server-side YANCEES extensions.



	EDEM	IMPROMPTU	CASSIUS
Siena	2.13	1.95	1.95
JavaSpaces	2.14	1.96	2.06
CORBA-NS	2.17	2.05	2.04
YANCEES (Client+Server)	1.90	2.27	1.84
YANCEES (Server)	2.19	3.11	1.66
YANCEES (Client)	1.68	1.56	2.01

Figure 49 Average cycloramic complexity per case study and infrastructure

6.2.5 Client code maintainability

We also analyzed the resulting modularity of the code produced for the three different case studies. We correlate the maintainability of the infrastructure with its modularity.

Even though we strived to keep the same implementation style for all infrastructures, small differences were observed due to the extensibility interfaces (in the case of YANCEES), and the differences in the infrastructures programming models (in particular JavaSpaces). In these cases, we adapted the features implementation as necessary. We measured both the diffusion of concerns over components (DOC) (Figure 50), and the degree of scattering over components (DOSC) (Figure 51).

Even though YANCEES code is more extensive in terms of number of classes used, the plug-in code tends to be more modular than the code developed around existing infrastructures. This comes as a consequence of the design for change principle applied in its design, which supports extensibility and configurability along the main publish/subscribe domain variability dimensions, as discussed in section 6.1.4.

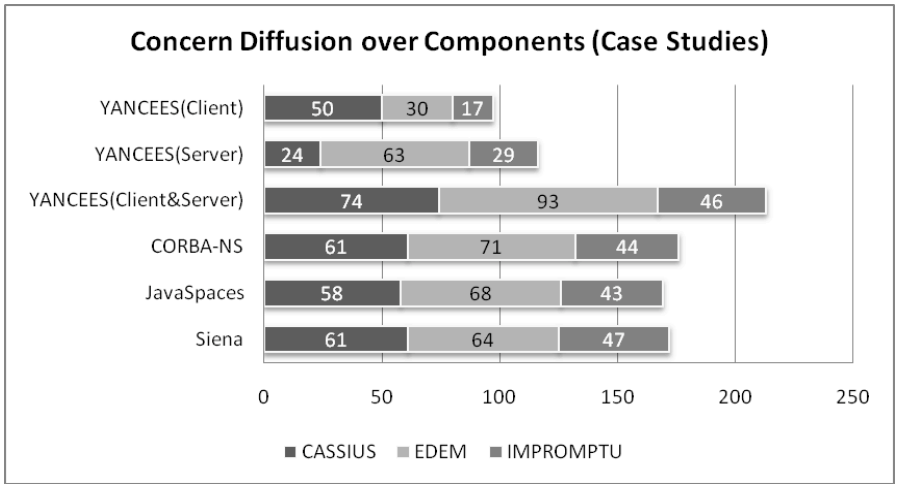


Figure 50 Comparing CDC for the tree case studies

In particular, YANCEES presented a better modularity for the CASSIUS and IMPROMPTU case studies, APIs that require specialized features in variability dimensions other than the subscription language alone.

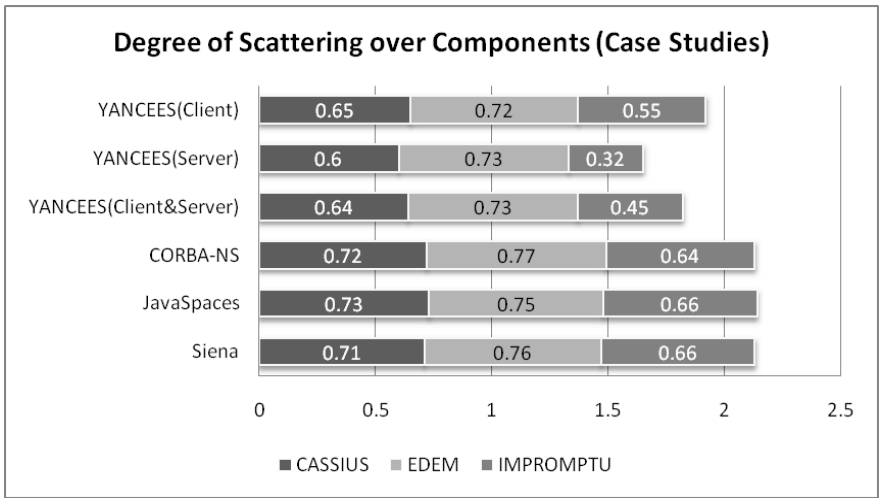


Figure 51 Comparative DOSC for the three case studies

The higher modularity of YANCEES has also consequences for reusability in a long run, allowing an ecosystem of components to be developed. In our case study, the push plug-in, the content-based router, and the standard filters were successively reused.

6.2.6 Performance

A critical feature in middleware infrastructures is the ability to match the performance requirements of the application domain. In this section, we compare the responsiveness of our three case studies we analyzed.

For each reference API implementation, we developed a simple benchmark that exercises their main API commands in support of common tasks. The goal of the benchmark is to measure the fitness and responsiveness of the underlying publish/subscribe infrastructures to the requirements posed by ach API.

The benchmarks were run on two Win32 Pentium 4 workstations, with 1GB of RAM, interconnected by a 100 Mbps Ethernet connection on a Local Area Network.

6.2.6.1 EDEM

In the first case study (Figure 52) we measured the performance of EDEM API when implemented on top of the selected infrastructures. In particular, we calculated the average responsiveness (in milliseconds), of four use cases that exercises: the tuple space manipulation API, simple content-based filtering, pattern matching and rules.

While YANCEES and Siena's performance were comparable, JavaSpaces and CORBA-NS experienced additional overhead. This overhead is explained by the extra communication costs between the server code and these infrastructures, that execute in different processes, and respectively rely on Java RMI and CORBA IIOP communication protocols. The JavaSpaces implementation also experienced higher delays due to a semantic mismatch between its notification model (pull) and EDEM push model, which was compensated with extra code.

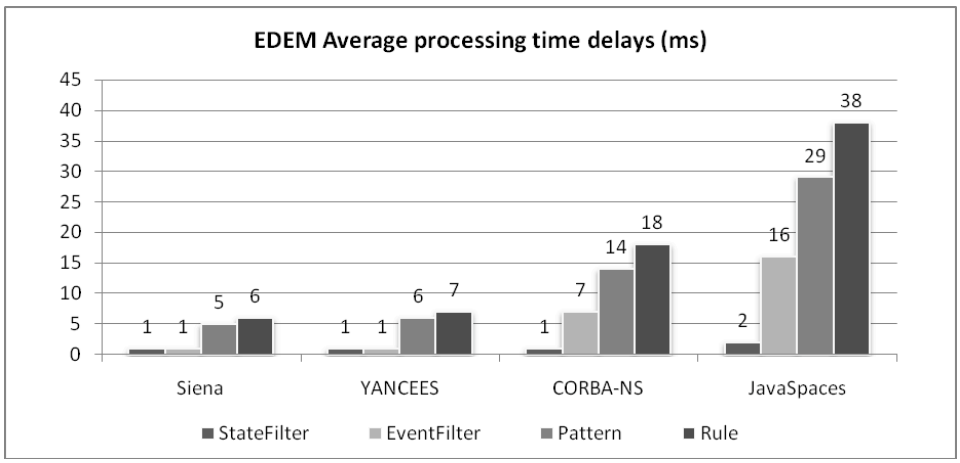


Figure 52 EDEM common tasks performance analysis

6.2.6.2 CASSIUS

In the CASSIUS benchmark, we calculate the average responsiveness (in milliseconds) of two use cases, one that builds and browses through a simple event source hierarchy, and another that performs a simple content-based subscription using pull notification style.

In the CASSIUS benchmark (see Figure 53), most infrastructures presented similar response times, with Siena being slightly faster. An exception was JavaSpaces, which native pull notification model was better fit to the application requirement. The result was a faster response of the API implemented on top of JavaSpaces.

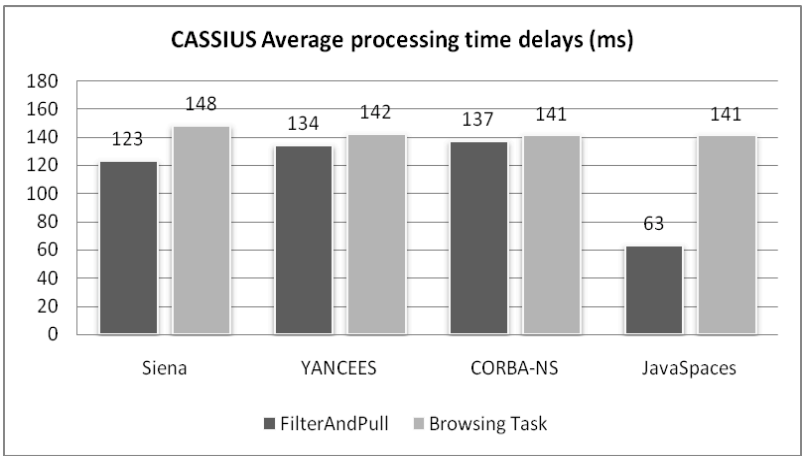


Figure 53 CASSIUS common tasks performance analysis

Note that in the CASSIUS benchmark, the browsing task is similar to all implementations since it is provided by an independent component in JavaSpaces, Siena and CORBA-NS, and a dedicated server-side plug-in in YANCEES.

6.2.6.3 IMPROMPTU

In the IMPROMPTU benchmark, we measured the average time from the publication of an event in one host, to the receiving of this event in another host.

In this benchmark, (presented in Figure 54), CORBA-NS and JavaSpaces delays are attributed to: API mismatches (the implementation of push notifications using JavaSpaces) and the fact that both JavaSpaces and CORBA-NS execute in separate processes, requiring RMI and ORB inter-process communication, respectively.

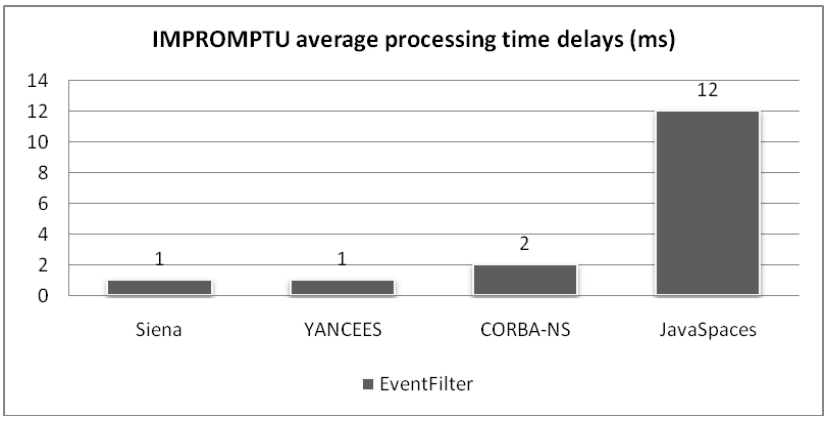


Figure 54 IMPROMPTU common task performance analysis

YANCEES ability to support multiple cores was also important in the IMPROMPTU case study, making YANCEES perform better than CORBA-NS. In this case study, a topic-based core was used in YANCEES to speed up the matching between IMPROMPTU events and topic-based subscriptions.

6.3 Summary of results

In this section, we summarize our findings presenting both a quantitative and qualitative assessment of the results.

6.3.1 Quantitative results

We quantified the selected versatility approaches, according to the different measures we collected. For such, we ranked these features from 1 (lowest) to N (highest), where N is the total number of infrastructures (4 in our case), with lowest values being best. We then derived an overall versatility approach score as the sum of all the scores that correspond to its features as shown in Table XI. In this calculation, we consider all the software qualities as having equal importance to the versatility of the approach.

Table XI Quantitative ranking of the versatility from developers and users perspectives (smaller is better)

	Minimal Core (Siena)	Coordination Languages (Java Spaces)	One-size-fits-all (CORBA-NS)	Flexible (YANCEES)
Infrastructure Code Size	2	3	4	1
Flexibility	2	4	3	1
Infrastructure Modularity	3	1	4	2
Reusability	2	3	4	1
Performance	1	3	4	2
Client API Task Analysis	2	3	4	1
Client Code Modularity	2	3	4	1
TOTAL	**14**	**20**	**27**	**9**

Overall, YANCEES achieves a more favorable balance among the different measures we collected. It is followed by Siena, JavaSpaces and CORBA-NS. We summarize the reason for these differences in Table XII, where we list the main costs and benefits of each approach.

6.3.2 Versatility approaches trade-offs

The results of our analysis are qualitatively summarized in Table XII.

Table XII Qualitative summary of the versatility strategies

INFRASTRUCTURE	BENEFIT	COSTS
Minimal Core (Siena)	- Efficiency - API simplicity - Small code size - Layered reusability	- Inflexibility - High abstraction distance - low maintainability
Coordination Languages (Java Spaces)	- API simplicity - Additional features - Layered reusability	- Moderate performance - *notify()* semantic mismatch - Inflexibility - Low maintainability - High abstraction distance
One-size-fits-all (CORBA-NS)	- Configurability: Variety of options and features - Compatibility with existing protocols - Layered reusability	- Moderate performance - API usage complexity - Low maintainability - High abstraction distance
Flexible (YANCEES)	- Efficiency - API simplicity - Small size - Higher flexibility - Reuse of existing components, distribution and threading - Improved maintainability - Reduced abstraction distance	- More complex plug-in code due to generality - Need for configuration management (handled by configuration files) - Framework reusability issues

We also present the results in terms of high/medium/low qualifiers as shown in Table XIII as follows.

Table XIII Qualitative evaluation of the infrastructures in terms of high/medium/low qualifiers

INFRA-STRUCTUE	Flexi-bility	Infra Main-tainability	Reusability	Client Code Maintainab.	Usability	Performance
Minimal Core (Siena)	Low	High	High	Low	Medium	High
Coordination Languages (Java Spaces)	Low	Low	Low	Low	Medium	Low
One-size-fits-all (CORBA-NS)	Medium	High	Low	Low	High	Medium
Flexible (YANCEES)	High	Medium	High	High	Low	High

6.3.3 Summary of findings

Minimal core approaches as Siena are efficient, have simple APIs and are easier to build than one-size-fits-all infrastructures. These infrastructures are reused by laying functionality on top of them. Through the use of generalized subscription and event representations, they can support a large set of application domains. In spite of these benefits, their core functionality is inflexible (not easily configurable or extensible), being limited by the generalized (but fixed) event, subscription and notification capabilities they provide. This approach supports black-box reuse, which results in higher middleware and adaptation costs if compared to YANCEES.

Coordination languages as JavaSpaces have the similar generalization benefits of minimal core infrastructures. In the particular case of JavaSpaces, we found problems with semantic mismatches and performance, a consequence of the inflexibility of the tuple space model with respect to filtering capability and the supported pull notification model.

One-size-fits-all infrastructures as CORBA-NS support a large set of features through specialized variability and configurability. In this approach, configurability is delegated to application developers, through programmatic interfaces (for example: factories, configuration methods, and composition). This manual configurability decreases the system usability. They are also slower than minimal core and flexible approaches, as shown in our performance benchmarks (Figure 52, Figure 53 and Figure 54). This poor performance is a consequence of the one-size-fits-all syndrome (Long 2001), where most common features end up paying the price for the extra features supported.

Flexible approaches as YANCEES may require, in some cases, more complex, lengthy and componentized code (Figure 49 and Figure 50). This is a consequence of the generalization and separation of concerns they support. However, the resulting code is more modular and reusable. The ability to customize the infrastructure to the application domain requirements reduces the abstraction distance between the required and provided functionality, reducing the client side development effort to build applications based on this infrastructure (Figure 47).

Whereas flexible and minimal core approaches have shown to be more versatile than one-size-fits-all, minimal core and coordination language approaches, the selection of each strategy depends on different factors related to the fitness of each infrastructure to the application domain

requirements. For example, minimal core infrastructures are fast and can have relatively low adaptation costs in many application domains; whereas flexible approaches are more indicated to support the development of software product lines (Clements and L. Northrop 2002), where the initial costs of adaptation and development are paid over successive reuses of the infrastructure in the development of slightly different publish/subscribe infrastructures.

In the next chapter, we compare the individual software qualities measured in our case studies, identifying possible correlations and trade-offs between these qualities. These correlations will be further used in support of the guiding principles discussed in Chapter 8.

Chapter 7. Analysis of Versatility Trade-offs

In this chapter, we analyze the trade-offs defined by the different software qualities measured in Chapter 6. Our goal is first to identify correlations (or lack thereof) between these software qualities; and second, to provide quantitative and qualitative data to support the principles and guidelines that we will discuss in Chapter 8.

7.1 Infrastructure modularity and flexibility trade-offs

First, we analyze the impact of modularity on software flexibility. Modularity is known for improving software maintainability and is generally accepted as a way to isolate software concerns improving the locality of change (Sullivan, Griswold et al. 2001). In order to analyze the correlation of these two software qualities in the selected infrastructures, we plot these two measures in the chart of Figure 55.

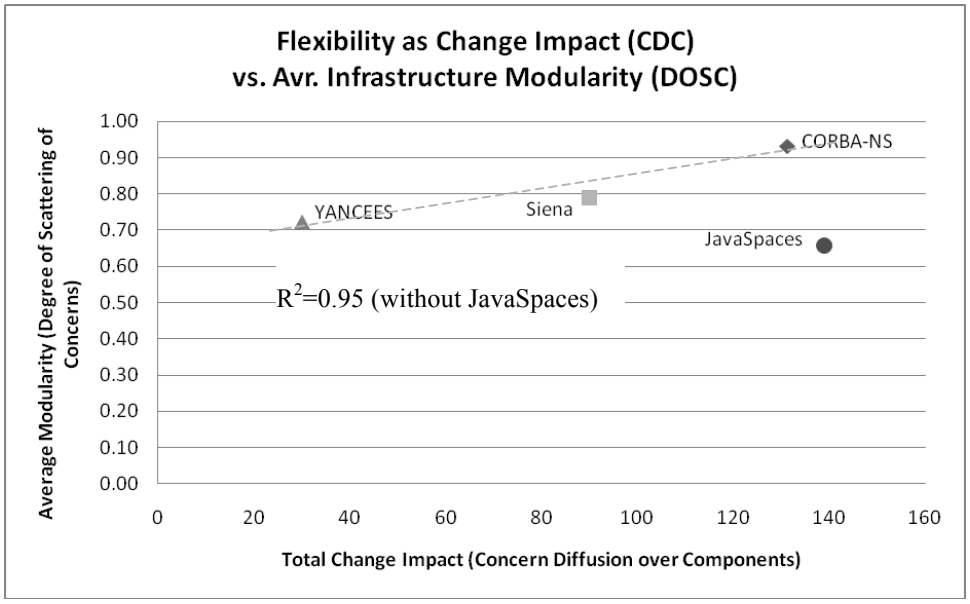


Figure 55 Total change impact (adding the change impact of each variability dimension) versus Average modularity of the analyzed infrastructures

As seen on Figure 55, YANCEES has the lowest total change impact (the sum of the change impacts presented in Figure 36), followed by Siena, CORBA-NS and JavaSpaces. Even though JavaSpaces is more modular (lowest DOSC), its lack of design for change and extensibility along the main publish/subscribe dimensions, places it as the infrastructure with the highest change impact (lowest flexibility).

CORBA-NS was designed to be configurable and to support a large set of features. Its decomposition of features based on proxies and methods, however, resulted in low modularity (high DOSC), and low flexibility (high change impact). Siena's use of generalization and its simple simplified implementation resulted in lower change impact, and average modularity (between JavaSpaces and CORBA-NS). Finally, YANCEES design for flexibility resulted in average modularity but low change impact (high flexibility).

JavaSpaces is an outlier, it has the most modular implementation, but is the least flexible of all infrastructure (towards the publish/subscribe design concerns). The coefficient of correlation with JavaSpaces is 0.23, without JavaSpaces, it is 0.95

Conclusions. By comparing the impact of modularity on flexibility, as shown in Figure 55, we conclude that even though modularity may improve the overall maintainability of the system, it does not automatically support flexibility. Flexibility is more a function of design for change, toward planned variability dimensions, than of the intrinsic modularity (or even configurability) of the infrastructure.

7.2 Infrastructures API usability trade-offs

The infrastructure API defines the major interface of a software infrastructure, it directly supports its reuse, configuration and extension.

In this section, we look for correlations between the infrastructures API characteristics as: the API size, and supported subscription format, with important software qualities such as: reusability (development effort, complexity, length of code), client code maintainability (modularity), and infrastructure performance (response delays of common operations).

7.2.1 Impact of API size on the total development effort

In order to analyze the impact of the infrastructure API size on the total development effort, we plotted these two measures together, obtaining the chart of Figure 56. If we set JavaSpaces apart for a while, we see a linear correlation ($R^2 = 0.86$) between the API size of the remaining infrastructures and the total development effort of the case studies using these systems.

API size indicates more features in terms of generalization, variability, and extensibility. This usually results in extra costs of adaptation (generalization), selection (variability), and extension (extensibility), which correlates with a higher application development effort.

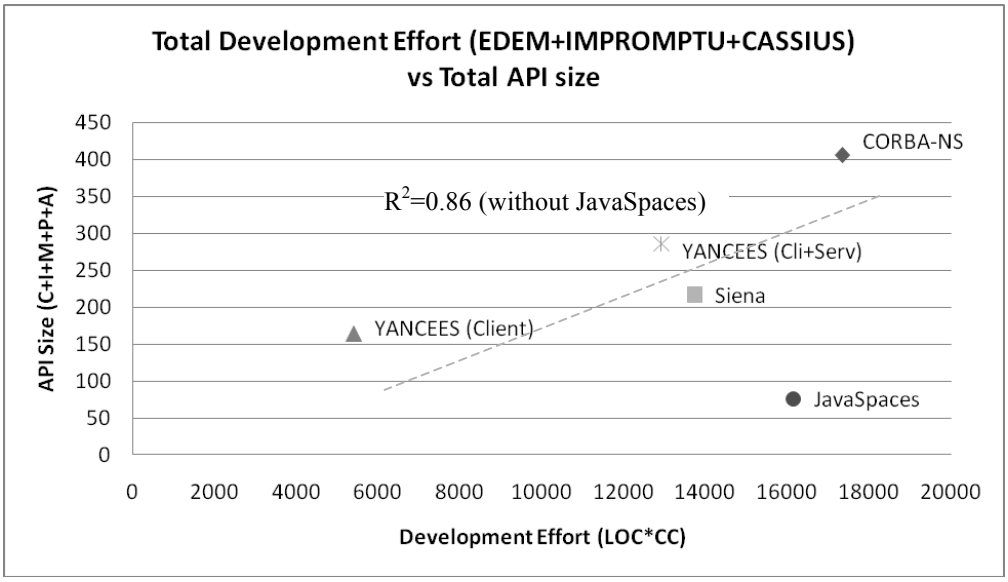


Figure 56 API size versus total development effort (considering IMPROMPTU, CASSIUS and EDEM case studies)

JavaSpaces has the lowest API size of all infrastructures analyzed. In spite of that, its development effort was one of the highest. This exception to the rule is a consequence of subscription and notification models semantic mismatches. Factors such as: the lack of push notification, the fact it does not include tuples in notifications, and the lack of numeric comparators in the tuple space filtering model increase the client side development effort. This shows that semantic mismatches can have a deeper impact on the development effort, than the infrastructure API size by itself.

If mismatches can significantly increase the development effort, closer matches to the application domain requirements can significantly decrease this effort. Case studies developed on top of YANCEES (Client) had the lowest development effort due to better match between required (application domain) and provided (infrastructure) functionality. Moreover, in the particular case study involving CASSIUS, YANCEES (Client) provides a much simpler API, since it does not use object-based subscriptions, which dispenses the need for manually building subscriptions.

In order to further analyze the impact of the API size on the development effort, we separated the total development effort components: code length (LOC) and complexity (measured inMcCabe's cyclomatic complexity), as shown in Figure 57 and Figure 58. The goal was to analyze any possible divergences or convergences between these two metrics.

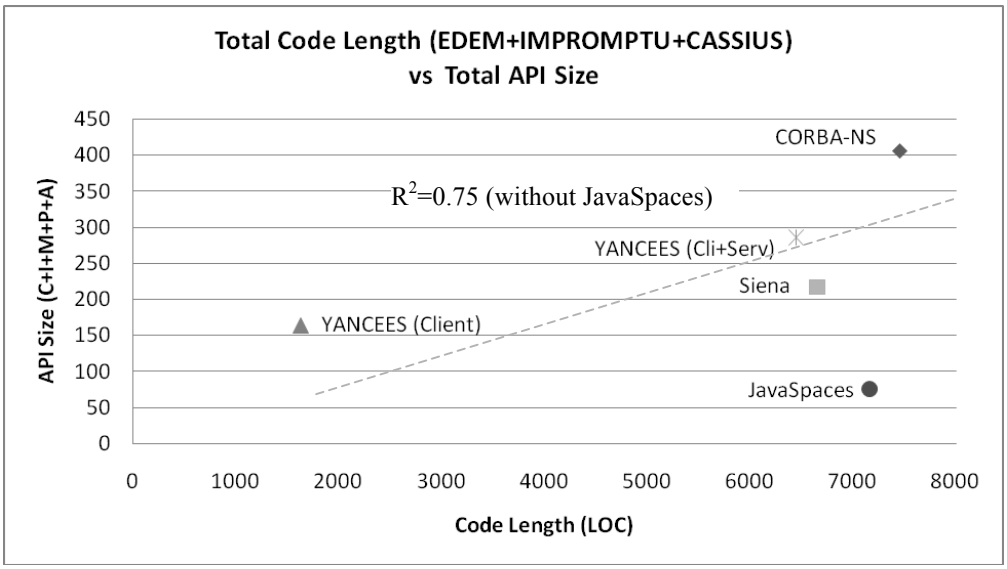


Figure 57 API size versus total client code length (considering IMPROMPTU, CASSIUS and EDEM case studies)

When comparing the overall "shape" of both charts, we see that complexity and LOC follow a similar trend for the systems analyzed (R^2=0.75, R^2=0.82 respectively). This confirms our previous observations (from Figure 56) that development effort grows with the API size. Not only YANCEES (Client) has the lowest LOC, but also the lowest average code complexity. This comes again from its ability to match the required application domain features, in particular, the subscription language commands, increasing the signal-to-noise ratio of the API.

CORBA-NS-based case studies had both the highest complexity and LOC due to its high API size, and lack of configuration management automation.

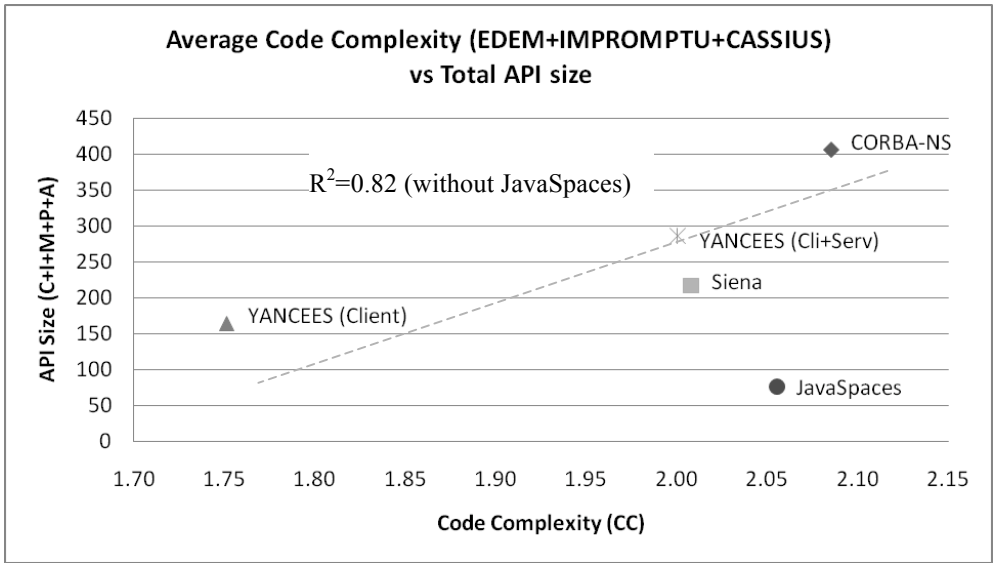


Figure 58 API size versus average client-side code complexity (considering IMPROMPTU, CASSIUS and EDEM case studies)

Conclusions. The API size and semantic mismatches are two important factors that impact the software development effort. In the absence of semantic mismatches, smaller APIs usually

result in lower development efforts. In particular, a small API size, that closely matches the application domain requirements, can significantly decrease the application development effort. If semantic mismatches exist, the development effort tends to increase disproportionally to the size of the API. In other words, semantic mismatches have a larger impact on development complexity than API size.

7.2.2 Textual versus object representation of subscriptions

One important part of a publish/subscribe infrastructure is its subscription language. The process of posting and removing subscriptions represents one of the most common tasks users perform with the infrastructure. In this section, we compare the impact of object versus textual subscription representations in the total application development effort when reusing the selected infrastructures.

In our case studies, both EDEM and IMPROMTPU reference APIs utilize object-based subscriptions, whereas CASSIUS API relies on text-based subscriptions.

In Figure 59, we plot the total tasks development effort of EDEM case study (the sum of the values presented in Figure 38), with the total API size of the infrastructures we analyzed. We do a similar comparison at Figure 60, where we plot the total tasks development effort for CASSIUS case study (the sum of the values presented in Figure 39). In considering these two case studies, our goal is to compare the overall impact of textual versus object representations in the total development effort of most common publish/subscribe operations.

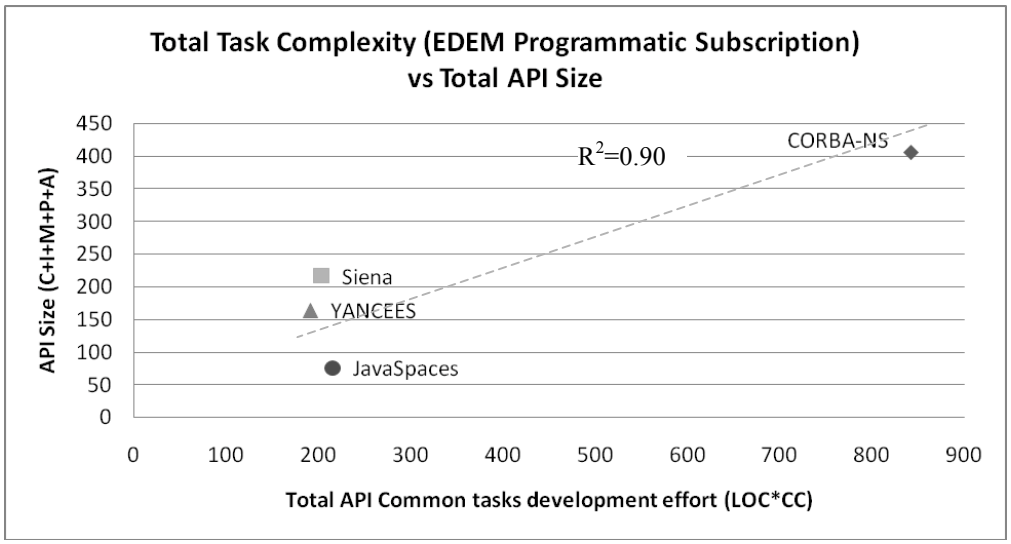


Figure 59 The relation between API size and the total task complexity for EDEM case study

When comparing Figure 59 and Figure 60, and the *createSubscription()* costs of Figure 38 and Figure 39, we verified that application domains that relied on object-based subscriptions (EDEM and IMPROMPTU) required lower adaptation costs (convert from application domain to infrastructure-specific infrastructures subscription) than application domains that rely on textual-based subscriptions (as the case with CASSIUS). This is caused by adaptation costs: the need for parsing these subscriptions into the native formats supported by each infrastructure. Note that these parsing costs exist regardless of the subscription format supported by the publish/subscribe infrastructures.

The only exception to this fact was YANCEES. Its native support for XML processing allows it to take advantage of the XML-based subscription format required by CASSIUS. When using YANCEES in support of CASSIUS, plug-ins were developed to match CASSIUS subscription commands, delegating the task of automatically allocating subscription commands to the infrastructure. The results are large savings in adaptation costs.

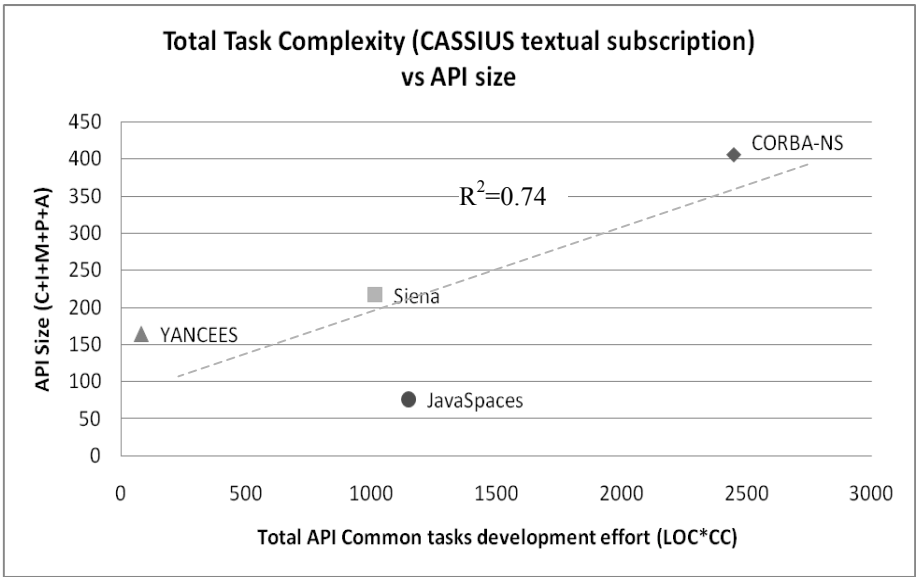


Figure 60 The relation between API size and the total task complexity for CASSIUS case study

Conclusions. Textual representations are only advantageous when adequate automation is provided. In particular, they work better if the application domain is text-based and the infrastructure supports extensibility in its subscription language, as the case with YANCEES. For other application domains, the use of object-based representations, while require manual (programmatic) assembly of subscriptions, result in lower adaptation costs (the costs of converting application domain object- or text-based expressions into the native subscription format of the infrastructure).

7.2.3 Impact of API size on client code maintainability

We also investigated the impact of the API size on the maintainability of the code developed when reusing the infrastructure. The results are shown in the chart of Figure 61. As seen in this chart, systems with heterogeneous API sizes, result in client code with similar modularity.

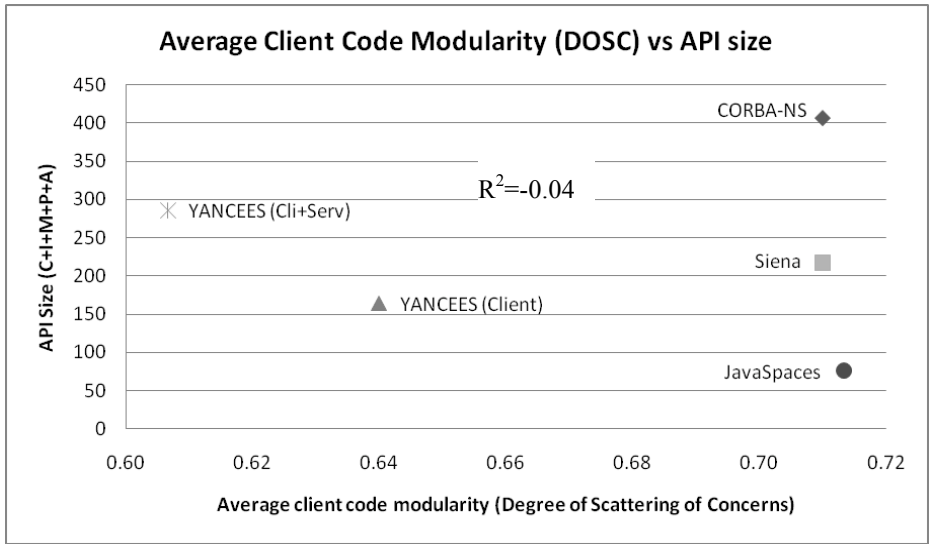


Figure 61 Average client code modularity versus total API Size for the three case studies (IMPROMPTU, CASSIUS & EDEM)

Conclusions. Based on Figure 61, we conclude that there is no linear correlation between the client code modularity and the API size. Maintainability is a function of the separation of concerns (and therefore the modularity) of the code. It is not directly dependent on the size or expressiveness of the API reused in the production of the application. Large APIs may mean a large set of features, but they do not imply high or low maintainability.

7.3 Infrastructure reusability & client code maintainability trade-offs

We further analyze the relation between client code maintainability (as a function of the average client code modularity) and the total infrastructure reusability (as a function of the total development effort) of our case studies. The results are shown in Figure 62.

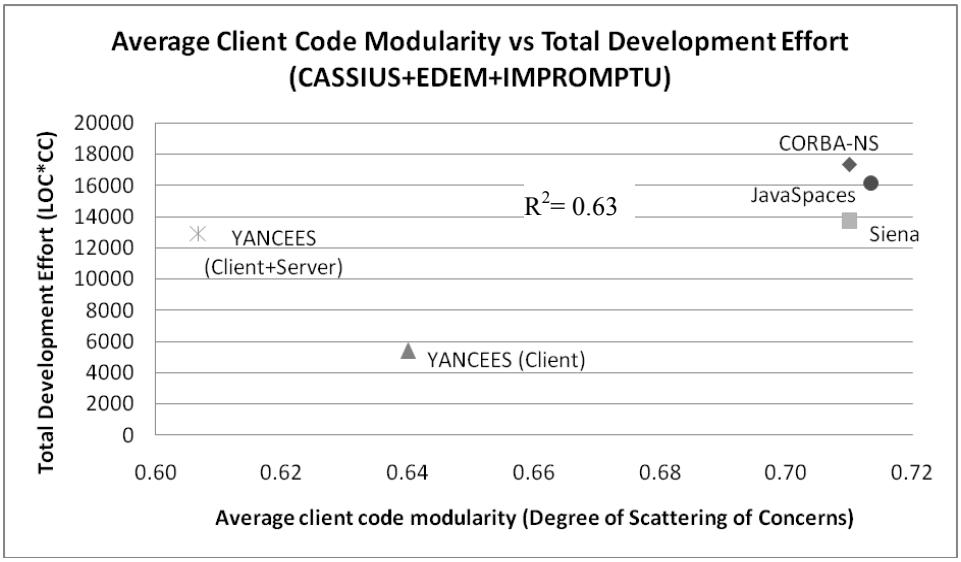


Figure 62 Relation between development effort, when reusing the infrastructures, and client-side code modularity

As seen on Figure 62, there is a clear separation between infrastructures reused as black boxes (CORBA-NS, JavaSpaces and Siena) and YANCEES, that is reused as a grey-box. Since YANCEES was designed to be extensible around the main publish/subscribe concerns, the extensions implemented on YANCEES server side are very modular. Moreover YANCEES configurability and extensibility, allows it to better fit the application domain requirements, resulting in more simple and modular client code.

Conclusions. Flexible approaches not only can reduce the development effort over successive reuses, but also results in more maintainable (modular) code than existing black box approaches. Moreover, if considered only the client code development effort, this advantage becomes more expressive, with larger gains of development effort.

7.4 Performance trade-offs

Performance is an emerging software quality that depends on different factors such as: programming language, architectural decisions (use of multithreading and infrastructure protocols, for example), and the adoption of specialized algorithms. In this section, we study the impact of different factors on the performance of the infrastructures we analyzed.

7.4.1 Relation between development effort and performance

As seen in our case studies, the abstraction distances between provided (infrastructure) and required (application domain) functionality is proportional to the development effort required to extend, configure and adapt an infrastructure to a new context.

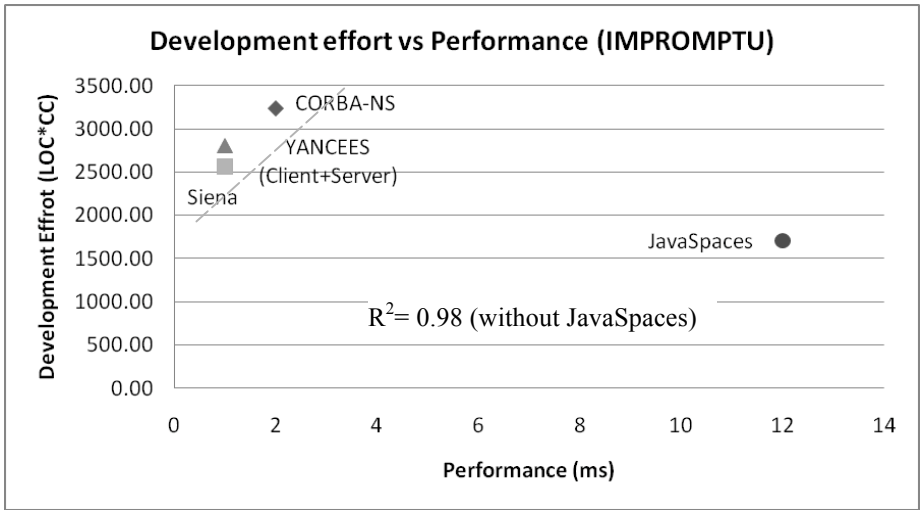


Figure 63 Development effort versus performance for the IMPROPTU case study

Other factors, however, may have a higher impact on performance than adaptation costs. In the IMPROMPTU case study, shown in Figure 63, the determinant performance factors were the delays associated to communication protocols of CORBA-NS (ORB marshaling/un-marshaling costs) and the algorithms and strategies devise to bridge the semantic mismatches of JavaSpaces. Hence, even though JavaSpaces has the lowest development effort, it did not perform as well as the other infrastructures.

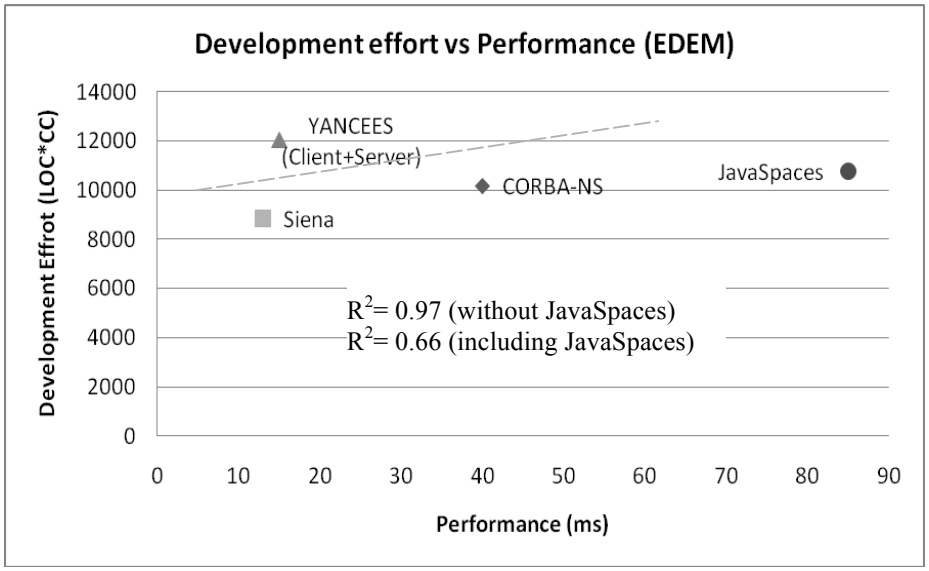


Figure 64 Development effort versus performance for the EDEM case study

The same is true for the EDEM case study shown in Figure 64, JavaSpaces mismatches and communication protocols make it the slowest of the infrastructures, while CORBA-NS performance gets jeopardized by its internal algorithms and communication protocols.

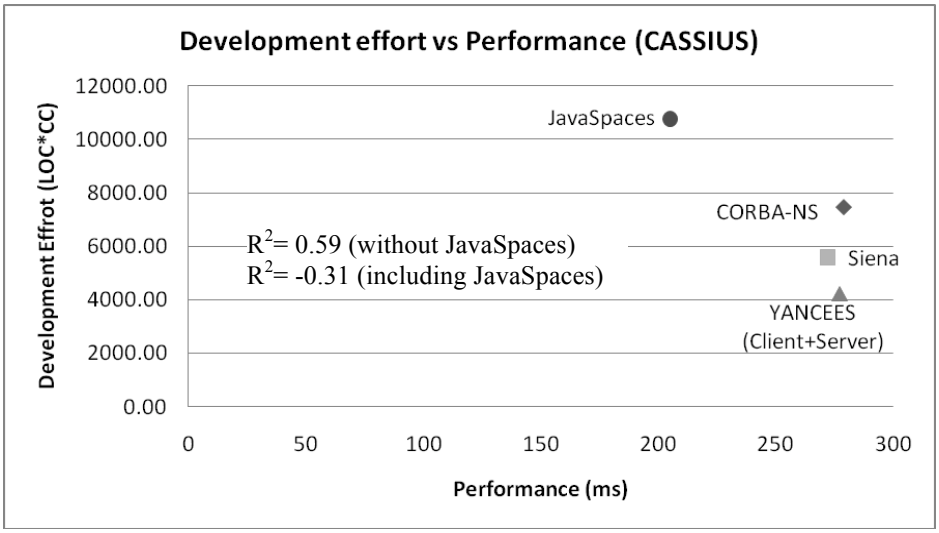


Figure 65 Development effort versus performance for the CASSIUS case study

In Figure 65 we see JavaSpaces outperforming existing infrastructures. This confirms the role of semantic mismatches in the performance of the infrastructure that was observed for IMPROMPTU and EDEM.

In Figure 66, we compare the total performance delays of our three benchmarks with the case studies total development effort.

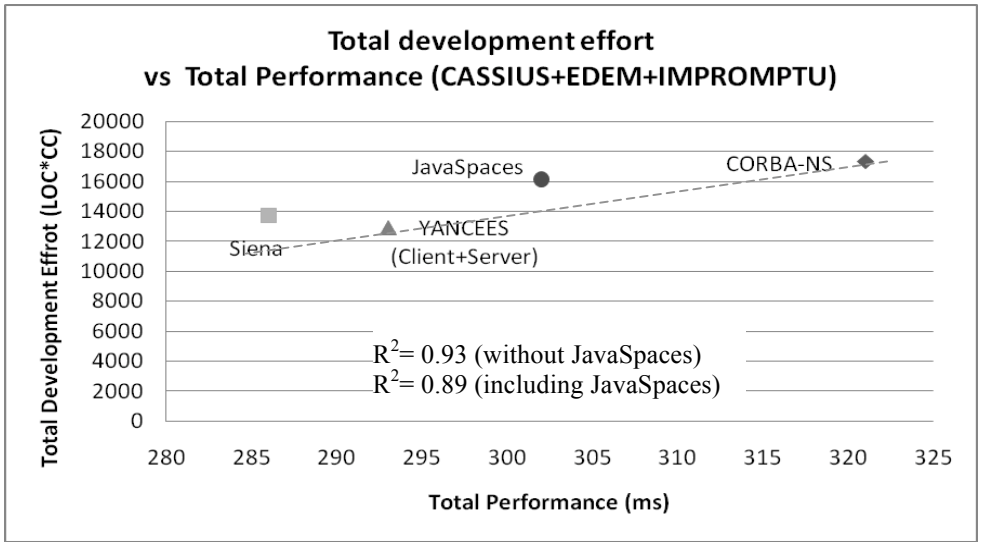


Figure 66 Total development effort versus total performance delay for the three case studies: CASSIUS, EDEM and IMPROMPTU

As shown in Figure 66, from all the case studies, those built on top of YANCEES and Siena were very close to each other in terms of performance and development effort. However, case studies built on top of Siena had better performance. This difference comes from factors such as YANCEES internal adaptation costs due to event generalization, the use of RMI, and the overhead of the framework itself. Siena's protocol implementation based on Sockets, and its lack of design for change, significantly improve its performance.

Conclusions. Overall, based on the chart of Figure 66, we see that the performance delays are generally proportional to the total development effort employed in the reuse of existing infrastructures. We also observed that factors such as semantic mismatches, protocols, and algorithms can have more significant impact on the performance than simple adaptation costs alone.

7.4.2 Relation between client code modularity and performance

Another interesting question to ask is if there are any important correlation between the modularity of the code (which is related to maintainability) and its performance. By plotting the total case studies performance with the average code modularity, we obtained Figure 67.

As seen on Figure 67, case studies with similar low modularity such as the code built on top of: Siena, JavaSpaces and CORBA-NS have different performance delays; whereas case studies built on YANCEES presents low performance and high modularity.

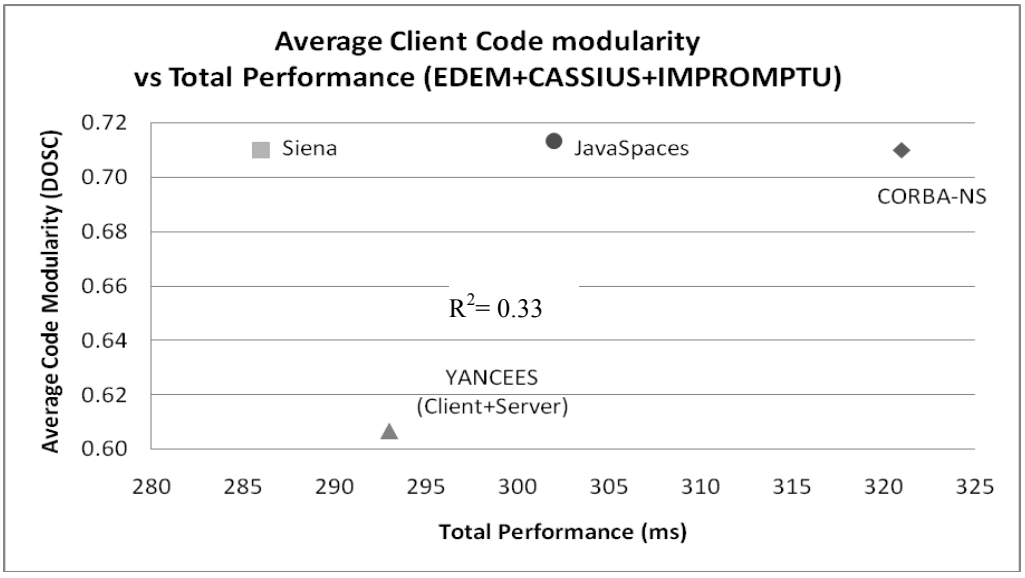


Figure 67 Average client code modularity versus total performance of the three case studies: EDEM, CASSIUS and IMPROMPTU

Conclusions. Based on the result of Figure 67 (R^2= 0.33), we see no particular correlation between client code modularity and performance.

7.4.3 Impact of API size on case studies performance

Another question that we seek to answer is if there were any correlation between the API size of the infrastructures and their performance (responsiveness) in our benchmarks. As shown in Figure 68, JavaSpaces, which has the lowest API size, did not perform well in these two case studies.

If we set JavaSpaces apart, for a while, we see that as the API size increases, there is a slight decrease in the responsiveness of the infrastructure (with R^2= 0.98). A larger API may indicate many options (as in CORBA-NS) or more complex features (extensibility interface of YANCEES), which usually results in performance penalties to the system.

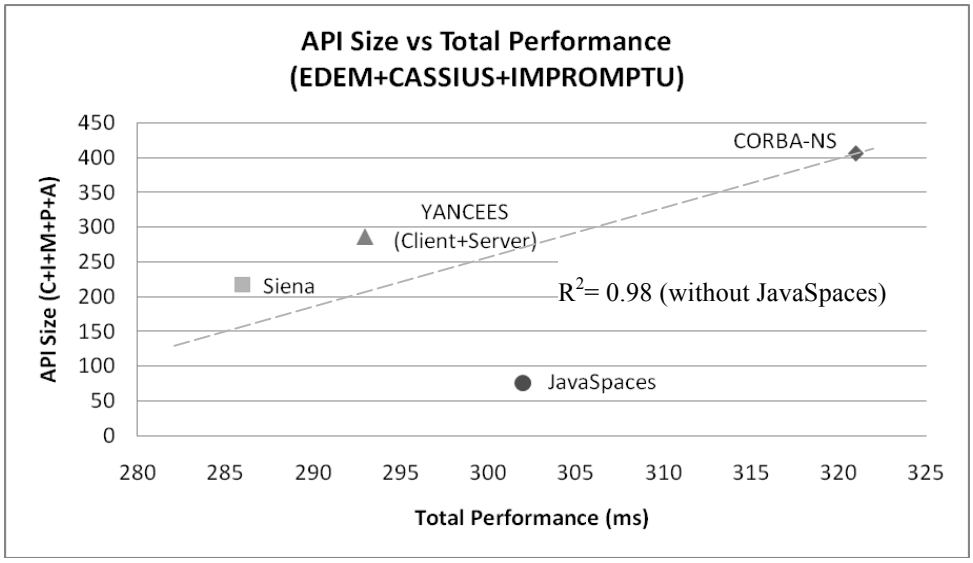


Figure 68 API size versus total performance (response delays) for the IMEDEM, CASSIUS and IPROMPTU case study

Conclusions. Similar to what was observed in Figure 60, the size of the API is proportional to the total performance delay experienced by the infrastructures under regular conditions. However, semantic mismatches have higher impact on performance than API size alone, making possible for systems with larger APIs to outperform more simple but incompatible infrastructures.

7.4.4 Performance trade-offs conclusion

With respect to performance, semantic mismatches and architectural decisions such as communication protocols and algorithms are more important than modularity, development effort or the size of the API. As such, developers must prioritize their selection criteria in terms of these factors; whereas infrastructure developers must pay extra attention to the application domain fundamental requirements as a way to prevent mismatches.

7.5 Trade-offs summary

In this chapter we analyzed the correlation (and lack thereof) between different software qualities. In particular, we found direct correlations between:

- API size and dev. effort (LOC*CC), including LOC and CC alone
- API size and performance
- Infrastructure flexibility and client code modularity
- Abstraction distance and development effort
- Development effort and performance

We found no direct correlation between the following software qualities, for the infrastructures analyzed, and the case studies we performed:

- Infrastructure modularity and its flexibility
- Public API size and client code modularity
- Case study performance and client code modularity

Overall, semantic mismatches have high impact on performance and development effort. In the particular case of subscription representation, we found that textual representations are only advantageous when adequate automation is provided. In all other situations, programmatic subscription representations are easier to adapt.

Chapter 8. Principles and Guidelines

Based on the software engineering literature, the lessons learned in the analysis of versatility trade-offs, on our case studies, and on our own experience in the development of YANCEES, we present in this chapter a set of guiding principles for the basic activities involved in the development (requirements analysis, design and implementation), and reuse (selection, adaptation, extension) of versatile software. In doing so, our goal is to help infrastructure developers in choosing the most appropriate versatility strategies for the infrastructures they develop; and infrastructure users in selecting the most appropriate infrastructure for their needs.

8.1 Requirements recommendations

As discussed in section 2.2, factors such as problem domain and configuration-specific dependencies, as well as the implicit trade-offs between different software qualities are important factors that limit software versatility. These earlier these factors are detected, documented and analyzed, in the software engineering process, the cheaper it is to correct the design and adopt measures to minimize their impact in the overall versatility of the infrastructure. Hence, in the analysis of requirements, we propose the following recommendations:

8.1.1 Consider the problem domain through multiple perspectives

There is a need to consider different factors in the design of a versatile system. In middleware analysis and design, performance tends to be the prime factor that guides the design decisions of the infrastructure. As shown in this dissertation, the price paid for good performance is many times poor APIs usability and reusability.

As discussed in Chapter 8, both API size and the complexity of common infrastructure API use cases can significantly impact the performance and the development effort of the application. Moreover, if semantic mismatches are not detected early in the design and reuse of an infrastructure, these costs may scale.

By understanding both the application domain requirements and the different trade-offs these design decisions define, developers can minimize the costs of development and reuse of the software.

Before these factors can be addressed, however, they must be made explicit. Throughout this paper, we showed how these factors are many times inter-related and how the combination of design decisions can lead to a favorable balance between different software qualities.

8.1.2 Perform an analysis of domain-specific dependencies

As previously discussed, the different publish/subscribe infrastructures concerns are many times non-orthogonal. They have implicit control and data dependencies that limit set of possible

variants the domain can support. As a consequence, the earlier these dependencies are detected, the better designers can choose between existing approaches to limit the effect of change in software.

In the software product line engineering domain, the analysis of commonality and variability in the design of versatile infrastructures is a common practice (Coplien, Hoffman et al. 1998). This analysis, however, is many times incomplete. It does not make explicit the different data and control dependencies that exist between features and variability dimensions.

In particular, we propose an approach to document these dependencies described at (Silva Filho and Redmiles 2006), where a UML notation shown in Figure 9, is used to make these dependencies explicit. With a good understanding of these dependencies, designers can opt, for example, to stabilize and generalize some design dimensions, thus preventing the costs of change associated to core system characteristics.

8.2 Design and implementation principles and guidelines

The design of versatile software requires the proper management of the trade-offs discussed in previous sections. In this section, we present a set of design principles that can be used to achieve a favorable balance between different software qualities that characterize a versatile infrastructure.

8.2.1 General design principles

General design principles are those that must be applied in the development of versatile software in general. In other words, they are not restricted to publish/subscribe infrastructures alone. These are: abstraction, modularity, (de)composition, and simplicity.

8.2.1.1 Abstraction

Abstraction is the design strategy used to hide unnecessary implementation details from software users, exposing only the necessary functionality, required for software reuse. The combination of abstraction (Liskov and Zilles 1974) and modularity (Parnas, Shore et al. 1976) is the way proposed by Parnas to support the development of large systems (Parnas, Clements et al. 1984), and to support software flexibility (Parnas 1978).

In the particular case of versatility, abstractions can be flexibilized to better match application domain requirements (as in YANCEES), and can be used in the definition of generalizations that capture the variability of a domain. For example, generalized even and subscription representations as adopted by the infrastructures we analyzed.

8.2.1.2 Modularity

Modularity is a general principle that must be applied in the construction of any kind of versatile software. Modularity implies in separation of concerns in the implementation of software as the composition of simpler parts (or modules) that are connected by clean interfaces. Separation of concerns as the criteria for modularization (Parnas 1972) supports design for change, whereas clean interfaces imply minimal modules, which tames complexity.

The decomposition of complex systems into modules is supported by different principles such as: The Single Responsibility Principle: *"A class should have only one reason to change"* (Martin 2003); The Open-Closed Principle: *"Modules should be open for extension but closed for*

direct code change" (Meyer 1997); the Liskov Substitution Principle: *"Subclasses should be substitutable for their base classes."* (Liskov 1987); and Design by Contract *"...when redefining a routine [in a derivative class], you may only replace its precondition by a weaker one, and its post-condition by a stronger one."* (Meyer 1992);

In the publish/subscribe domain, one can modularize concerns around the common process of publication and subscription of events. In particular, subscription commands can also be modularized, allowing their composition into more complex expressions according to the process trellis architectural style (Factor 1990).

8.2.1.3 (De) Composition

The power of modularity and abstraction can only be leveraged through the decomposition of complex systems into modular concerns, followed by their re-composition into working systems. In other words, complex systems are built as a composition of modules, implementing well defined abstractions (Parnas, Clements et al. 1984).

The composition of modules in the construction of systems is supported by different principles such as: the Dependency Inversion principle (a.k.a. Inversion of Control): *"(a) High-level modules should not depend on low-level models. Both should depend on abstractions and (b) Abstractions should not depend on details. Details should depend upon abstractions."* (Martin 2003; Fowler 2004); the Interface Segregation Principle: *"Clients should not be forced to depend on methods that they do not use"* (Martin 2003); and the Law of Demeter: *"Talk only to your friends"* (Lieberherr and Holland 1989).

Flexible infrastructures as YANCEES adopt decomposition around major publish/subscribe concerns, better supporting the development of application-specific infrastructures.

8.2.1.4 Simplicity

A design must be as simple as possible. Complexity must be added only when strictly necessary. A practical implication of this principle is to avoid one-size-fits-all or bloated implementations, ones with features that are rarely or never used.

For every new feature that an infrastructure must support, there is an increment in its complexity. As shown in our studies, and based on our own experience, the dependencies between features that must be supported together, negatively impact the quality of individual features of the set. Quantitatively speaking, and based on the literature, for each 25% increase in problem complexity, there is a 100% increase in the solution complexity (Woodfield 1979).

In the particular case of publish/subscribe infrastructures, the more variability or configurability one provides, the more complex the implementation becomes, requiring measures such as automation, generalization, and configuration management to support developers in dealing with the complexity of the infrastructure.

After presenting the most basic set of software engineering principles, we proceed to discuss principles that more specific to the design of versatile software.

8.2.2 General versatility design principles

The design of versatile software infrastructure must balance two major forces: *unpredictability*, the need to support unforeseen requirements, and *fitness*: the need to support the exact sets of features of an application domain. These two requirements are usually conflicting. Fitness requires specialization and simplicity, which usually requires the elimination of irrelevant features to the problem domain. Unpredictability leads designers to adopt versatility approaches as gener-

alization, variation and flexibility, which performance, and development effort costs we discussed throughout this paper.

As a consequence, developers must choose among a large pool of design options, for example: should one support unpredictability through variability (with configurability), extensibility or generality? Should one adopt a hybrid approach? The following principles helps in the choice process considering these options.

8.2.2.1 Ockham's Razor

This principle states that given a choice between functionality equivalent designs, one must select the simplest design. This principle builds upon the simplicity design principle, and comes from the fact that unnecessary elements in a design, decrease its efficiency, usability, performance, and may lead to unanticipated consequences, for example, feature interaction (Silva Filho and Redmiles 2007).

As seen in our study, infrastructures that support optional features such as JavaSpaces and CORBA-NS suffer from costs associated to features that are not necessary for the task at hand. In the case of JavaSpaces, the handling of exceptions produced by features such as leasing, transactions and access control increase the complexity of software, even though they are not used in our studies. Likewise, the costs of configuration of CORBA-NS, results in complex and lengthy code, which usually increases the development effort of the whole infrastructure.

Hence, according to this principle, one should adopt either minimal core approaches as Siena, or flexible approaches as YANCEES. In particular, we found that flexible approaches should be applied in situations where one can afford two software teams: one that customizes the infrastructure, and other that reuses customized infrastructures in the development of different applications. The application-specific infrastructures produced through the reuse of flexible approaches can significantly reduce the application development effort, addressing the application performance requirements. However, the development effort and learning curve involved in the adaptation and extension of flexible infrastructures can be high. This costs can be amortized through the use of software development teams that knows the internals of the infrastructure and take advantage of the reusability of its components in the production of slightly similar systems out of a common set of reusable assets (Bockle, Clements et al. 2004).

Minimal core generalized infrastructures such as Siena should be used in situations where there are no semantic mismatches and the abstraction distance between the provided infrastructure and required application features is not high. If these conditions are match application developers can implement any additional feature required, resulting in development efforts that are better or comparable to extend and reuse flexible infrastructures.

8.2.2.2 Satisficing designs

It is often preferable to choose a satisficing solution (good enough given the problem constraints), rather than an optimal one. This comes from the fact that the costs of optimality are usually excessive complexity. Hence, from the point of view of application users, the choice of the simplest solution is preferable, whereas from the point of view of the infrastructure developers, one must opt for satisficing infrastructure designs (Simon 1996). In other words, in many circumstances it is better to produce a design that roughly satisfy the requirements, but produce simpler but good solutions to a problem, than to produce a design that optimally satisfies all the requirements, at the expense of excessive complexity.

For example, YANCEES' initial design strived to support event format variability. This choice, whereas increased the infrastructure scope, resulted in an increase in configuration man-

agement complexity. Moreover, the support for event format variability resulted in incompatibilities with existing plug-ins, decreasing their reusability. By opting for a generalized, but fixed, event format, YANCEES achieved a satisficing trade-off between event variability and infrastructure complexity as illustrated in Figure 69.

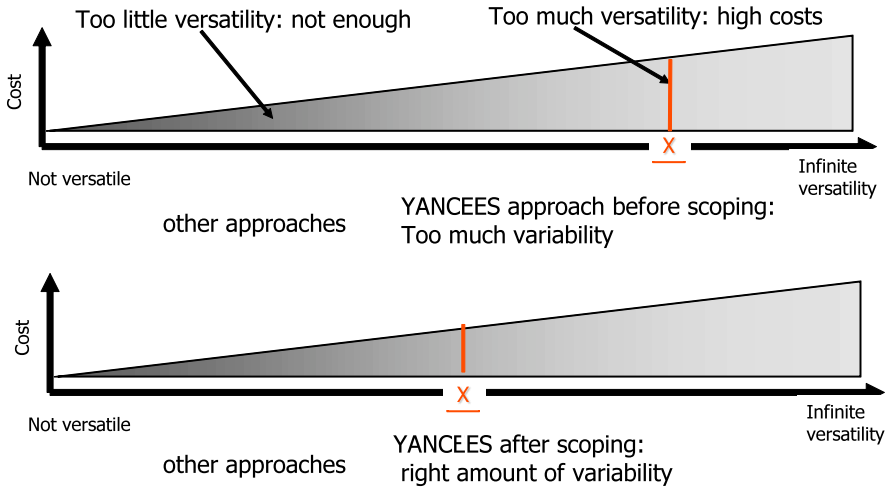


Figure 69 Scoping down YANCEES variability to improve its versatility

This approach is coherent with what Richard Gabriel calls "New Jersey" approach to design (Gabriel 1991), where a worse (or satisficing solution) is better than an optional, but complex one.

8.2.3 Publish/subscribe versatility common strategies

In spite of the differenced in the versatility approaches discussed throughout this work, we found a common set of versatility strategies adopted by the different infrastructures we analyzed. More specifically, we found a convergence towards the use of generalized event representations, independent and compositional subscription commands (or plug-ins), switchable routing strategies (CORBA-NS and YANCEES), and compositional approaches to handle configurability and extensibility. Convergence usually indicates optimality of solutions to common problems in a domain (Lidwell, Holden et al. 2003). In our case, they represent good design principles that can be observed in the development of versatile infrastructures, addressing issues related to the lack of orthogonality of the different publish/subscribe design concerns, and the inter-dependencies between different software qualities. We further explain these design decisions as a set of principles as follows.

8.2.3.1 Composition of subscription commands

Event processing languages are usually expressed in terms of filters, sequence detectors, and rules. These commands represent increasing levels of abstraction, expressed in terms of lower-level filters. In other words, higher-level commands such as rules depend on features provided by lower-level features as filters. This characteristic allows subscription commands to be both modular and compostable, thus improving their reusability. All infrastructures analyzed rely on this characteristic. Siena supports pattern detection based on the composition of lower-level Filters; JavaSpaces provide primitive tuple manipulation commands that are composed in the implementation of more complex features. Infrastructures such as CORBA-NS and YANCEES support text-based subscriptions. They implement commands as independent components that are automatically combined in the execution of complex subscription expressions.

This application domain characteristics supports the reusability of subscription commands and the incremental development and reuse of infrastructures, and the dynamic allocation of plug-ins in YANCEES.

8.2.3.2 Switchable routing strategies

Both CORBA-NS and YANCEES support the ability to select between different routing strategies. This ability, which is according to the strategy selection open implementation guideline (Kiczales, Lamping et al. 1997), allows the use of the most appropriate algorithm to the rouging requirements at hand, improving the system performance.

8.2.3.3 Generalization of event representation

Generalization, instead of variation, should be applied on design dimensions which changes can impact core design concerns. In the publish/subscribe domain, the most expressive example is the event format. All infrastructures analyzed adopt, in different degree, generalized event representations.

For example, CORBA-NS supports event representation variability by adopting a common internal event representation: the *StructuredEvent*, which is general enough to encapsulate all other event representations it supports. Both Siena, and YANCEES adopt attribute/value pair formats, whereas JavaSpaces supports general programming language objects, considering their attributes as tuples.

8.2.4 Flexibility design principles

As shown in our work, flexible approaches as YANCEES can achieve a favorable balance between the different versatility qualities we measured. In order to reap these benefits, however, the design of flexible infrastructures must adequately tame the complexity that comes from extensibility and configurability. In particular, we propose the following recommendations in the design of flexible software.

8.2.4.1 Support separation between mechanisms and policies

Flexible approaches are usually supported by designs that separate commonality and variability. A good way to reuse commonality and support variability is to separate and modularize these concerns into policies and mechanisms (Wulf, Cohen et al. 1974). As such, policies are used to represent variable features, whereas mechanisms are used to capture the commonality of the domain.

For example, in YANCEES, plug-ins implement application-specific features (policies), whereas the core infrastructure supports the common publish/subscribe process. This separation allows the construction of application-specific infrastructures through the combination of a common publish/subscribe process with application-specific plug-ins.

8.2.4.2 Design for change, supporting extensibility and configurability

Flexibility implies both extensibility and configurability. Any flexible approach must provide mechanisms that support these two characteristics. As seen in our case studies, flexibility can only be achieved on planned variability dimensions. This therefore requires the design for change along planned variability dimensions, and extra support for configuration management.

For example, CORBA-NS supports configurability through the use of factories and configuration interfaces. YANCEES supports configurability through runtime parsers and static configuration managers, whereas provides extensibility through plug-ins and extensible languages.

JavaSpaces represents a counter-example. It is not designed to support publish/subscribe interaction specifically. As a consequence, it is not modular toward common publish/subscribe variability dimensions such as notification model and subscription language. This lack of design for change according to these dimensions, resulted in high costs of adaptation and performance penalties.

8.2.4.3 Support late binding of features

A common way to support unpredictability is to defer design decisions to as late as possible in the design process. Late binding supports runtime activation and composition of features, in response to immediate changes in the application requirements.

Both CORBA-NS and YANCEES support the concept of late binding. While CORBA-NS allows users to select among existing features and to create new event channels, and producers and consumers proxies; YANCEES allocates plug-ins according to subscription language expressions posted at runtime. Late binding allows the allocation of resources only when necessary, besides supporting the runtime configuration of different policies.

8.2.4.4 Provide architectural reflection

It is usually the case that complex features are not implemented by a single component. Instead, they are built as a composition of extensions that need to communicate with one another. Reflection allows features to find each other at runtime. It also supports automatic configuration management, allowing the infrastructure to detect incompatible configurations.

Both CORBA-NS and YANCEES provide architectural reflection. CORBA-NS allows the location of active proxies through the use of architecture managers; while YANCEES supports reflection through the use of a plug-in register.

8.2.4.5 Adopt customizable and modular abstractions

Changes in the features provided by the infrastructure must be reflected in its public API. Customizable and modular abstractions allow the infrastructure to change its public API to better match the application domain requirements. In doing so, it relieves users from unnecessary implementation details, and avoid their exposure to options and commands that are not required by the target application domain. As a results, there is an increase in the signal to noise ratio of the system, reducing the client-side development effort.

While CORBA-NS supports different abstractions for the policies it supports (for example: pull and push producers and consumer policies), it does not isolate users from configuration details. YANCEES is the only infrastructure that supports the development of application-specific subscription languages, isolating configuration and extension concerns from end-users, achieving lower client-side development effort costs.

8.2.4.6 Employ automation to improve usability

Customizable and modular abstractions are supported by extensibility, configurability and automation. While extensibility and configurability allows the representation of domain-specific concerns that closely match the application needs, automation hides from end-users the process of expressing these higher-level abstractions in terms of more primate ones.

When comparing YANCEES and CORBA-NS, the automation and abstraction provided by YANCEES can significantly reduce the client-side application development effort, while still keeping the configurability of the infrastructure.

8.2.5 API usability design guidelines

As shown by our case studies and analysis of trade-offs, the design of a usable API can significantly reduce both the development effort while increases the performance of the infrastructure. This section discusses some API design principles derived from our case studies and experience in the design of YANCEES.

8.2.5.1 Strive for minimalism and completeness

An API should provide an essential set of features that support the common requirements of the majority of applications. Large APIs with lots of convenience functions are rarely used and have the potential for increasing the infrastructure complexity. Incomplete APIs result in mismatches and extra development effort for the infrastructure clients. Hence, the bottom limit of minimalism is completeness. Completeness assures that the API supports all the common requirements of a domain. For example, in JavaSpaces, the lack of numeric comparators in the anti-tuple model shows the incompleteness of the tuple-space model to the set of application domain requirements we supported.

8.2.5.2 Support multiple user roles by separating API concerns

An infrastructure should support different APIs according to the needs of different stakeholders, thus, reducing the API complexity, and increasing its "signal-to-noise ratio". In other words, an API should hide from different types of users, concerns that do not belong to their common tasks. This strategy goes in line with the open implementation design guidelines (Kiczales, Lamping et al. 1997), and the minimalism of APIs (Henning 2009). Moreover, it recognizes the needs of different stakeholders (infrastructure developers and consumers), besides of reducing both API size and individual task efforts. For example, the separation between configuration, extension and regular publish/subscribe APIs in YANCEES had a positive impact in the reduction of the client-side development effort.

Separation of API concerns, however, should be balance with possible integration costs. For example, in the case of YANCEES, that separates protocol and publication APIs, further communication between protocol, publication and subscription plug-ins was required in order to uniquely identify the subscriber. These extra costs were not needed in the other case studies, since there were no such separation.

8.2.5.3 Support API customizability

As previously discussed in the flexibility design guidelines, the ability to tailor an API to the end-users' needs can have a dramatic impact on the reusability and usability of an infrastructure. From the analyzed infrastructures, YANCEES is the only one to explicitly support the complete redefinition of its subscription and notification languages. This ability has shown to significantly reduce the adaptation costs (more dramatically in CASSIUS case study), and prevent semantic mismatches.

8.2.5.4 Minimize user choices

While choices support configurability, and the ability to support different application domains; they reduce the usability of the system when supporting individual application domains.

Users do not want to pay the price for extra API complexity coming from features that they do not require. An ideal API provides only the necessary functionality. No more, no less.

8.2.5.5 Minimize adaptation

From our case studies, a large amount of code is devoted to the adaptation of data and control formats to and from different application domains. These costs only contribute to the decrease of system usability and reusability.

A good approach to reduce adaptation costs is the use of application-specific languages and APIs as demonstrated by YANCEES.

8.2.5.6 Give preference to object-based subscription formats

As described in 7.2.2, textual representations are only advantageous when adequate automation is provided. In particular, they work better if the application domain is text-based and the infrastructure supports extensibility in its subscription language, as the case of CASSIUS over YANCEES. For other application domains, the use of object-based representations, while require manual (programmatic) assembly of subscriptions, result in lower adaptation costs (the costs of converting application domain object- or text-based expressions into the native subscription format of the infrastructure).

8.2.6 Maintainability principles

In our case studies, we correlate maintainability with software modularity. By analyzing the selected infrastructures, we observed an improvement in maintainability when design for change along main publish/subscribe variability dimensions was adopted.

8.2.6.1 Design for change

As seen on section 7.3, the design for change along the main variability dimensions of the application domain, applied in the construction of flexible software, results in more maintainable (modular) client code. Since changes along these dimensions are more likely to occur, a design for maintainability should follow the same design of flexible software, modularizing and composing concerns along these dimensions.

8.3 Reuse recommendations

From a reuse perspective, *"the defining characteristic of good reuse is not the reuse of software per se, but the reuse of human problem solving. (...) Reuse multiplies the effectiveness of human problem solving by ensuring that the extensive work or special knowledge used to solve specific development problems will be transferred to as many similar problems as possible."* (Barns and Bollinger 1991).

As made evident in our case study, the value of an infrastructure rests not only in its ability to support middleware distribution, communication and coordination concerns, but also on its ability to support application-specific requirements. Any extra features that are not necessary for the application represent additional development costs that, in addition of being an unnecessary investment for that particular application, can actually demeanor the value of an infrastructure by adding extra complexity, performance, usability and development costs.

As discussed in section 2.1.2, the process of reusing versatile infrastructures, built according to different versatility strategies, involves different steps comprising: *selection*, *extension*, *con-*

figuration, *adaptation* and *composition (or integration)* (Krueger 1992). In this section, we discuss guidelines to support these operations.

8.3.1 Selection

Selection is one of the most important activities in software reuse. The selection of a good infrastructure will reduce the costs of extension, configuration, adaptation and composition. Two factors are key to the selection of an infrastructure: its *fitness* to the problem domain requirements, and the *absence of semantic mismatches*. As such, one must observe these factors as follows:

8.3.1.1 Avoid semantic mismatches

Semantic mismatches are those that represent deficiencies in the fundamental features provided by an infrastructure. It is easier to reuse infrastructures that require extra adaptation effort on the client side, but have no semantic mismatches in its core, than to reuse infrastructures that are close fits to the application domain, but provide slightly different features that do not completely match the fundamental requirements of the problem domain.

For example, even though CORBA-NS supports pull notification, its semantic was not compatible to that required by CASSIUS (which adopts a message box approach). As a result, a proprietary pull notification module was required to reuse CORBA-NS in CASSIUS case study.

8.3.1.2 On the absence of semantic mismatches, select based on problem domain fitness

When inflexible infrastructures are reused, and no semantic mismatches are found, choose infrastructures which features are closer to the application domain at hand, or which mapping and transformation between required and provided features is trivial. For example, Siena can support a large set of application domains by providing a generalized but adaptable subscription and event representation. Its filter capability and event representation are generic enough to support a large set of application domains.

8.3.1.3 Consider flexible approaches when supporting software product lines

Flexible approaches, even though present higher domain-specific implementation costs, provide lower middleware and adaptation costs. The initial higher component development complexity can pay-off over successive reuses. For example, over three consecutive reuses, YANCEES case study total development effort was lower than the case studies reusing the other infrastructures.

8.3.2 Adaptation

As shown in our case studies, adaptation costs, if not well managed, can significantly impact the performance and reusability of the infrastructure. Two major factors are important in the choice of a publish/subscribe infrastructure, its subscription and event formats, and the fundamental set of features it supports.

8.3.2.1 Consider the predominant event and subscription representations

Adaptation costs are higher when there is a mismatch between these formats. For example, Siena and JavaSpaces had fewer adaptation costs when supporting domains using object-based subscription, while YANCESS and CORBA-NS better supported text-based subscription domains.

8.3.2.2 Consider layered adaptation

The filtering capability of publish/subscribe infrastructures not only supports the modularization of server-side features (as discussed in 8.3.3.1), but also affords the reuse of existing infrastructures through the laying out of code that incrementally refine and reuse existing core functionality. Through this approach, more complex event filtering capabilities can be implemented as a function of more simple filtering mechanisms. This is observed in all infrastructures analyzed.

This layering reuse process also allows the bi-directional translation of existing event formats between application domain and infrastructure-specific formats. It also supports the development of different notification policies, and the independent implementation of user and infrastructure protocols. Due to this domain-specific characteristics, it was possible to reuse all the selected infrastructures, under some complexity and performance penalties as described in section Chapter 5.

8.3.3 Configuration

Configuration is the act of selecting specific sub-sets of features in support of application-specific requirements. Different recommendations can be observed in the process of configuration.

8.3.3.1 Consider configuration management costs

As seen in our case studies, configuration costs are very important. In infrastructures as CORBA-NS can be as high as 20% of the total cost of development; whereas configuration and extension represent 45% of the costs of reuse of YANCEES (most of its value, however, represent extension costs).

Hence, application developers must understand the impact of configuration in the costs of their applications, selecting infrastructures that balance these costs with other benefits.

8.3.3.2 Prefer infrastructures that implement the open implementation design guidelines

The open implementation design guidelines (Kiczales, Lamping et al. 1997) prescribes separation between configuration and normal use APIs, while provide mechanisms to allow users to choose between different strategy implementations. Open implementation allows the selection of application-specific strategies that can improve the performance of the infrastructure as a whole.

From the analyzed infrastructures, only YANCEES supports the separation of API concerns and the automation of the selection process based on subscription commands, making it easier to select valid sub-sets of features. CORBA-NS partially implements the open implementation design guidelines, by allowing users to select among different routing strategies through its configuration API.

8.3.4 Extension

Generally speaking, extension is a costly operation that requires certain amount of comprehension of the inner mechanisms of an infrastructure. In the best case scenario, it involves the use of extension APIs provided by the infrastructure, in the worst case scenario, it involves the understanding and direct modification of the source code of the infrastructure.

8.3.4.1 Consider the costs of extension, preferring approaches that support automation, documentation and enforcement of dependencies

The process of extension is usually costly. It requires a deeper knowledge of the infrastructure, its capabilities and extension rules. As discussed in section 2.2, different fundamental and configuration-specific dependencies may exist. The lack of documentation and enforcement of these dependencies may result frequent programming errors and incompatibilities, increasing the development costs. Hence, as a general guideline, prefer flexible infrastructures as YANCEES that provide mechanisms to document and enforce dependencies between different features (Silva Filho and Redmiles 2007).

8.4 Conclusions

In this chapter, we presented a set of requirements analysis, design, implementation, selection and reuse principles and guidelines that can better support infrastructure developers and users. These principles allow infrastructure developers and users to better understand the concerns and challenges involved in the design and reuse of versatile infrastructures. With this basic knowledge, infrastructure developers can make better decisions when developing and evolving publish/subscribe infrastructures; whereas application developers can make better choices when reusing existing infrastructures in the development of their applications.

Chapter 9. Study Limitations

Like most empirical studies, the validity of our results is subject to several threats. In particular, the results discussed in this work are based on a selected set of metrics and a small number of infrastructures and case studies. There are a number of other existing metrics and other versatility dimensions that we could be exploited in our study. Nevertheless, there is no practical way in a single study to explore all the possible dimensions. For every possible measure there will be some dimensions that will remain uncovered, for example, in our study we did not consider software qualities such as testability, nor interoperability concerns. In the particular case of YANCEES, we did not consider the program comprehension costs involved in learning its extensibility and configurability APIs.

The evaluation of infrastructures based on metrics such as LOC and McCabe cyclomatic complexity, or more recent modularity metrics, used in our study are usually subject to criticism (Fenton and Neil 1999). As such, we considered not only the quantitative results they provide, but also our qualitative assessment of the infrastructure, discussing as much as possible our impressions and experiences in the use of the selected infrastructures.

The limited size and complexity of the examples used in the implementations may restrict the extrapolation of our results to other classes of middleware. In addition, our assessment is restricted to the specific publish/subscribe infrastructures and their implementation details. Moreover, the applications domains we selected in our studies may benefit certain types of infrastructures. For example, CORBA-NS has been widely adopted in support of real-time avionics applications, relying on different architectural and compilation optimizations (Harrison, Levine et al. 1997). The ORB we used in our studies, the community OpenORB does not provide any of these optimizations, which surely impacted its performance. JavaSpaces has been used in support of asynchronous collaborative applications and mobility (Murphy, Picco et al. 2006), two application domains not tested in our case studies.

Our benchmark (and its winners) is based purely on the objective test data and does not evaluate or consider other factors that may be of important to specific users' needs, for example interoperability, support for industry standards (that would benefit, for example, CORBA-NS), and specific application needs not tested in the benchmark.

An ideal assessment would require the analysis of different and independent applications, reusing the selected infrastructures. Since some of the infrastructures are research prototypes (Siena and YANCEES), this was not possible. Even industry standards such as CORBA-NS and JavaSpaces are not being used in large and diverse enough sets of open source applications, to support the complete (and automatic) assessment of their versatility approaches. As a result, we see our approach to build controlled implementations using these infrastructures, as the only feasible approach given the resources, man power, and time constraints we had.

Chapter 10. Related Work

In this chapter, we describe related work in the areas of middleware, software engineering, software product lines and design. Whereas there is an overwhelming volume of literature giving advice on software design, highlighting the benefits of novel software engineering approaches; there is equally an overwhelming need for research to validate the effectiveness of this advice (Kelly 2006). Our work contributes to these different areas by providing quantitative and qualitative data supporting the development and reuse of versatile software.

We further describe the related work in different research fields as follows.

10.1 Middleware versatility

In the middleware literature, existing research in flexible publish/subscribe infrastructures have reported the benefits of flexible approaches in support of application domain variability. Examples include GREEN (Sivaharan, Blair et al. 2005), FACET (Pratap, Hunleth et al. 2004) and YANCEES (Silva Filho and Redmiles 2005). Recent studies of the benefits of AOP in the modularity and maintainability of middleware have also been published (Hunleth and Cytron 2002; Zhang and Jacobsen 2004).

In the particular case of Database Management Systems (or DBMSs) research, the limitations of one-size-fits-all design, adopted by many infrastructures, have been discussed (Seltzer 2008). As a result, current research in flexible DBMs has emerged as one important research venue.

In spite of this variety of approaches, few studies discuss the factors that may hinder middleware adoption and success (Henning 2008). Moreover, most of these studies are restricted to one specific approach, and do not provide extensive comparative data comparing it with alternative approaches, nor perform a multi-dimensional analysis of trade-offs with a broad set of software qualities. Our work contributes with a broader and more comprehensive evaluation of different software versatility approaches, comparing their benefits and weaknesses with respect to a broad set of software qualities.

10.2 Software product line engineering

Software product line (SPL) engineering (Bockle, Pohl et al. 2005) is a relatively new approach to software development. It builds upon research in the areas of software reuse, evolution and flexibility, and strives to achieve the benefits of such approaches, while manages the trade-offs involved in the construction of similar software systems.

In SPL, while flexibility copes with software extensibility and configurability; reuse permits the reduction of the costs of producing similar software systems (over repetitive reuse cycles) (Bockle, Clements et al. 2004), decreasing time-to-market and defects faced in the development of software. SPL engineering goes one step ahead of previous software reuse approaches by systematizing reuse through the adoption of adequate design, implementation strategies and tools that automate the process of configuration management and derivation of software instances (Sinnema and Deelstra 2007). It takes advantage of scope, by focusing on a particular application domain, thus achieving a balance between generality and specificity. It also leverages on domain

knowledge gained through the repetitive development of software, supporting the reduction of the cognitive distance and the improvement of infrastructure usability.

Our work contributes to software product line engineering with quantitative and qualitative data that studies the effect of different design decisions in the versatility of software. It quantitatively and qualitatively analyzes the benefits and costs of flexible software (YANCEES), thus providing insights to the development of software product lines.

It also contributes with YANCEES, an infrastructure and architectural style that provides insights on the role of dependencies in SPL design and implementation, providing common solutions to the problems originated by these dependencies.

10.2.1 Analysis of dependencies in software product lines

The study of the role of dependencies in software product lines has gained recent attention from the research community. The focus, however, has been more on the use of those dependencies to prevent architecture configuration mismatch, and less on the study of the impact of those dependencies on the system design complexity and their impact on the variability of software. For example, in the FODA (Kang, Cohen et al. 1990), FORM (Kang 1998), RSEB (Griss, Favaro et al. 1998) methods and in the generative approach in (Czarnecki and Eisenecker 2000), dependencies are used to model usage interactions (alternative, multiple, optional and mandatory) as well as incompatibility relations (exclusive or excludes), with the focus on configuration management and conflict resolution. Recently, (Ferber, Haag et al. 2002) stresses the importance of dependency analysis in feature diagrams, and proposes a separate feature-dependency model that complements the existing feature tress. Additionally, it characterizes different interactions between features such as intentional, environmental, and usage dependencies. Finally, in a more recent work, (Lee and Kang 2004) studied the role of dependencies on modeling runtime feature interactions, introducing the notion of activation and modification dependencies in feature diagrams.

In the implementation domain, feature dependencies usually manifest themselves in the form of coupling between the components that implement those features, in special data and control coupling occur as a consequence of activation and usage dependencies. Those dependencies have different impacts in the variability of the final solution. Whereas control coupling usually limits the activation order of the different pieces of software, data coupling can limit the variability and reuse of those components. (Parnas 1978; Stevens, Myers et al. 1999)

On the light of those problems, different variability realization approaches have been used. For example, (Lee and Kang 2004) propose a set of object-oriented realization strategies to address activation dependencies. Those strategies are presented in the form of design patterns derived from existing Factory, Proxy and Builder patterns (Gamma, Helm et al. 1995). In essence, those patterns focus on managing and enforcing activation dependencies by promoting the late-binding of the components that implement the many software features. Whereas useful in many contexts, this modular (object-oriented) decomposition is not always sufficient to address other kinds of dependencies, especially crosscutting variability dimensions or aspects, originated from more fundamental problem dependencies. This motivated recent work such as (Garcia, Sant'Anna et al. 2005), where Aspects are used to modularize design patterns.

In our work, we argue towards a more deep understanding of the role in dependencies in software product lines. Not only as important information for configuration management support, but as main factors to be considering in the design, bounding and variability realization selection phases, as discussed in section 2.2.

10.2.2 Software product lines economic models

From an economic perspective, SPL yields an economy of scope through the reuse of existing assets. Reuse is planned, enabled and enforced. Hence, the gains in productivity through the use of SPL engineering are directly proportional to the variability in the domain and the number of different software instances that one needs to support in a program family (Bockle, Clements et al. 2004).

Existing software product line economic models allows the comparison between software product line approaches as YANCEES with more traditional approaches. For example, the work of (Bockle, Clements et al. 2004) and (Frakes and Terry 1996). These high-level economic models, however, do not provide insights on the difficulties and challenges involved in the development and reuse of SPLs. For example, they do not account for domain-specific factors that can hinder reusability, such as adaptation costs, performance and subscription and notification models mismatches.

Our work contributes to software product line economic models by discussing in detail the main challenges and factors that may hinder the development of versatile software (discussed in Chapter 2), while provides insights that can guide the engineering of such systems, thus achieving a favorable balance between the main versatility software qualities (as discussed in Chapter 3and Chapter 8).

10.3 Software design and analysis methodologies

Existing methods such as the Software Engineering Institute's ATAM (Architecture Trade-off Analysis Method) (Kazman, Klein et al. 2000), SAAM (Software Architecture Analysis Method) (Paul Clements 2001), and ADD (Attribute-Driven Design) (Wood 2007) can be used to assess the consequences of architectural decisions at the light of quality attribute requirements. These methodologies, however, are design-time methodologies, being limited to the estimation, instead of the analysis of actual trade-offs defined by existing design and implementation decisions of actual components. Our work fills these gaps by analyzing existing infrastructures design and implementations in realistic scenarios.

10.4 Empirical software engineering

From an empirical software engineering perspective, multi-dimensional analysis of different versatility approaches are rare. Nevertheless, some quantitative research has been done on the analysis and validation of design principles and their impact in different software qualities.

For example, the study of the impact of middleware stability on supporting changes in non-functional requirements (Bahsoon, Emmerich et al. 2005); on the benefits of *reuse* on large software projects (Mohagheghi and Conradi 2008) and software product lines (van Ommering 2005); on the role of software architecture *stability* in software evolution (Jazayeri 2002); on the issues in the modularization of software using aspects (Garcia, Sant'Anna et al. 2005; Lopes and Bajracharya 2006; Greenwood, Bartolomei et al. 2007); on the *usability* of design patterns such as Factory (Ellis, Stylos et al. 2007); on the flexibility (and inflexibility) of design patterns (Mens and Eden 2005); and the impact of different middleware on performance (Tselikis, Mitropoulos et al. 2007).

These quantitative studies, however, focus on a punctual software qualities such as *reusability*, *stability*, *modularity, usability, flexibility* and *performance* respectively. To the best of our knowledge, no in-depth analysis exist that evaluates and compares the different trade-offs between important software qualities such as: *infrastructures complexity, reusability*, *maintainabil-*

ity, *flexibility,* and *performance.* Moreover, most studies are performed from the points-of-view infrastructure developers, with little or no focus on API *usability* analysis.

10.5 Design principles literature

Different general design principles have been advocated by Parnas (Parnas 1978), Raymond (Raymond 2004), Gabriel (Gabriel 1991) and Kiczales (Kiczales, Lamping et al. 1997); different API Guidelines have been proposed by (McConnell 2004), (McLellan, Roesler et al. 1998), (Bloch 2006) and (Henning 2009); whereas lower-level implementation guidelines are popular in the object-oriented literature (Gamma, Helm et al. 1995; Martin 2003). Very few studies, however, have investigated and quantitatively analyzed the effects of the application of these principles on realistic systems, thus providing hard evidence of their benefits and possible costs. Our work confirms some of these principles and provides hard evidence of the impact of their application on important software qualities.

Chapter 11. Conclusions

The development and reuse of versatile software infrastructures faces different and not so well understood trade-offs, a consequence of different domain-specific requirements, and architectural and implementation decisions. By analyzing and elucidating these trade-offs in the publish/subscribe domain, and deriving principles and guidelines, we seek to better support publish/subscribe infrastructure developers in the design of better middleware, and infrastructure consumers in selecting among the existing options available.

In particular, in this work, we presented a quantitative and qualitative study that analyzed the costs and benefits of existing publish/subscribe infrastructures representing different versatility approaches. The results are summarized in the form of data presented throughout this document, in the design and implementation of YANCEES (Chapter 3), and a set of design principles and guidelines presented in Chapter 8.

11.1 Summary of contributions

The contributions of this work crosscut different research areas as follows:

11.1.1 Contributions in software engineering in general

This works contributes to software engineering research in the following manner:

- We propose the concept of versatility, together with an analytical framework that describes major operators employed in the development and reuse of versatile software (discussed in section Chapter 2), describing their main benefits and costs;
- We also perform a non-exhaustive survey of major architectural approaches adopted in the development of versatile software in general, and pub/sub infrastructures specifically (discussed at (Silva Filho and Redmiles 2005) and in section 4.1), evaluating infrastructures developed according to these approaches in our case studies;
- In order to analyze different versatility approaches, we designed comprehensive evaluation framework to compare the versatility of heterogeneous software infrastructures (Chapter 4).
- In doing so, we designed and applied a metrics suite, which quantifies software qualities as: usability, reusability, performance, flexibility, and maintainability in terms of lower-level attributes (section 4.4). In particular, we propose a new metric called development effort, which is the product of the total lines of code and cyclomatic complexity. This metric is the basis for our measurement of usability and reusability.
- The collected data was analyzed for correlations between these different software qualities, thus identifying trade-offs (Chapter 7). In particular, we provide empirical data showing that flexibility is more a consequence of design for change rather than the mere application of good software practices.
- Based on our case study, we also contribute with a set of principles and guidelines for requirements analysis, development and reuse of versatile publish/subscribe infrastructures (Chapter 8).

- Finally, we show the impossibility of the construction of an ideally versatile publish/subscribe infrastructure, one that can have its characteristics evolved independently from each other, pointing out the role of dependencies in limiting variability (as discussed in Chapter 2).

11.1.2 Contributions in the software product line engineering

In the software product line research, we contribute with:

- A deeper understanding of the impact of dependencies in limiting software flexibility, and an analysis of different feature interference problems in YANCEES (as discussed at (Silva Filho and Redmiles 2007) and in Chapter 2 of this document)
- An analysis of the role of dependencies in limiting variability, and a notation to express dependencies (as discussed at (Silva Filho and Redmiles 2006) and in Chapter 2 of this document).
- A comparative study of the versatility trade-offs in publish/subscribe infrastructures which compares flexible software product line approach (as YANCEES) with more traditional alternatives as: coordination languages (JavaSpaces), one-size-fits-all (CORBA-NS), and minimal core (Siena), as discussed in Chapter 7.

11.1.3 Contributions to middleware research

With respect to middleware research, we contribute with:

- YANCEES, a flexible pub/sub infrastructure (Silva Filho, de Souza et al. 2003; Silva Filho and Redmiles 2005), and a set of design principles supporting its development, showing how to achieve a favorable balance between different software qualities in this domain;
- The extended Rosenblum and Wolf (Rosenblum and Wolf 1997) design model for publish/subscribe infrastructures, showing the importance to support protocols (discussed at (Silva Filho, de Souza et al. 2003) and in section Chapter 3 of this document);
- We also contribute with a quantitative and qualitative study of publish/subscribe middleware, where we show the complexity of using, extending and reusing different infrastructures.

11.2 Future work

The work discussed in this dissertation, including the analysis of versatility trade-offs, the principles we derived, and the design and implementation of YANCEES, only represents the beginnings of new and promising approach to the creation of versatile software in general and publish/subscribe infrastructures specifically. The development of versatile software still faces many challenges that can benefit from further research in areas such as: program comprehension, API usability and programming language design.

11.2.1 Tool support for software comprehension and evolution

Even though different design and implementation approaches exist to support the development of versatile software (for example, software product line engineering (Kang, Lee et al. 2003; Lee and Kang 2004), component models, plug-ins, frameworks, and many others (Svahnberg,

Gurp et al. 2005)), the process of design, evolution and reuse of software developed according to these approaches is still not fully supported.

For example, a common problem faced by users of versatile software is the understanding of the original intention, assumptions and rationale of software (Bosch, Florijn et al. 2002). Whereas some approaches have already been proposed to document and automate the process of understanding and reusing software product line assets (Sinnema, Deelstra et al. 2004; Sinnema, Deelstra et al. 2006), much work still needs to be done. In particular, there is a lack of good approaches to capture, represent and enforce versatility context. By versatility context I mean: *the information necessary to correctly understand, extend, adapt, configure and ultimately reuse a piece of software to a particular need*. Context requires the timely gathering of otherwise hidden, scattered information, their enforcement and presentation in meaningful ways to software developers thus supporting their activity at hand. Moreover, many times, this information is tacit, not written in any documentation form, but is part of the expertise of few developers, which makes it event difficult to locate, combine and present this information.

As a future research in this area, I plan on answering the following questions: What kind of context information do developers need? How can this information be gathered, presented and enforced? Can we derive usable and useful ways to capture, document, present and enforce this information? How can we support tacit knowledge capturing, representing and sharing?

An initial attempt to answer some of these questions is in our early work on the documentation and enforcement of dependencies in YANCEES (Silva Filho and Redmiles 2007). Thus, we plan to generalize this work to include other types of systems, and extensible APIs in general.

11.2.2 API usability metrics, guidelines and tool support

Application Programming Interfaces (or APIs) define reusable abstractions applied in the construction of complex software systems. They not only support the management of software complexity, but also work as boundaries between relatively independent software development teams (De Souza 2005). In spite of their importance, very little research has been done on the design and evaluation of APIs (Henning 2009). In this dissertation work, the size of the APIs and the number of concerns API users need to master have shown to be important factors in the total effort of adapting, extending and configuring an infrastructure.

As a future work, I plan on broadening the scope of the initial API usability research done in this work, by answering the following questions: What is a good API? How can we adequately measure API usability? What's the impact of sound software engineering approaches in the resulting API usability? Can we develop better principles, guidelines and tools to better support the development of APIs?

The answering of these questions will not only benefit the development of versatile software, but all sorts of software systems in general. Moreover, by better understanding the process of design and development of APIs, we can better support the development of automated tools to guide developers in their development process.

11.2.3 Study of the impact of programming paradigm in software versatility

Different programming paradigms are able to impact important software qualities such as flexibility, usability and reusability. In fact, many of the problems found in our versatility analysis were influenced by the programming paradigm adopted (Object-Oriented), its dominant de-

composition (Objects) and integration mechanism (method invocation). For example, some incidental dependencies, and the scattering of functionality throughout different components of YANCEES are a consequence of the Object-Oriented decomposition model adopted, that many times cannot modularize concerns into single classes.

Approaches such as Aspect-Oriented Programming, for example, can improve the modularity of software (Lopes and Bajracharya 2006), supporting better locality of change. This approach, however, has also implicit cots, some of them related to the scalability and usability of program comprehension (Ruengmee, Silva Filho et al. 2008). These sets of benefits as well as costs must be better understood in order to inform tools and techniques to better support developers in applying these paradigms.

In particular, we are interested in better understanding the role of programming paradigm in software versatility. For such, I plan to further analyze and compare the impact of different programming paradigms, such as Object-Oriented, Aspect-Oriented and Implicit Invocation languages in the resulting versatility of existing Middleware. While some initial evidence was collected by (Leung 2006), on implicit invocaiton languages, and (Lopes and Bajracharya 2006) on AOP, a comparative approach between these paradigms are still missing.

In particular, I plan to investigate the following research questions: What benefits different programming paradigms afford to software versatility? At what costs? Can we derive principles and tools to help developers leverage on these paradigms benefits, while managing their incidental issues? By answering these questions, our goal is to better support versatile software development and reuse, informing the design of novel tools for versatile software engineering.

References

Bahsoon, R., W. Emmerich, et al. (2005). "Using real options to select stable middleware-induced software architectures." IEE Proceedings - Software Engineering **152**(4): :167 - 186.

Baldoni, R., M. Contenti, et al. (2003). The Evolution of Publish/Subscribe Communication Systems. Future Directions of Distributed Computing. Springer-Verlag. **2584**.

Baldwin, C. Y. and K. B. Clark (2000). Design Rules, Vol. 1: The Power of Modularity. Cambridge, MA, MIT Press.

Bandi, R. K., V. K. Vaishnavi, et al. (2003). "Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics." IEEE Transactions on Software Engineering **29**(1).

Barns, B. H. and T. B. Bollinger (1991). "Making reuse cost-effective." IEEE Software **8**(1): 13-24.

Batory, D., J. N. Sarvela, et al. (2003). Scaling step-wise refinement. Proceedings of the 25th International Conference on Software Engineering. Portland, Oregon, IEEE Computer Society**:** 187 - 197.

Bergmans, L. and M. Aksit (2001). "Composing Crosscutting Concerns Using Composition Filters." Communications of the ACM **44**(10): 51-58.

Birsan, D. (2005). On Plug-ins and Extensible Architectures. ACM Queue. **3:** 40-46.

Biscotti, F., T. Jones, et al. (2008). Market Share: Application Infrastructure and Middleware Software, Worldwide, 2007. Stamford, CT, Gartner Group**:** 31.

Bloch, J. J. (2006). How to design a good API and why it matters. OOPSLA Companion.

Bockle, G., P. Clements, et al. (2004). "Calculating ROI for software product lines." IEEE Software **21**(3): 23- 31.

Bockle, G., K. Pohl, et al. (2005). Software Product Line Engineering. Heidelberg, Berlin, New York, Springer.

Bosch, J. (2004). Software Architecture: The Next Step. Lecture Notes in Computer Science. Berlin / Heidelberg, Springer **3047:** 194-199.

Bosch, J., G. Florijn, et al. (2002). "Variability Issues in Software Product Lines." Lecture Notes in Computer Science - 4th International Workshop on Software Product Family Engineering - PFE'2002 **2290/2002**: 303-338.

Bowen, T. F., F. S. Dworack, et al. (1989). The feature interaction problem in telecommunications systems. Software Engineering for Telecommunication Switching Systems.

Boyer, R. T. and W. G. Griswold (2004). Fulcrum – An Open-Implementation Approach to Context-Aware Publish/Subscribe. San Diego, UCSD.

Brooks, F. P. (1987). No Silver Bullet: Essence and Accident in Software Engineering. IEEE Computer 20. **10:** 10-19.

Cabrera, L. F., M. B. Jones, et al. (2001). Herald: Achieving a Global Event Notification Service. Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau, Germany, IEEE Computer Society.

Cacho, N., C. Sant'Anna, et al. (2006). Composing design patterns: a scalability study of aspect-oriented programming. 5th international conference on Aspect-oriented software development, Bonn, Germany.

Cardone, R., A. Brown, et al. (2002). Using Mixins to Build Flexible Widgets. 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands.

Carzaniga, A., D. S. Rosenblum, et al. (2000). Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. Nineteenth ACM Symposium on Principles of Distributed Computing, Portland, OR, ACM Press.

Carzaniga, A., D. S. Rosenblum, et al. (2001). "Design and Evaluation of a Wide-Area Event Notification Service." ACM Transactions on Computer Systems **19**(3): 332-383.

Castro, M., P. Druschel, et al. (2002). "SCRIBE: A Large-Scale and Decentralized Application-Level Multicast Infrastructure." IEEE Journal on Selected Areas in Communications **20**(8): 1489-1499.

Clarke, S. (2004). Measuring API Usability. Dr. Dobb's Journal Windows/.NET Supplement**:** S6-S9.

Clements, P. and L. Northrop (2002). Software Product Lines: Practices and Patterns, Addison-Wesley.

Coplien, J., D. Hoffman, et al. (1998). Commonality and Variability in Software Engineering. IEEE Software. **15:** 37-45.

Cugola, G., E. D. Nitto, et al. (2001). "The Jedi Event-Based Infrastructure and Its Application on the Development of the OPSS WFMS." IEEE Transactions on Software Engineering **27**(9): 827-849.

Czarnecki, K. and U. W. Eisenecker (2000). Generative Programming - Methods, Tools, and Applications, Addison-Wesley.

Czarnecki, K., S. Helsen, et al. (2005). "Formalizing Cardinality-based Feature Models and their Specialization." Software Process Improvement and Practice, special issue of best papers from SPLC04, **10**(1): 7 - 29.

De Souza, C. R. B. (2005). On the Relationship between Software Dependencies and Coordination: Field Studies and Tool Support. Ph.D. dissertation, Donald Bren School of Information and Computer Sciences, University of California, Irvine,. Irvine, CA, USA.

DePaula, R., X. Ding, et al. (2005). "In the Eye of the Beholder: A Visualization-based Approach to Information System Security." International Journal of Human-Computer Studies - Special Issue on HCI Research in Privacy and Security **63**(1-2): 5-24.

DePaula, R., X. Ding, et al. (2005). Two Experiences Designing for Effective Security. Symposium on Usable Privacy and Security, Pittsburgh, PA.

Eaddy, M., T. Zimmermann, et al. (2008). "Do Crosscutting Concerns Cause Defects?" Software Engineering, IEEE Transactions on **34**(4): 497-515.

Eden, A. H. and T. Mens (2006). "Measuring Software Flexibility." IEEE Software **153**(3): 113-126.

Ellis, B., J. Stylos, et al. (2007). The Factory Pattern in API Design: A Usability Evaluation. 29th International Conference on Software Engineering ICSE '07, Minneapolis, MI, IEEE Computer Society.

Emmerich, W. (2000). Software Engineering and Middleware: A Roadmap. The Future of Software Engineering. A. Finkelstein, ACM Press.

Factor, M. (1990). The process trellis architecture for real-time monitors. 2nd ACM SIGPLAN symposium on Principles & practice of parallel programming, Seattle, Washington, United States.

Fenton, N. E. and M. Neil (1999). "Software Metrics: Successes, Failures and New Directions." Journal of Systems and Software **47**(2-3): 149-157.

Ferber, S., J. Haag, et al. (2002). "Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line." Lecture Notes in Computer Science. Second International Conference on Software Product Lines, SPLC'02 **2379**: 235-256.

Fitzpatrick, G., T. Mansfield, et al. (1999). Instrumenting and Augmenting the Workaday World with a Generic Notification Service called Elvin. European Conference on Computer Supported Cooperative Work (ECSCW '99), Copenhagen, Denmark, Kluwer.

Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection Pattern, http://www.martinfowler.com/articles/injection.html.

Frakes, W. and C. Terry (1996). "Software reuse: metrics and models." ACM Computing Surveys **28**(2): 415-435.

Freeman, E., S. Hupfer, et al. (1999). JavaSpaces Principles, Patterns, and Practice, Book News, Inc.

Gabriel, R. P. (1991). Lisp: Good News, Bad News, How to Win Big (Keynote). EuroPAL (European Conference on the Practical Applications of Lisp) Cambridge, UK, http://www.dreamsongs.com/WIB.html.

Gamma, E., R. Helm, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company.

Garcia, A., C. Sant'Anna, et al. (2005). Modularizing design patterns with aspects: a quantitative study. Aspect-oriented software development, Chicago, Illinois, ACM Press.

Garlan, D., R. Allen, et al. (1995). "Architectural Mismatch: Why Reuse Is So Hard." IEEE Software **12**(6): 17-26.

Gelernter, D. (1985). "Generative communication in Linda." ACM Transactions on Programming Languages and Systems (TOPLAS **7**(1).

Geyer, W., R. S. Silva Filho, et al. (2008). "The Trade-Offs of Blending Synchronous and Asynchronous Communication Services to Support Contextual Collaboration." Journal of Universal Computer Science (special issue on Groupware) **14**(1): 4-26.

Glass, R. L. (1994). "The Software-Research Crisis." IEEE Software **11**(6): 42-47.

Gore, P., I. Pyarali, et al. (2004). The Design and Performance of a Real-Time Notification Service. 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04).

Greenwood, P., T. Bartolomei, et al. (2007). On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. LNCS - Proceedings of the ECOOP 2007 – Object-Oriented Programming. Berlin Heidelberg, Springer-Verlag. **4609/2007:** 176-200.

Griss, M. L., J. Favaro, et al. (1998). Integrating Feature Modeling with RSEB. Fifth International Conference on Software Reuse.

Group, O. M. (2003). Deployment and Configuration of Component-based Distributed Applications Specification. Adopted Submission. OMG Document ptc/03-07-08, OMG.

Gruber, R. E., B. Krishnamurthy, et al. (1999). The Architecture of the READY Event Notification Service. ICDCS Workshop on Electronic Commerce and Web-Based Applications, Austin, TX, USA.

Harrison, T. H., D. L. Levine, et al. (1997). The Design and Performance of a Real-time CORBA Object Event Service. OOPSLA'97, Atlanta, GA, ACM.

Heimbigner, D. (2003). Extending the Siena Publish/Subscribe System. Technical Report: CU-CS-946-03. Boulder, Colorado, CU, Bolder.

Henning, M. (2008). "The rise and fall of CORBA." Commun. ACM **51**(8): 52-57.

Henning, M. (2009). "API design matters." Commununications of ACM **52**(5): 46-56.

Hilbert, D. and D. Redmiles (1998). An Approach to Large-scale Collection of Application Usage Data over the Internet. 20th International Conference on Software Engineering (ICSE '98), Kyoto, Japan, IEEE Computer Society Press.

Hunleth, F. and R. K. Cytron (2002). Footprint and feature management using aspect-oriented programming techniques. Joint conference on Languages, Compilers and Tools for Embedded Systems, Berlin, Germany, ACM Press.

IEEE (1993). IEEE standard for a software quality metrics methodology (IEEE Std 1061-1992).

Jamil, T. (1995). "RISC versus CISC." Potentials, IEEE **14**(3): 13-16.

Jazayeri, M. (2002). "On Architectural Stability and Evolution." Lecture Notes In Computer Science. Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies **2361**: 13 - 23.

Jingyue, L., R. Conradi, et al. (2009). "Development with Off-the-Shelf Components: 10 Facts." Software, IEEE **26**(2): 80-87.

Kang, K., K. Lee, et al. (2003). Feature Oriented Product Line Software Engineering: Principles and Guidelnes. Domain Oriented Systems Development: Practices and Perspectives. UK. **1:** 29-46.

Kang, K. C. (1998). "FORM: A Feature-Oriented Reuse Method with Domain Specific Architectures." Annals of Software Engineering **5**: 345-355.

Kang, K. C., S. G. Cohen, et al. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study - CMU/SEI-90-TR-021. Pittsburgh, PA, Carnegie Mellon Software Engineering Institute.

Kantor, M. and D. Redmiles (2001). Creating an Infrastructure for Ubiquitous Awareness. Eighth IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001), Tokyo, Japan.

Kazman, R., M. Klein, et al. (2000). ATAM: Method for Architecture Evaluation CMU/SEI-2000-TR-004. Pittsburgh, PA, CMU**:** 83.

Kelly, D. (2006). "A Study of Design Characteristics in Evolving Software Using Stability as a Criterion." IEEE Transactions on Software Engineering **32**(5): 315-329.

Kiczales, G. (1995). Towards a New Model of Abstraction in the Engineering of Software (Why Are Black Boxes So Hard To Reuse?). Invited Talk, 17th International Conference on Software Engineering, Seattle, WA.

Kiczales, G., J. Lamping, et al. (1997). Open Implementation Design Guidelines. International Conference of Software Engineering (ICSE'97), Boston, MA, ACM Press.

Kiczales, G., J. Lamping, et al. (1997). Aspect-Oriented Programming. European Conference on Object-Oriented Programming, Jyväskylä, Finland, Springer-Verlag.

Kim, J. and D. H. Bae (2006). "An approach to feature-based software construction for enhancing maintainability." Software Practice and Experience **36**(9): 923-948.

Krueger, C. (2006). Software Product Line Concepts: www.softwareproductlines.com/introduction/concepts.html, The Software Product Lines site.

Krueger, C. W. (1992). "Software Reuse." ACM Computing Surveys **24**(3): 131-184.

Larman, C. and V. R. Basili (2003). Iterative and Incremental Development: A Brief History. IEEE Computer. **36:** 47-56.

Leclercq, M., V. Quema, et al. (2005). "DREAM: a Component Framework for the Construction of Resource-Aware, Reconfigurable MOMs." IEEE Distributed Systems Online **6**(9): 1-12.

Lee, K. and K. C. Kang (2004). "Feature Dependency Analysis for Product Line Component Design." Lecture Notes in Computer Science - 8th International Conference on Software Reuse, ICSR'04 **3107**: 69-85.

Lehman, M. M. and F. N. Parr (1976). Program evolution and its impact on software engineering. 2nd international conference on software engineering, San Francisco, CA, USA, IEEE Computer Society Press.

Lehman, M. M., J. F. Ramil, et al. (1997). Metrics and laws of software evolution-the nineties view. 4th International Software Metrics Symposium, Albuquerque, NM, USA, IEEE.

Leung, W.-H. F. (2006). "Program entanglement, feature interaction and the feature language extensions." Computer Networks **51**(2): 480-495.

Li, W. and S. Henry (1993). "Object-oriented metrics that predict maintainability." Systems and Software **23**(2): 111-122.

Lidwell, W., K. Holden, et al. (2003). Universal Principles of Design. Beverly, MA, Rockport.

Lieberherr, K. J. and I. M. Holland (1989). Assuring good style for object-oriented programs. IEEE Software. **6:** 38-48.

Liskov, B. (1987). Keynote address - Data Abstraction and Hierarchy. Object Oriented Programming Systems Languages and Applications (OOPSLA'87), Orlando, Florida.

Liskov, B. and S. Zilles (1974). Programming with abstract data types. Proceedings of the ACM SIGPLAN symposium on Very high level languages. Santa Monica, California, United States, ACM.

Long, J. (2001). "Software reuse antipatterns." ACM SIGSOFT Software Engineering Notes **26**(4): 68-76.

Lopes, C. and S. Bajracharya (2006). "Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value." Transactions on Aspect-Oriented Software Development I (TASOD) - Lecture Notes in Computer Science **3880**(1): 1-35.

Lopes, C. V. and S. Bajracharya (2006). "An Analysis of Modularity in Aspect-Oriented Design." Springer LNCS Transactions on Aspect-Oriented Software Development **1**(3880): 1-35.

M. Eaddy, A. A. and G. C. Murphy (2007). Identifying, Assigning, and Quantifying Crosscutting Concerns. Workshop on Assessment of Contemporary Modularization Techniques (ACoM), Minneapolis, Minnesota, USA.

Mahdy, A. and M. E. Fayad (2002). A Software Stability Model Pattern. 9th Conference on Pattern Language of Programs - PLOP2002, Monticello Illinois.

Martin, R. C. (2003). Agile Software Development, Principles, Patterns, and Practices. Englewood Cliffs, NJ, Prentice Hall.

McCabe, T. J. (1976). A Complexity Measure. IEEE Transactions on Software Engineering. **SE-2:** 308-320.

McConnell, S. (2004). Code Complete, Second Edition, Microsoft Press.

McLellan, S. G., A. W. Roesler, et al. (1998). "Building more usable APIs." IEEE Software **15**(3): 78-86.

Mens, T. and A. H. Eden (2005). "On the Evolution Complexity of Design Patterns." Electronic Notes in Theoretical Computer Science **127**(3): 147-163.

Meyer, B. (1992). "Applying `design by contract'." Computer **25**(10): 40-51.

Meyer, B. (1997). Object-Oriented Software Construction, 2nd Edition. Upper Saddle River, NJ, Prentice Hall.

Mohagheghi, P. and R. Conradi (2008). "An empirical investigation of software reuse benefits in a large telecom product." ACM Trans. Softw. Eng. Methodol. **17**(3): 1-31.

Murphy, A. L., G. P. Picco, et al. (2006). "LIME: A Coordination Middleware Supporting Mobility of Hosts and Agents." ACM Transactions on Software Engineering and Methodology **15**(3): 279-328.

Naslavsky, L., R. S. Silva Filho, et al. (2004). Distributed Expectation-Driven Residual Testing. Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS'04). Edinburgh, UK.

OMG (2001). CORBA Event Service Specification (version 1.1), Object Management Group.

OMG (2004). CORBAcos Notification Service version 1.1 formal/04-10-13, Object Management Group**:** 229.

Parnas, D. L. (1972). On the Criteria to Be Used in Decomposing Systems into Modules. Communications of the ACM. **15:** 1053-1058.

Parnas, D. L. (1978). Designing software for ease of extension and contraction. 3rd international conference on Software engineering, Atlanta, Georgia, USA, IEEE Press.

Parnas, D. L. (1994). Software Aging. 16th international conference on Software engineering, Sorrento, Italy.

Parnas, D. L., P. C. Clements, et al. (1984). The modular structure of complex systems. International Conference on Software Engineering, Orlando, Florida, United States, IEEE Press Piscataway, NJ, USA.

Parnas, D. L., J. E. Shore, et al. (1976). "Abstract types defined as classes of variables." SIGMOD Rec. **8**(2): 149-154.

Paul Clements, R. K., Mark Klein (2001). Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley.

Pratap, R. M., F. Hunleth, et al. (2004). "Building fully customizable middleware using an aspect-oriented approach." IEE Proceedings - Software Engineering **151**(4): 199-216.

Raymond, E. S. (2004). The Art of UNIX Programming, Addison-Wesley.

Redmiles, D., A. van der Hoek, et al. (2007). "Continuous Coordination: A New Paradigm to Support Globally Distributed Software Development Projects." Wirtschaftsinformatik (Special Issue on the Industrialization of Software Development) **49**(3).

Robillard, M. P. and G. C. Murphy (2007). "Representing concerns in source code." Transactions on Software Engineering and Methodology (TOSEM) **16**(1).

Robillard, M. P. and F. Weigand-Warr (2005). ConcernMapper: Simple ViewBased Separation of Scattered Concerns. Eclipse Technology Exchange at OOPSLA, San Diego, CA, ACM Press.

Roeller, R., P. Lago, et al. (2006). "Recovering architectural assumptions." Journal of Systems and Software **79**(4): 552-573.

Rosenblum, D. S. and A. L. Wolf (1997). A Design Framework for Internet-Scale Event Observation and Notification. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, Springer-Verlag.

Ruengmee, W., R. S. Silva Filho, et al. (2008). XE (eXtreme Editor) - Bridging the Aspect-Oriented Programming Usability Gap. Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on.

Rus, D., R. Gray, et al. (1997). "Transportable Information Agents." Journal of Intelligent Information Systems **9**(3): 215-238.

Sangwan, R. S., L. Li-Ping, et al. (2008). "Structural Complexity in Architecture-Centric Software Evolution." Computer **41**(10): 96-99.

Schwartz, B. (2004). The Paradox of Choice: Why More Is Less. New York, NY, Harper Collings.

Seltzer, M. (2008). "Beyond Relational Databases." Communications of the ACM **51**(7): 52-58.

Siegel, J. (1998). OMG overview: CORBA and the OMA in enterprise computing. Communications of the ACM. **41:** 37-43.

Silva Filho, R. S., C. R. B. de Souza, et al. (2003). The Design of a Configurable, Extensible and Dynamic Notification Service. International Workshop on Distributed Event Systems (DEBS'03), San Diego, CA.

Silva Filho, R. S., W. Geyer, et al. (2005). Architectural Trade-Offs for Collaboration Services Supporting Contextual Collaboration - RC23756. Cambridge, MA, IBM T. J. Watson.

Silva Filho, R. S. and D. Redmiles (2005). Striving for Versatility in Publish/Subscribe Infrastructures. 5th International Workshop on Software Engineering and Middleware (SEM'2005), co-located with ESEC/FSE'05 Conference, Lisbon, Portugal, ACM Press.

Silva Filho, R. S. and D. Redmiles (2005). Striving for Versatility in Publish/Subscribe Infrastructures. 5th International Workshop on Software Engineering and Middleware (SEM'2005), Lisbon, Portugal., ACM Press.

Silva Filho, R. S. and D. F. Redmiles (2005). A Survey on Versatility for Publish/Subscribe Infrastructures. Technical Report UCI-ISR-05-8. Irvine, CA, Institute for Software Research**:** 1-77.

Silva Filho, R. S. and D. F. Redmiles (2006). Extending Desktop Applications with Pocket-size Devices. Symposium on Usable Privacy and Security (SOUPS'06), Pittsburgh, PA.

Silva Filho, R. S. and D. F. Redmiles (2006). Towards the use of Dependencies to Manage Variability in Software Product Lines. Workshop on Managing Variability for Software Product Lines. (SPLC'2006), Baltimore, MD.

Silva Filho, R. S. and D. F. Redmiles (2007). Managing Feature Interaction by Documenting and Enforcing Dependencies in Software Product Lines. 9th International Conference on Feature Interaction, Grenoble, France.

Simon, H. A. (1996). The Sciences of the Artificial (3rd edition). Cambridge, MA, MIT Press.

Sinnema, M. and S. Deelstra (2007). "Classifying variability modeling techniques." Information and Software Technology **49**(7): 717-739.

Sinnema, M., S. Deelstra, et al. (2006). The COVAMOF Derivation Process. 9th International Conference on Software Reuse (ICSR 2006), Torino, Italy.

Sinnema, M., S. Deelstra, et al. (2004). "COVAMOF: A Framework for Modeling Variability in Software Product Families." Lecture Notes in Computer Science **3154/2004**: 197-213.

Sivaharan, T., G. S. Blair, et al. (2005). GREEN: A Configurable and Re-Configurable Publish-Subscribe Middleware for Pervasive Computing. Distributed Objects and Applications (DOA'05), Agia Napa, Cyprus.

Stevens, W. P., G. J. Myers, et al. (1999). "Structured Design." IBM Systems Journal **38**(2-3): 231 - 256.

Sullivan, K. J., W. G. Griswold, et al. (2001). The structure and value of modularity in software design. 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, Viena, Austria.

Sun Microsystems (2003). Java Message Service API, Sun Microsystems. **2003**.

Svahnberg, M., J. v. Gurp, et al. (2005). "A Taxonomy of Variability Realization Techniques." Software Practice and Experience **35**(8): 705-754.

Szyperski, C. (2002). Component Software: Beyond Object-Oriented Programming, 2nd edition, ACM Press.

Tarr, P., H. Ossher, et al. (1999). N degrees of separation: multi-dimensional separation of concerns. International Conference on Software Engineering, Los Angeles, CA, ACM.

Tselikis, C., S. Mitropoulos, et al. (2007). "An evaluation of the middleware's impact on the performance of object oriented distributed systems." Systems and Software **80**(7): 1169-1181.

van Gurp, J., J. Bosch, et al. (2001). On the notion of variability in software product lines. Working IEEE/IFIP Conference on Software Architecture - WICSA'2001, Amsterdam, IEEE.

van Ommering, R. (2005). "Software reuse in product populations." Software Engineering, IEEE Transactions on **31**(7): 537-550.

Wood, W. G. (2007). A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0 CMU/SEI-2007-TR-005. Pittsburgh, PA, CMU.

Woodfield, S. N. (1979). "An Experiment on Unit Increase in Problem Complexity." IEEE Transactions on Software Engineering **SE-5**(2): 76-79.

Wulf, W., E. Cohen, et al. (1974). "HYDRA: the kernel of a multiprocessor operating system." Commun. ACM **17**(6): 337-345.

Wyckoff, P. (1998). "TSpaces." IBM Systems Journal **37**(3).

Zavattaro, G. and N. Busi (2001). Publish/subscribe vs. Shared Dataspace Coordination Infrastructures. 10th IEEE Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises, Boston, MA.

Zhang, C. and H.-A. Jacobsen (2004). Resolving feature convolution in middleware systems. 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, Vancouver, BC, Canada, ACM.

Appendix A. APIs of the Selected Infrastructures

A.1 Siena API

Siena provides a simple API with calls that allow the subscription and publication of events as follows:

- *void publish(Notification e)* publish a notification.
- *void subscribe(Filter f, Notifiable n)* subscribes for events matching Filter f.
- *void subscribe(Pattern p, Notifiable n)* subscribes for sequences of events matching pattern p. A pattern is a list of *Notifiable* event templates.
- *void unsubscribe(Filter f, Notifiable n)* cancels the subscriptions, posted by n, whose filter f' is covered by filter f.
- *void unsubscribe(Notifiable n)* cancels all the subscriptions posted by n.
- *void unsubscribe(Pattern p, Notifiable n)* cancels the subscriptions, posted by n, whose pattern p' is covered by pattern p.

The subscription is supported by *Filter* and *Pattern* objects that are manually assembled using wildcard attributes to match events (Notification instances) by their content. Additional calls control the advertisement of events and the federation of servers in the network. A code sample containing Siena's basic publication and subscription operations is presented in Table 14 as follows.

Table 14 Producing and consuming events with Siena

```
// connecting to Siena Server
   HierarchicalDispatcher mySiena = new HierarchicalDispatcher();

// Subscriber listener interface:
//receives notifications (or events)
   Notifiable subscriber;

// posting a subscription
   Filter f = new Filter();
   f.addConstraint("message", OP.ANY, null);
   mySiena.subscribe(f, subscriber);
   ...
// publishing an event
// events are instances of Notification in Siena
   Notification n = new Notification();
   n.putAttribute("message","Hello, World!");
   mySiena.publish(n);
```

A.2 CORBA-NS API

A schematic representation of CORBA-NS's main APIs (or interfaces) is presented in Figure 70 as follows. The picture shows all possible types of consumers and suppliers of events as well as the administrative interfaces of the service. Note that through backward compatibility with CORBA-ES, the CORBA-NS supports all the existing CORBA-ES interfaces. The abbreviations in Figure 70 correspond to: EC – Event Channel original interfaces, ECA – Event Channel Administrative Interfaces, NC – the extended Notification Channel Interfaces, and NCA – the extended Notification Channel Administrative interfaces. The (*) indicates the possibility for multiple instances of an interface.

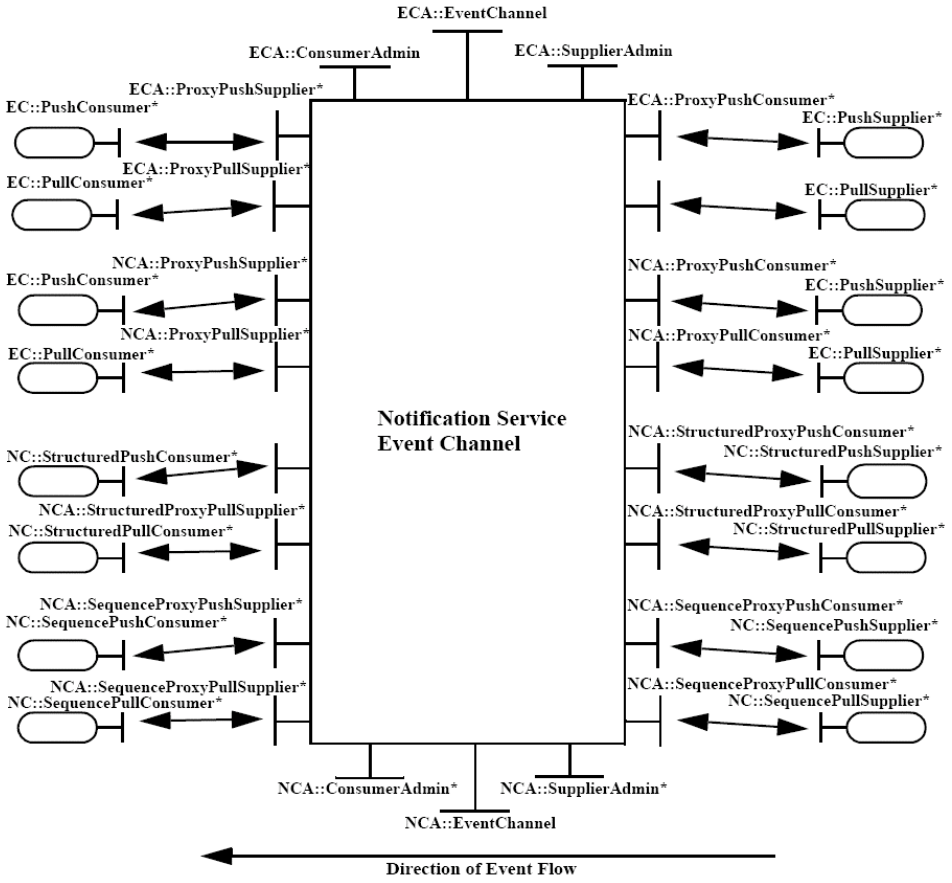


Figure 70 CORBA-NS Architectural overview (source (OMG 2004))

Architecture wise, the *EventChannelFactory* is the main Façade of the CORBA-NS. It allows the notification service users (consumers and suppliers) to create independent event channel instances according to different qualities of service (for example, through the definition of: queue size, time to live and number of clients per channel). This dynamic characteristic is shared by all the components of CORBA-NS. The architecture of the service is also hierarchical in nature, and all objects defined as part of an event channel are created by some parent object. For instance, consumer and supplier admin instances are created by event channels, and all proxy objects (the client access points to the channel) are created by some admin instance. The CORBA-NS is part of an ecology of CORBA services. As such, Naming and communication are usually provided by

extra CORBA services such as CORBA Naming Service (COS Name), and the CORBA-ORB respectively. Summary of CORBA-NS design principles:

- Independent event channels with their underlying queues and qualities of service
- Hierarchical and dynamic structure of components (channels, filters, suppliers and consumers)
- Support for multiple suppliers and producers per channel
- Integration with underlying ORB protocol
- Dependence on existing COSServices: Persistency, Evaluation, Name Services.
- Backward compatibility with the Event Channel specification

The code showing a simple production and consumption of events using CORBA-Ns is shown in Table XV as follows.

Table XV and consuming events with CORBA-NS (exception handling is omitted)

```
      org.omg.CORBA.ORB orb;
      org.omg.CORBA.Object obj;

   // connect to the notification server
      obj = orb.resolve_initial_references( "NotificationService" );
      EventChannelFactory eventChannelFactory;
      eventChannelFactory = EventChannelFactoryHelper.narrow( obj );
      Property[] initialQoS = new Property[ 0 ];
      Property[] initialAdmin = new Property[ 0 ];
      org.omg.CORBA.IntHolder channelId = new org.omg.CORBA.IntHolder();
      eventChannel = eventChannelFactory.create_channel(
                                          initialQoS, initialAdmin,channelId);
      objRef.set( eventChannel );
      ...

   // subscribe to an event channel of push consumer type
      ConsumerAdmin consumerAdmin = eventChannel.default_consumer_admin();
      org.omg.CORBA.IntHolder proxyId = new org.omg.CORBA.IntHolder();
      ProxySupplier proxySupplier = null;
      proxySupplier = consumerAdmin.obtain_notification_push_supplier
                                  (ClientType.ANY_EVENT, proxyId);
      ProxyPushSupplier proxyPushSupplier = null;
      proxyPushSupplier = ProxyPushSupplierHelper.narrow(proxySupplier);
      PushConsumerPOA subscriber = (PushConsumerPOA) new MySubscriber();
      proxyPushSupplier.connect_any_push_consumer(
                                          subscriber.pushConsumer());
   // set the filtering parameters
   // (domain = "OpenORB", type = "HelloWorld", and a data filter)
      ConstraintExp constraints[] = new ConstraintExp[1];
      constraints[0] = new ConstraintExp();
      constraints[0].event_types = new EventType[1];
      constraints[0].event_types[0] = new EventType();
      constraints[0].event_types[0].domain_name = "*";
      constraints[0].event_types[0].type_name = "*";
      constraints[0].constraint_expr ="$.filterable_data(filter:content) ==
                                    'HelloWorld' ";
      ConstraintInfo[] add_constraints_results;
```

```
        Filter filter = null;
        add_constraints_results = filter.add_constraints(constraints);
        int filter_id = proxyPushSupplier.add_filter(filter);
        ...

    // publish an event supplier
        org.omg.CORBA.IntHolder proxyId = new org.omg.CORBA.IntHolder();
        ProxySupplier proxySupplier = null;
        proxySupplier = consumerAdmin.obtain_notification_push_supplier
                                    (ClientType.ANY_EVENT, proxyId);
        ProxyPushSupplier proxyPushSupplier = null;
        proxyPushSupplier = ProxyPushSupplierHelper.narrow(proxySupplier);
    // create an event
        String eventTypeName = "Message";
        String message = "HelloWorld";
    // Event type has domain_name and type_name
        EventType eventType = new EventType( "OpenORB", "HelloWorld" );
        FixedEventHeader fixedEventHeader = new FixedEventHeader(
                                              eventType,   eventName.toString()
);
    // Event - Variable header
        Property[] variableHeader = new Property[2];
        variableHeader[0] = new Property("variable:type", orb.create_any());
        variableHeader[0].value.insert_string( eventTypeName );
        variableHeader[1]= new Property("variable:content",orb.create_any());
        variableHeader[1].value.insert_string( message );
        EventHeader    eventHeader    =    new    EventHeader(fixedEventHeader,
                                                  variableHeader);
    // Event - Filterable data declaration
        Property[] filterableData = new Property[2];
        filterableData[0] = new Property("filter:type", orb.create_any());
        filterableData[0].value.insert_string( eventTypeName );
        filterableData[1] = new Property("filter:source",orb.create_any());
        filterableData[1].value.insert_string( message );
        Any msg = orb.create_any();
        msg.insert_string( msgBody );
        StructuredEvent event = new StructuredEvent (eventHeader,
                                                     filterableData, msg);
        Any anyEvent = orb.create_any();
        StructuredEventHelper.insert( anyEvent, event );

    //publish the event
        proxyPushConsumer.push( event );
```

The CORBA-NS allows the definition of filters in their event channels. These filters are expressed using the TCL (Trader Control Language). Some examples of queries using this language are presented in Table XVI as follows.

Table XVI CORBA-NS event filter language examples

Accept all *CommunicationsAlarm* events but no lost_packet messages:

```
$event_type == 'CommunicationsAlarm' and
not ($event_name == 'lost_packet')
```

Accept *CommunicationsAlarm* events with priorities ranging from 1 to 5:

```
($event_type == 'CommunicationsAlarm') and
($priority >= 1) and ($priority <= 5)
```

Select MOVIE events featuring at least three of the Marx Brothers:

```
($event_type == 'MOVIE') and ((('groucho' in $.starlist) +
('chico' in $.starlist) + ('harpo' in $.starlist) +
('zeppo' in $.starlist) + ('gummo' in $.starlist)) > 2)
```

Accept only recent events:

```
$origination_timestamp.high + <>2 < $curtime.high
```

Accept students that took all three tests and had an average score of at least 80%:

```
($.test._length == 3) and ((($.test[1].score + $.test[2].score
+ $.test[3].score) / 3) >= 80)
```

Select processes that exceed a certain usage threshold:

```
$memsize/5.5 + $cputime * 1275.0 + $filesize * 1.25 > 500000.0h
```

A.3 JavaSpaces API

The tuple space model as implemented by IBM TSpaces (Wyckoff 1998) and Sun JavaSpaces (Freeman, Hupfer et al. 1999) combines the traditional Linda API with DBMS features such as transactional semantics, allowing, for example roll-back of operations, access control, and event notification (applications can register to be notified whenever the tuple space is changed). In our case studies, we used the open source implementation of JavaSpaces provided by SUN. The basic primitive operations supported by JavaSpaces are:

- *long[] write(EntryRep tuple, Transaction txn, long lease)* Adds a tuple to the space, equivalent to a publish command. Transaction and lease are optional parameters
- *Object take(template_tuple, transaction, timeout, query_cookie)* Performs an associative search for a tuple that matches the template. When found, the tuple is removed from the space and returned. If none is found, returns null. Transaction, timeout and query_cookie are optional parameters.
- *Object takeIfExists(EntryRep tmpl, Transaction txn, long timeout, QueryoCookie cookie)* Performs an associative search for a tuple that matches the template. Blocks until match is found. Removes and returns the matched tuple from the space. Transaction, timeout and query_cookie are optional parameters.

- *Object read(EntryRep tmpl, Transaction txn, long timeout, QueryCookie cookie)* Same as the "*take*" command above, except that the tuple is not removed from the tuple space.
- *Object readIfExists(EntryRep tmpl, Transaction txn, long timeout, QueryCooie cookie)* Same as the "*takeIfExists*" command above, except that the tuple is not removed from the tuple space.
- *EventRegistration notify(EntryRep tmpl, Transaction txn, RemoteEventListener listener, long lease, marshalledObject handback)* Registers a listener to entries matching the provided template. Whenever a match occurs (on write() commands), a notification is sent to the listener interface together with a handback that provides more information about the entry that matched the template.
- *contents(EntryRep[] tmpls, Transaction tr, long leaseTime, long lmit)* Same as the "read" command above, except returns the entire set of tuples that match the templates provided.

A sample of the use of tuple spaces to publish and subscribe to events is presented in Table XVII as follows.

Table XVII Producing and consuming events with JavaSpaces (exception handling is omitted)

```
// connecting to the tuple space
// Alternative and shorter way

   Class[] classes = new Class[]{ aServiceInterface };
   ServiceTemplate tmpl = new ServiceTemplate(null, classes, null );

   // Locate the JavaSpaces service and create a JavaSpace
   //proxy attached to it.
   LookupLocator locator = new LookupLocator(address);
   ServiceRegistrar sr = locator.getRegistrar();
   JavaSpace space =  (JavaSpace)sr.lookup(tmpl);

// writing to the tuple space
   Tuple t = new Tuple();
   t.stringField  = " some value";
   t.intField = new Integer(123);
   // publish the event with a minute lease time
   space.write(t, null, 60 * 1000);
   ...

// subscribing to notifications
// null = wild card
   Tuple template = new Tuple("Key2", null);
   reg = space.notify(template, null, tsListener,
                  Lease.FOREVER, null);

// we need to implement TSLisetener that will read the tuple from
// the space when a notification is received.
```

A.4 YANCEES client-side API

For the point of view of the application engineers that use different YANCEES instances, the API is similar to Siena. It provides methods for publishing and subscribing to events. Unlike Siena, that uses object templates (or anti-tuples) as filters and patterns, YANCEES uses extensible subscription and notification languages. For the point of view of the infrastructure software engineer, the extensibility and configurability API requires the use of abstract classes, interfaces and configuration files. A code example containing the basic subscription and publication operations in YANCEES is presented in Table XVIII as follows.

Table XVIII Producing and consuming events with YANCEES (exception handling is omitted)

```
//connect to YANCEES server
   YanceesRMIClient client;
   client = new YanceesRMIClient("hostname.mydomain.com");
   ...

// publishing an event
   YanceesEvent event = new YanceesEvent();
event.put("name", "Roberto");
event.put("Office", 247);
...

// subscribe to events
   GenericMessage msg = new GenericMessage("" +
   " <subscription>" +
   "   <filter>" +
   "     <EQ>" +
   "       <name> name </name>" +
   "       <value type=\"yanceesString\"> Roberto </value>" +
   "     </EQ>"+
   "   </filter>" +
   " </subscription>");

   // 'this' implements YanceesClientInterface
client.subscribe(msg, this);
```

Appendix B. Extending YANCEES

B.1 Case study: implementing CASSIUS services with YANCEES

To illustrate the use of the architecture extensibility and configurability, this section presents some examples on how to implement plug-ins in YANCEES. It also shows how YANCEES can be customized to provide the functionality required by different application domains.

For example, suppose that YANCEES needs to be adapted to support awareness applications, providing a set of features similar to CASSIUS, which requires: event persistency, content-based filtering, sequence detection, and the pull notification. Moreover, a special feature provided by CASSIUS is the ability to browse and later subscribe to the event source hierarchies. This feature, called event source browsing, provides information about the publishers and the events they publish.

B.1.1 Implementing a sequence detection subscription command

Sequence detection requires the extension of the subscription model with the addition of a new keyword <**sequence**>. It will operate over a set of content-based filters, which are already supported by YANCEES baseline configuration.

The first step is to extend the YANCESS subscription language with the new <**sequence**> tag. This is illustrated in the code fragment as follows**.**

```
<complexType name="SequenceSubscriptionType">
  <complexContent>
    <extension base="sub:SubscriptionType">
      <sequence minOccurs="0" maxOccurs="1">
        <element name="sequence" type= "FilterSequenceType" />
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="FilterSequenceType">
  <sequence minOccurs="1" maxOccurs="unbounded">
    <element name="filter" type="FilterType"/>
  </sequence>
</complexType>
```

The next step is to implement the sequence detection plug-in, extending the subscription model. For such, developers may choose to extend *AbstractPlugin,* a convenience class that pro-

vides default implementations to YANCEES *PluginInterface*, which methods are described below.

```
interface PluginInterface extends PluginListenerInterface {
  long getId();
  String getTag();
  String getFullContext();
  String getFullPath();
  Node getSubtree();
  void addListener (PluginListenerInterface plugin);
  void removeListener (PluginListenerInterface plugin);
  void addRequiredPlugin (PluginInterface plugin);
  PluginInterface[] getRequiredPluginsList();
  boolean hasChildren();
  void dispose();
}
```

Note that the *PluginInterface* is a listener to events produced in other plug-ins. As such, it implements the *PluginListenerInterface* as follows.

```
interface PluginListenerInterface {
  void receivePluginNotification (EventInterface evt, PluginInterface
source);
  void receivePluginNotification (EventInterface[] evtList, PluginIn-
terface source);
}
```

A simple sequence detection implementation will collect events, in the right order, that come from two or more content-based filter plug-ins. When a successful sequence is detected, the sequence plug-in returns the set of events collected, publishing it to higher-level plug-ins (listeners) as an array of *YanceesEvent* objects. Note that we are assuming that the event dispatcher guarantees the in-order delivery of events. If this is not the case, more complex algorithms must be used.

In order to be dynamically loaded, at runtime, every plug-in must provide a factory implementation that implements the *PluginFactoryInterface* as follows.

```
interface PluginFactoryInterface {
  String[] getTags();
  PluginInterface createNewInstance (Node subTree);
}
```

A simple factory implementation will return a new instance of the plug-in each time the *createNewInstance*() method is invoked in its interface. The plug-in factory must then be registered under the "*sequence*" tag name in the YANCEES configuration file as described below.

```
<subscription>
...
  <plugin>
    <name> sequence.plugin </name>
    <mainClass>
      <javaClassName>
        plugin.sequence.SequencePlugin
      </javaClassName>
    </mainClass>
    <factoryClass>
      <javaClassName>
        plugin.sequence.SequencePluginFactory
      </javaClassName>
    </factoryClass>
    <depends> siena.plugin </depends>
  </plugin>
...
</subscription>
```

The plug-in is then ready to be used. It will be activated each time a subscription is provided that uses the <**sequence**> tag as its part. An example of a subscription using this new extension is presented in the code below. The Java DOM parser automatically checks the subscription for syntax errors by using the XML schema definition of the <**sequence**> command.

```
<subscription>
  <sequence xsi:type="FilterSequenceType">
    <filter xsi:type="FilterType">
      <EQ>
        <name> status</name>
        <value> Fail </value>
      </EQ>
    </filter>
    <filter xsi:type="FilterType">
      <LT>
         <name> cooler Temp </name>
         <value> 90 </value>
      </LT>
    </filter>
  </sequence>
</subscription>
<notification>
  <push>
</notification>
```

B.1.2 Pull delivery mechanism implementation

Pull delivery allows subscribers to periodically poll (or check) the server for new events matching their subscriptions. This mechanism copes with the requirements of some mobile applications, where subscribers usually get temporarily disconnected.

This mechanism is provided by a pull notification plug-in. In order to temporarily store the events that are not being delivered, the pull mechanism needs an event persistency service (or static plug-in). As a consequence, together with the pull notification plug-in, an event persistency service must also be defined.

Users need to control when to collect and when to store the events being routed to them as a result of a subscription. This usually requires a polling interaction protocol. This interaction is not part of the regular *publish*() and *subscribe*() commands of a notification server, so a protocol plug-in must be defined. In short, the implementation of a pull delivery mechanism requires:

- The extension of the notification language to add pull support
- The implementation of a pull notification plug-in
- The implementation of a persistency service
- The definition of a polling protocol
- The implementation of a polling protocol plug-in

The implementation of the pull notification plug-in follows the same steps as the sequence detection plug-in previously described. The same is true for the notification language extension. An extension is provided to the notification language that defines the **<pull>** tag. Additionally, a factory to instantiate this plug-in is also provided. In order to activate the pull plug-in, a **<pull>** tag must be provided in the **<notification>** session of a subscription message (see Table 6). As a consequence, a pull plug-in instance is created and registered to handle the events that match the subscription.

The pull plug-in implementation is very simple; it directs the events to the persistency service component and registers them under their target subscriber interface.

The poll plug-in responds to commands such as **<poll-interval>, <stop-polling>** and **<poll>**, which define different polling mechanisms. It collects the events stored in the persistency service, and delivers them periodically to the subscriber (poll-interval command); it then collects the notifications whenever requested (using the poll command) or deactivates the periodic delivery (using the stop-polling command) in case of a temporary disconnection.

These sets of plug-ins define a configuration, a set of components that need to be present in order for a service to operate. The dependencies between these are checked by YANCEES with the help of the <**depends**> clause in the configuration file.

B.1.3 Implementing CASSIUS features

In addition to the features described in the previous sessions, CASSIUS provides event typing and the ability to browse through hierarchies of event sources.

The browsing of event sources in CASSIUS allows publishers to register events in a hierarchy based on accounts and objects. This model and the API required to operate the server are described elsewhere.

In the YANCEES framework, the CASSIUS functionality is implemented by the use of protocol plug-ins and a *CassiusService* component. The CASSIUS protocol plug-in interacts with the *CassiusService*, which allows the creation and management of objects, accounts, and their events. These operations include registering/un-registering accounts, objects, and events, as well as polling commands.

CASSIUS uses events with a fixed set of attributes. These events can be easily identified and checked for correctness by an input filter. This filter checks all incoming events for the proper CASSIUS template format. Once a CASSIUS event is identified and validated, it is copied to the *CassiusService*, which stores it in a database in its proper account/object record.

Polling of events, in this case, is handled by the CASSIUS protocol plug-in, which allows the collection of events by account, object, or sub-hierarchies. Note that this approach does not prevent the simultaneous installation of both services, the simple pull and the CASSIUS pull protocol.

The poll mechanism is not the only way to collect CASSIUS events. At any time, subscriptions can also be performed on regular CASSIUS events.