ISR Institute for Software Research
University of California, Irvine

A Survey of Versatility for Publish/Subscribe Infrastructures

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

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

May 2005

ISR Technical Report # UCI-ISR-05-8

Institute for Software Research
ICS2 210
University of California, Irvine
Irvine, CA 92697-3425
www.isr.uci.edu

http://www.isr.uci.edu/tech-reports.html

A Survey of Versatility for Publish/Subscribe Infrastructures

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-05-8

May 2005

Abstract:

Current publish/subscribe middleware infrastructures fall short of mechanisms that allow their customization and configuration to comply with the requirements of different application domains. This shortcoming is a consequence of their original design which does not account for mechanisms or approaches that allow the evolution of this kind of service.

This survey introduces the concept of versatility in publish/subscribe infrastructures and examines the current approaches to versatility in publish-subscribe middleware as well as approaches to versatility that have been applied in other kinds of middleware and may possibly succeed in the context of publish/subscribe infrastructures.

In this context, versatility is defined as a set of properties (such as variability, reuse, dynamism and usability) that allows the customization, extension and compression of middleware. This paper surveys existing and advanced software engineering approaches to address those requirements. A comparative framework on software versatility, as a set of properties, is presented to help researches and practitioners to evaluate and compare the strengths and limitations of such approaches that have been or might be applied to this problem. Our goal is not to compare the approaches with one another, but to show how those approaches can be used to provide some of the versatility properties we identify. An agenda for future research in this topic is also presented.

This survey addresses the following questions: What is versatility? How is versatility defined in the context of publish/subscribe middleware? Which software engineering techniques have been used to provide versatility to middleware in general, and specifically to publish/subscribe middleware? What other techniques may be used to approach this problem? What are their limitations and strengths? What are some of the important open research questions in this area?

A Survey of Versatility for Publish/Subscribe Infrastructures

Roberto S. Silva Filho and David F. Redmiles
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3430 USA

{rsilvafi, redmiles}@ics.uci.edu

Abstract

Current publish/subscribe middleware infrastructures fall short of mechanisms that allow their customization and configuration to comply with the requirements of different application domains. This shortcoming is a consequence of their original design which does not account for mechanisms or approaches that allow the evolution of this kind of services.

This survey introduces the concept of versatility in publish/subscribe infrastructures and examines the current approaches to versatility in publish-subscribe middleware as well as approaches to versatility that have been applied in other kinds of middleware and may possibly succeed in the context of publish/subscribe infrastructures.

In this context, versatility is defined as a set of properties (such as variability, reuse, dynamism and usability) that allows the customization, extension and compression of middleware. This paper surveys existing and advanced software engineering approaches to address those requirements. A comparative framework on software versatility, as a set of properties, is presented to help researches and practitioners to evaluate and compare the strengths and limitations of such approaches that have been or might be applied to this problem. Our goal is not to compare the approaches with one another, but to show how those approaches can be used to provide some of the versatility properties we identify. An agenda for future research in this topic is also presented.

This survey addresses the following questions: What is versatility? How is versatility defined in the context of publish/subscribe middleware? Which software engineering techniques have been used to provide versatility to middleware in general, and specifically to publish/subscribe middleware? What other techniques may be used to approach this problem? What are their limitations and strengths? What are some of the important open research questions in this area?

Keywords

Middleware versatility, publish/subscribe, software engineering, variability

1 Contents

1.1 Table Index

1.2 Figure Index

2 Introduction

Publish/subscribe infrastructures (or publish/subscribe for short) are message-oriented middleware (or MOMs) that implement the publish/subscribe architectural style. This architectural style provides an inherent loose coupling communication mechanism between information publishers and consumers, which defines clear separation of communication from computation and carries the potential for easy integration of autonomous, heterogeneous components into complex systems that are easy to evolve and scale (Dingel, Garlan et al. 1998). They also provide a one-to-many communication mechanism with which multicast and broadcast-based applications can be implemented.

For such characteristics, they have been used as the basic communication and integration infrastructure for many application domains such as software monitoring, awareness, enterprise integration, groupware, distributed user monitoring and so on. This wide range of applications had required new services from the infrastructure such as advanced event processing (event sequence detection, abstraction, and summarization); event persistency, mobility support, transactions, security communication channels, and a whole new set of domain-specific features. As a consequence, in spite of the availability of standardized solutions such as CORBA-NS (CORBA Notification Service) (OMG 2002) or JMS (Java Message Service) (SUN 2003), new notification servers continue to be developed to address the needs of novel applications such as user and software monitoring (Hilbert and Redmiles 1998), groupware (Dourish and Bly 1992), collaborative software engineering (Sarma, Noroozi et al. 2003), workflow management systems and mobile applications (Cugola, Nitto et al. 2001), among others.

In this context, the proliferation of specialized solutions reveals limitations on the way event-based infrastructures are being designed and implemented. First and foremost, the publish/subscribe paradigm appears seductively simple. A basic service can be programmed quickly before the complexities of the application it serves reveal themselves. Then, when complexities manifest, they require significant extensions already implemented in existing, sophisticated infrastructures. A second deterrent is that current publish/subscribe infrastructures are not designed to be extensible nor programmable, which hinders the addition or customization of new application services. For instance, CORBA-NS does not support event source discovery protocols, such as those provided by CASSIUS (Kantor and Redmiles 2001). The implementation of this feature using CORBA-NS would require the direct change of the publish/subscribe service source code or even aspects of the client application. Third, with rare exceptions such as the READY (Gruber, Krishnamurthy et al. 1999) (a CORBA compliant notification service), current solutions are not configurable with respect to the place where event processing happens in a distributed setting, a feature important in some application domains. For instance, some applications such as software monitoring (Hilbert and Redmiles 1998), require the execution of event processing on the application side where the events are collected, whereas applications running on mobile devices may need a restricted set of services and components. Forth, with the proliferation of specialized middleware, interoperability becomes a problem. In large organizations, for the reasons previously mentioned, it is common to find different event-driven applications, designed for specific purposes, that rely on different event-based infrastructures. Due to differences in purpose and scale, they usually do not interoperate. For example, server monitoring applications, e-mail servers, workflow management systems and so on, that do not share a common data format, data schema or even computing platform. Finally, with the exception of a few research prototypes such as YANCEES (Silva-Filho, Souza et al. 2004) and some others, none of existing event-based middleware approaches support a more generalized selection and customization of features that the publish/subscribe infrastructure should provide.

Those limitations motivated our survey of existing software engineering techniques and approaches that have been or can be used to provide versatility to publish/subscribe infrastructures. The term versatility was chosen to embrace the set of good software engineering requirements that would improve the support for customization and evolution of publish/subscribe middleware. Our concept of versatility is a combination of the following software properties: extensibility, programmability, reuse, dynamic and static variability and usability.

Hence, in this survey, we present the concept of versatility and classify existing systems according to this new concept. We also survey existing software engineering versatility techniques and present their strengths and limitations, discussing how they can be applied to address some of the versatility properties in the context of publish/subscribe infrastructures. We expect with that to provide a framework that allows software engineering practitioners and researches to choose one or another according to their needs.

In the next section, we present our goals and research methods in the preparation of this survey. In section 0 we motivate and scope our problem by presenting three application scenarios that require different services from the publish/subscribe infrastructure. Section 6 provides an overview of publish/subscribe technology and some background in the area of publish/subscribe infrastructures. Section 6 defines our concept of software versatility in terms of good software qualities. Section 7 surveys existing software engineering, architectures and applications that can be used to address the versatility properties from section 6. Section 8 presents some related techniques. After that, section 10 surveys existing publish/subscribe infrastructures, both versatile and non-versatile systems. Section 11 correlates the existing publish/subscribe infrastructures with some of the versatility approaches presented in section 6. After that, section 12 briefly presents some promising research topics. Finally, section 12 draws some conclusions and observations resulting from this survey.

3 Goals and research method

The fundamental goal of this survey is to identify and motivate the need for versatility in publish/subscribe infrastructures, and define versatility in terms of key properties the publish/subscribe infrastructure must provide, pointing possible approaches to the problem. In order to do so, we divide this survey in two main parts:

In the first part, we enlist a set of software versatility approaches that have been or may be used in publish/subscribe domain. Whenever possible, we present examples and relate to existing publish/subscribe infrastructures. The goal of this first part is to help middleware researchers, designers and practitioners to categorize, evaluate, and compare the strengths and limitations of those approaches, when applying them to the publish/subscribe problem.

In the second part, we survey existing publish/subscribe infrastructures, identifying how they address the versatility properties we propose. Because the survey of all existing systems and technologies is impossible, a representative set of each class of system is presented. For such, we attempt to present those systems according to their most prominent characteristics, thus dividing them in more or less natural categories at the end of the survey.

In the realization of this survey, we first searched the existing literature and the web for both industrial and scientific publications on publish/subscribe infrastructures, paying special attention on how they can be used to support the requirements of different application domains, and which mechanisms they provide to address the versatility properties we propose. Then, we identified the need for versatility in publish/subscribe middleware as a set of good software engineering qualities a system must have in order to better support its evolution, configuration, programmability and usability.

After surveying existing systems and identifying how they address the versatility issues, the existing literature was searched for existing software engineering techniques and programming language approaches that are being used in middleware other than publish/subscribe, or even techniques we think can be used to provide those qualities to publish/subscribe infrastructures. As a result, a matrix was built where we compared the characteristics of current publish/subscribe infrastructures with the versatility qualities that we propose, showing how those systems provide such properties. The resulting matrix allows the identification of main limitations in current infrastructures, and allows the visualization of areas which further research is necessary. Throughout the survey, a set of tables were build which list the each versatility approach and explains how they can address the versatility requirements we propose. The resulting set of tables allow the visualization of techniques that can be applied or tested in the implementation of the versatility properties in the context of publish/subscribe infrastructures.

4 Application scope and motivation scenarios

In order to understand the kinds of infrastructures we are interested in surveying, we list three examples of applications in different domains such as: software monitoring (EDEM), awareness-based applications (CASSIUS), and peer-to-peer synchronous collaborative environments, and briefly describe how publish/subscribe infrastructures are used in those domains. The goal is to give the reader a feeling of the set of requirements those systems need to support.

First, the EDEM (Expectation Driven Event Monitoring) system (Hilbert and Redmiles 1998) is a user interface monitoring tool. In EDEM, events collected from end-user‘s interaction with the system are intercepted and compared with expected use scenarios. Those scenarios are defined by the application developers and described in terms of user interface event sequences. They are deployed with the final application instrumented with EDEM, which monitors the application and detects whenever those expectations are broken. The results are then sent back to the developers using the Internet, for further analysis. In this application, the event processing and the notification delivery are all performed in the application site, whereas expectations represented as agents (or subscriptions) are deployed to that site. Thus, there is no central server, and few data is transmitted through the network. The event processing language is complex and requires advanced event correlation features as event summarization, event abstraction (generation of events in response to a pattern detection) and sequence detection.

In another example, the CASSIUS notification server (Kantor and Redmiles 2001) is used as an awareness information router and integrator. Events are collected from many sources possibly including sensors, application execution traces, webcams and others, and are delivered to interested parties based on different policies (subscriptions), where they can be used to implement different awareness visualizations and mechanisms. A distinctive feature of CASSIUS is its ability to define event hierarchies, and collect information about event sources, allowing end-users to browse through different event sources and hierarchy spaces when building their subscriptions. Another important feature is persistency, the ability to store events in different user accounts for further reference, coping with mobility and disconnection of clients. Hence, in this application, the architecture is centralized, subscriptions are based on event sources and types, and the event’s content is multi-modal. Events are less frequent and more diverse in their content, if compared to the software monitoring scenario.

In a third example, we need to support a peer-to-peer file sharing tool where security visualizations are used to help end users assess their current security (DePaula, Ding et al. 2005). In this application, an ad-hoc collaboration tool is built on top of a peer-to-peer publish/subscribe infrastructure able to collect events from distributed WebDAV repositories and to synchronize the GUI visualization as file visibility properties are changed (read only, read-write, full control and

so on). In this context, events are used to leverage security awareness. They provide insight about the application execution and allow end-users to assess the security of the system, visualizing interactions in the shared artifacts as they collaborate. In this scenario, the publish/subscribe infrastructure is decentralized (Peer-to-Peer), and need to execute in small devices such as palm-size computers, that perform "real-time" visualizations. The event frequency is high, and their content is small (representing Web-DAV access events such as PUT, GET, PROPFIND, PROPPATH, and so on). Events also indicate changes in the user interface such as drag and drop of files, changes in visibility and others.

Other examples include application sharing, software monitoring (Naslavsky, Silva Filho et al. 2004), and awareness in general. On all these scenarios, our challenge is to use a single publish/subscribe infrastructure to provide the functionality demanded by each one of those applications, and to be able to evolve in order to address new applications to come.

Hence, our main motivating problem and scope of this survey is study how to provide versatility to publish/subscribe infrastructures that are used for information integration and communication in collaborative settings. We want an infrastructure that is able to integrate and process information from different sources, and deliver this information to different interested parties in a variety of ways. Moreover, we want the infrastructure to be customizable and extensible to address the requirements of different application domains. Considering these goals, publish/subscribe infrastructures are appealing to our research for their ability to isolate producers and consumers of information, for their scalability in the sense of supporting many information producers and consumers, and the ability to process those events into higher-level pieces of information. The challenge, however, is to support the specific requirements of each one of those applications in a common infrastructure that can be adapted, extended and customized for their needs, without loosing its integration and interoperability characteristic. Those requirements are usually associated to the subscription and notification languages, as well as the event representation and different infrastructure and interaction protocols.

5 Background and definitions

This section serves two purposes: first it introduces publish/subscribe infrastructures, their main characteristics and components such as "notification service", "publishers", "subscribers", "events", "notifications" and so forth; Second it analytically describes the main components of such systems with respect to a proposed design framework, allowing a better comprehension of the publish/subscribe technology and the main characteristics of the systems it provides.

Middleware refers to the software layer, between applications and the network protocols, that supports software engineers in developing distributed applications. Historically, middleware has been used to address issues related to heterogeneity, communication, and distribution of software components, relieving software engineers from the burden of solving low-level, network issues, such as lower-level communication protocols, concurrency control, transaction management, distributed object location, among others. Thus, middleware allows software engineers to focus on the actual application requirements, relieving them from communication and coordination details, which facilitates the development of high-quality software with less coding (Emmerich 2000). Because of these advantages, middleware such as RPC (Remote Procedure Call) and TP (Transaction Processing) monitors had become very popular. In fact, in recent years, other standardized solutions such as the OMG CORBA (Common Object Request Broker Architecture) (Siegel 1998) and SUN JMS (Java Message Service) (Sun Microsystems 2003) along with their many implementations have been used as a basic platforms for the development of a large spectrum of distributed applications. While CORBA defines a standard object-oriented communication broker based on a distributed implementation of the remote method invocation paradigm (the IIOP –

Internet Inter-Orb Protocol), JMS and their implementations are examples of message-oriented middleware (or MOM in short), which main goal is to integrate components in a distributed system through the exchange of asynchronous messages (that usually represent system or "real world" events). In this survey, we are interested in the former technology, i.e. publish/subscribe infrastructures.

A *publish/subscribe service* implements a distributed publish/subscribe architectural style. It provides a logically centralized service that mediates the communication between publishers and consumers of information in distributed system. Applications that are built upon this architectural style are also known as event-driven applications. In these systems, some components (or information producers) announce (or broadcast) events, while other components (information consumers) advertise their interest in these events. This is performed by means of subscriptions. In this survey, the word *subscription* denotes the act of expressing interest on some specific content, which can be performed in different ways such as: opening a communication channel between two or more parties, posting a filter expression, defining rules and queries on parts of this information content, becoming part of a group where this content is produced, and many other ways. Subscriptions may be revoked by an unsubscribe command or similar operation, and can be changed by unsubscribing the existing one and posting a new one or similar approaches. An *event* expresses a state change in a (possibly distributed) component, or represents some temporal fact in the world. An event is computationally expressed in the form of a message, which conveys content or information about this event. Hence, a message can have different representations, such as plain text, programming language records, objects, tuples (attribute/value pairs) and so on. On the course of this survey, we use the terms events and messages interchangeably, referring to their computational representation.

As illustrated in Figure 1, a publish/subscribe service (a.k.a. notification service) is responsible for receiving: (1) subscriptions, coming from *event consumers*, and (2) events, coming from their *producers*. With these two sets of information, it efficiently performs the matching of subscriptions with their corresponding subset of events, routing the resulting events, as notifications, to the interested parties.

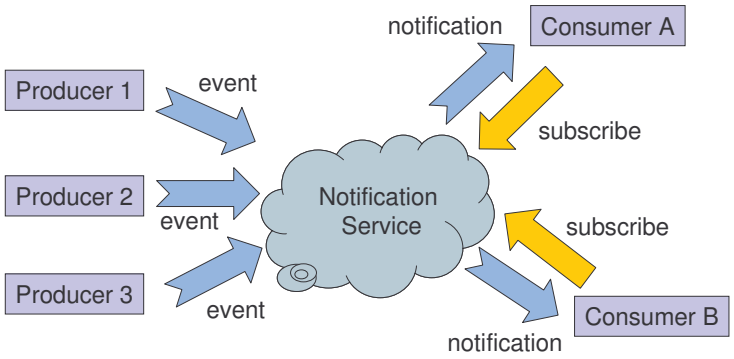

Figure 1 Basic components in a distributed publish/subscribe system.

From the software architectural point of view, event notification services provide a logically centralized service, which is usually implemented by a single server or a set of federated servers. Since all communication is mediated by this service, it represents a loosely coupled and asynchronous communication mechanism that facilitates the one-to-many or many-to-many combination and routing of information between heterogeneous components. As a result, this communication paradigm provides interesting characteristics to the applications such as: the interacting par-

ties do not need to know each other (publisher and subscriber anonymity), which provides location transparency for both producers and consumers; publishers and subscribers do not need to be up at the same time (time decoupling), and the publishing and subscription of events do not block participants (decoupling with respect to event flow). For such characteristics, the publish/subscribe paradigm has been largely used for application-level communication and information systems integration.

Another interesting characteristic of the publish/subscribe paradigm is the diversity that exists in the way the users interact with the system: the way users can publish events, express their interest, and specify how notifications are delivered. In other words, differently from more popular forms of interaction as remote method invocation (RMI) or remote procedure calls (RPC), which rely on pure programmatic interfaces, publish/subscribe infrastructures can support different interaction languages, have events represented in different formats: records, objects, text; can define different notification policies: push, pull, program executions, window pop-ups, and usually support rich subscription languages that are, in many cases, application-specific. The next session motivates all those differences, and shed some light in the diversity existent in current publish/subscribe infrastructures, and the concerns one should have on designing those systems.

5.1 The many flavors of publish/subscribe

Publish/subscribe infrastructures evolved over time, incorporating new functionality and interaction models. They also became specialized, being applied in different application domains (Baldoni, Contenti et al. 2003). Among the first systems to employ the idea of publication and subscription of information were tuple spaces. They were created in response to the need for scalable mechanisms to support concurrent inter-process communication in distributed systems. More recently, both commercial and research MOMs and content-based routing networks have been developed and are being employed in different application domains.

5.1.1 Tuple spaces (or tuple-oriented middleware)

The Tuple space concept was originally proposed by Gelernter as part of the Linda coordination language (Gelernter 1985). A tuple space provides a persistent and shared memory (or space), accessed through an API that allows distributed processes to read, write and remove information represented as tuples (type, attribute, value pairs). In the Linda system, tuples can be concurrently read or removed from the space by different processes. In this programming paradigm, concurrency and interoperability mechanisms can be easily implemented, as well as well as more advanced communication and coordination mechanisms such as distributed queues and locks. Queries on the tuple space can also be defined, allowing interested processes (subscribers) to retrieve existing tuples or to block until tuples matching this query are added to the space. Those queries are type-based (also known as templates or anti-tuples). A template matches a tuple if both have an equal number of fields and each template field matches the corresponding tuple field. Those two mechanisms combined provide a powerful publish/subscribe semantics to this model.

Current examples of systems that implement this model are IBM TSpaces (Wyckoff 1998) and JavaSpaces (Freeman, Hupfer et al. 1999) from Sun. IBM TSpaces, for example, combines the traditional Linda API with DBMS features such as transactional semantics, database indexing, dynamically modified behavior (download and installation of new data types to tuples and new operators); 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.

For having the characteristics presented here, tuple spaces fill the gap between message-oriented middleware and database systems. For not adhering to a fixed database schema, it is more flexible, since it does not restrict the format of the tuples stored nor the types of data the tuple space contains. At the same time, they provide all the assynchronicity and anonymity of publish/subscribe middleware, working as an inter-process communication and the basis for parallel programming and artificial intelligence techniques. For such characteristic, it's becoming more and more popular in mobile and ubiquitous computing applications.

5.1.2 Message-Oriented Middleware

With the popularization of the Internet and the need for scalable infrastructures for enterprise integration, Message-Oriented Middleware (or MOMs) became very popular. They provide a communication infrastructure based on messages (or events), operating as a communication medium between publishers and subscribers. As opposed to the tuple-space model, which was originally developed for inter-process communication and synchronization, by means of a shared persistent data space, MOMs are designed for integration of processes by means of efficient message routing mechanisms. As a consequence, the message persistency is usually optional, and issues such as scalability and efficient delivery of messages are prominent in these systems. In fact, recent studies show that the tuple-space model can be reduced to the publish/subscribe message-oriented model, whereas the reverse is not true. (Zavattaro and Busi 2001). But, due to scalability concerns, the message-oriented publish/subscribe model is preferable for Internet-scale applications.

In spite of their diversity, current MOMs can be subdivided in two main categories, if considered the way consumers express interest in the events of the system, and how these events are routed from their producers to the respective consumers. In the first (and earliest) category, the event dispatching mechanisms is either group based (also known as channel-based) or subject-based (also known as topic-based). In the second more recent approach, the routing is performed according to the whole event content, and is called content-based routing.

Channel (or queue) and subject (or topic)-based routing

In the first category, the difference between subject and group is just implementation related. In the group (or queue)-based approach, producers broadcast events to groups, queues or channels, whereas consumers subscribe to one or more of these channels to receive events. In the topic (or subject)-based approach, publishers are required to annotate each event with a special field, usually a string, called subject (or topic) which describes its content. Information consumers specify their subscriptions based on this specific field. Both approaches are depicted in Figure 2 as follows.

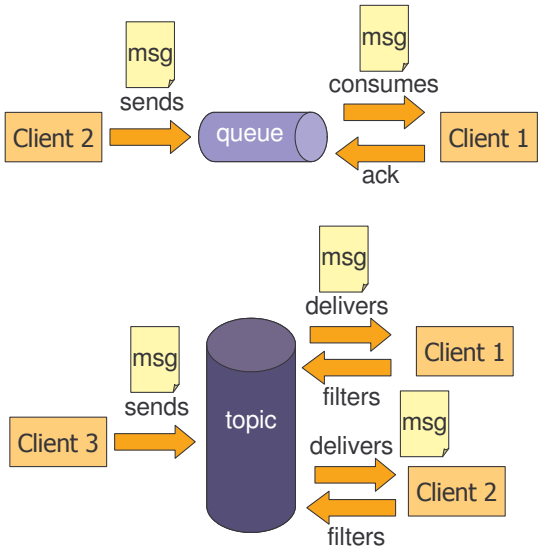

Figure 2: Topic or channel-based routing

In a queue-based model, the interaction is usually peer-to-peer: events sent by one client are routed to another client of the service through a communication queue. In a topic-based style, the communication is many-to-many, and filtering can be applied to distinguish one topic from another. Both provide asynchronous interaction channels between processes and a central service mediates the creation of a communication channel or queue between processes before they can start exchanging messages. This approach copes with the scalability requirements of the application domains that use this technology, which favor efficient routing algorithms instead of expressive event processing capability.

Examples of queue-based systems include the Microsoft® Message Queuing (MSMQ) (Microsoft 2003), SonicMQ (Sonic 2003) and IBM MQSeries (IBM 2003); as well as the JMS (Java Message Service API) (SUN 2003) specification from SUN Microsystems. In those systems, the communication channel becomes a first-class entity where functionality such as message persistency, transactions, cryptography and secure channels, load balancing, scalability support, XML messages; guaranteed message delivery and others can be provided.

Content-based routing

Recently, a second and more general category of publish/subscribe infrastructures have been developed. They employ an event dispatching mechanism known as content-based routing (or dispatching) (Carzaniga and Wolf 2001). Content-based services allow event consumers to describe advanced queries over the whole content of an event or sets of events. A network of federated servers (or routers) ensure that events published in one end of the network arrive to another end where the subscription was posted. A network of content-based routers with a publisher (client 1) and two subscribers (client 3 and client 2) is presented in Figure 3. For their sophistication and generality these systems usually face a trade-off between expressiveness and efficiency (Carzaniga, Rosenblum et al. 1999). They usually have to process, route and combine events coming from different sources. Examples of such systems include Siena (Carzaniga, Rosenblum et al. 2001), Jedi (Cugola, Nitto et al. 2001), Gryphon(Banavar, Chandra et al. 1999; Banavar, Chandra et al. 1999), Herald (Cabrera, Jones et al. 2001) and Elvin (Fitzpatrick, Mansfield et al. 1999).

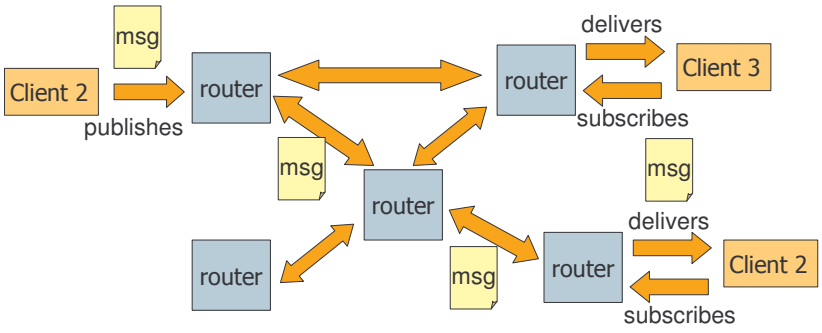

Figure 3 Content-based routing network

5.1.3 Event-based languages and infrastructures

In a way orthogonal to the generalized message routing, subscription and event representation approaches that we just described, different infrastructures and languages build upon the conventional publish/subscribe paradigm in order to provide application specific infrastructures. Those systems include notification servers as CASSIUS (Kantor and Redmiles 2001) and KHRONIKA (Lövstrand 1991), and specialized event processing languages such as GEM (Mansouri-Samani and Sloman 1997) and YEAST (Krishnamurthy and Rosenblum 1995).

Notification servers

Notification servers, as defined by Patterson et al. (Patterson, Day et al. 1996), provide a simple common service for sharing state in synchronous multi-user applications. They have their application in the support for groupware applications in collaborative contexts, and address the problem of maintaining consistency in collaborative applications and supporting awareness. In this sense, they are similar to tuple spaces, with the addition of specialized services for managing the event space and for supporting different notification policies, required to improve all sorts of activity awareness.

Examples of such systems include CASSIUS and KHRONIKA, which handle events as persistent and shared pieces of information about the objects involved in collaborative sessions. In this example, both systems break the publisher/subscriber isolation, allowing the discovery of information providers. They also are able to group events in hierarchies, allowing the classification and storage of events as collaborative information.

Event processing languages

Another interesting group of services that use the publish/subscribe paradigm are event monitoring systems. They provide an event language with programming capability that allows the manipulation of events, combining them in more useful information. An example of such systems is GEM and EDEM (Hilbert and Redmiles 1998), which monitoring capability requires a rich subscription language, not only able to guide the routing of events, but also able to group, abstract and combine those pieces of information in more meaningful data. In those systems, events are usually produced by software applications, and represent runtime execution aspects such as method invocations, GUI events and others.

Another example is YEAST, a general-purpose event-action system that uses the concept of active subscriptions (or active rules). Rules allow the definition of actions in response to subscription matching. Those actions can involve the execution of programs or definition of new events in response to events that can be published back to the system. YEAST in particular is designed to monitor and integrate different objects from a UNIX system (such as files, users, system events) with shell scripting.

Event-driven frameworks

Besides those two common uses of the publish/subscribe paradigm, it has gained attention in other areas such as architectural styles, as the example of C2 (Taylor, Medvidovic et al. 1996) communication bus (or connector), and many other applications that will be presented throughout this survey.

5.1.4 Active database systems

Active database systems support mechanisms that enable them to respond automatically to events that are taking place either inside or outside the database system itself (Paton and Diaz. 1999). In other words, they allow events to be raised by a variety of sources as a consequence of: changes in the database structure or data; on the execution of transactions; based on clock ticks, or even external sources. That behavior is programmed by the use of ECA (Event condition Action) rules which are triggered by the database system. Databases are essentially pull in their interaction model, but the use of rules complement this model with the ability to react in response to changes in data.

Active database systems combine a relational data model with the ability to produce events in response to changes in both data and its representation. This allows the execution of programs that can enforce data integrity, apply transformations, generate notifications, aggregate different content and other possible extensions. The database provides persistency, scalability, interoperability through the use of APIs such as JDBC, and standard query languages such as SQL, which by nature performs content-based filtering.

Those systems are usually enterprise-scale databases, designed to operate over a data centric model, having a big memory footprint and a centralized architecture. The implementation of publish/subscribe infrastructures in this model needs to adhere to the database data and programming models. Protocols need to be implemented separately, as well as the timing and event models. In other words, events must be represented as database tables or objects, and policies such event discarding should be implemented.

The ability to define triggers and associate actions to changes in the database system is a feature common to database systems as Oracle, IBM DB2, Microsoft SQL Server and other enterprise-scale databases. Each of those systems provide a specific programming language, event and data models that allow their implementation of ECA rules and even the execution of external applications as a parameter in the action part of the rule.

Active database systems are interesting in the context of publish/subscribe middleware since they can be programmed to be a persistent core on top of which different notification services are be implemented. In a database system, persistency is a fundamental assumption, and not an optional element. Hence, in the implementation of a publish/subscribe semantic, for example, policies must be defined in order to schedule the discarding of old events, and queries must be limited by timing constraints or event order in order to prevent previous data to be returned. ECA rules can be used to implement different notification policies. SQL queries can combine information from different events into abstracted events, allowing more elaborated subscriptions. SQL queries do not provide support for timing constraints such as those provided by GEM or YEAST, requiring the implementation of such language extensions using the database model.

5.2 Publish/subscribe design dimensions

After this brief introduction, of the existing publish/subscribe models and their main components, we proceed to better understand the main concerns involved in their design. For such, we use a publish/subscribe design framework that captures the main concerns existing in the design

of current infrastructures. This framework is an extension of Rosemblum and Wolf (Rosenblum and Wolf 1997) and Cugola et al (Cugola, Nitto et al. 2001) design frameworks, with the addition of two new dimensions: the protocol and the versatility dimensions, which we further use to survey existing publish/subscribe infrastructures.

In order to understand the concerns involved in the design of a publish/subscribe system, Rosenblum and Wolf (Rosenblum and Wolf 1997) proposed a design framework for such systems. In this framework, the *object model* describes the components that receive notifications (subscribers) and generate events (publishers). The *event model* describes the representation and characteristics of the events; the *notification model* is concerned with the way the events are delivered to the subscribers; the *observation model* describes the mechanisms used to express interest in occurrences of events; the *timing model* is concerned with the casual and temporal relations between the events; the resource model defines where, in the distributed system architecture, the observation and notification computations are located, as well as how they are allocated and accounted; finally, the *naming model* is concerned with the location of objects, events, and subscriptions in the system.

This design framework, however, does not consider additional services, other than the publication and subscription of events present in current publish/subscribe infrastructures, a feature that became very popular recently, with new research in the area of mobility, Internet-scale event notification systems, context-aware applications, peer-to-peer networks, and the wide use of publish/subscribe infrastructures in different application domains. Hence, we introduced a new dimension to this model, the protocol. The protocol model is necessary to capture other forms of interaction with the notification service that goes beyond the common publish/subscribe interaction. This extension to Rosenblum's and Wolf's framework is necessary to express the ability that a notification service has to handle different functionality other than the common publish and subscribe activities, such as: guaranteed delivery, mobility and roaming protocols, security messages, event source discovery primitives and other possible interaction mechanisms with the service and within its distributed components. Also, as proposed by Cugola and colleagues (Cugola, Nitto et al. 2001), we combine the naming and observation models in the subscription model.

Moreover, this framework does not account for the need for versatility, a model that captures the mechanisms and approaches used to configure, extend and program the notification service features do the requirements of different application domains. Hence, this extended model will be used to analyze publish/subscribe infrastructures presented in this survey. A summary of the model is defined in Table 1 as follows. The concept of versatility is further expanded and explained in section 6.

Table 1 Publish/subscribe infrastructures design framework, their dimensions and examples.

	MODEL	DESCRIPTION	EXAMPLE
Interaction	**Event model**	Specifies how events are represented	Tuple-based; Object-based; Record-based, XML files and so on
	Subscription model	Specifies how subscribers express their interest on sub-sets of events and how they are combined and processed in higher-level events (if necessary)	Content-based; Topic-based; Channel-based; Advanced event correlation capabilities
	Notification model	Specifies how notifications are delivered to the subscribers	Push; pull; both, others
Hybrid	**Protocol model**	This model deals with other necessary interaction mechanisms with the system (other than publish/subscribe), to support important requirements such as mobility, security, notification mechanisms and so on. They are subdivided in interaction protocols (that require end-user interaction), and infrastructure protocols (that mediate the communication between software components in the notification service)	**Interaction protocols**: Mobility; Security; Authentication; Advanced notification policies. **Infrastructure protocols**: federation, replication, fault tolerance and so on.
Infrastructure	**Resource model (combining resource and naming)**	defines how the components of the system (publishers/ subscribers and infrastructure are organized and distributed over the network	Centralized; hierarchical (federated); peer-to-peer, configurable
	Timing model	defines different time constraints with respect to the interval or frequency events are produced, and the order they arrive	partial ordering versus total order; real-time constraints, long duration versus instantaneous events, and so on.
crosscutting	**Versatility model**	defines mechanisms and approaches that allow the evolution, extension, configuration and programmability of the publish/subscribe infrastructure.	simple source code modification, plug-in oriented approach, AOP, component-based approaches, mixings, and so on.

Finally, previous classifications did not account for differences between interaction and infrastructure aspects of the publish/subscribe infrastructures. As can be seen from Table 1, the design of a publish/subscribe system involves different concerns ranging from architectural (or infrastructure) concerns such as the resource model, quality of service such as the timing constraints, interoperability and infrastructure concerns such as the protocol model, and also user (in this case the infrastructure programmer) interaction concerns such as subscription, notification and event models. The protocol model has two aspects, the interaction part, and the implementation part, which may or may not come together. For example, in a peer-to-peer implementation, the protocol model may deal with the interaction between notification server peers alone, whereas in a mobility protocol, end-user interaction may be necessary, demanding a roaming end-user protocol. Because of that, we regard as a separate category as in the table above (hybrid). This combined set of characteristics makes the study of versatility in the context of publish/subscribe infra-

structures a challenging endeavor. On considering versatility in this domain, one must match the configuration, variability and evolution of the infrastructure with the evolution and variability of the user interaction models, which comprises the notification, subscription and event models.

In order to better understand our concept of versatility, the next section discusses this aspect in more detail and surveys existing approaches to this problem in the context of publish/subscribe infrastructures.

6 Software Versatility

In this section, we introduce and motivate the concept of versatility for publish/subscribe infrastructures, presenting and explaining the main software engineering requirements involved in this concept. The concept of versatility sets the basic dimensions of the framework that will be used as a guide to evaluate current approaches to each one of those dimensions.

As any other software system, publish/subscribe infrastructures should evolve to accommodate new requirements demanded by the applications it supports. According to Lientz and Swanson (Lientz and Swanson 1980), the software maintenance phase (which includes software adaptation, fault repair and functionality addition and modification) represent 50 percent of the total software cost, having 65% of this cost directly related to the implementation of new requirements. Hence, the importance of using techniques during software design and development, that improves software maintainability and evolution, due to the great impact it has in reducing the total cost of software (Sommerville 2001).

In spite of this fact, current publish/subscribe infrastructures are not designed to cope with software evolution. This is not a surprise. As observed by Parnas (Parnas 1978), the majority of software systems are not designed for change: instead, they are built to solve specific and well defined problems, which ends up hindering their ability to evolve due to its high costs of maintainability. Publish/subscribe infrastructures are not an exception to this observation.

On the light of this problem, Parnas proposes that, in order to support evolution and variability, software must be designed and implemented not as a single program, but as a family of programs that can be extended and contracted according to different application needs. This approach is motivated by his observation that software change is usually driven by the need to support: (1) Extensions motivated by social, organizational or technological evolution; (2) New and different hardware configurations; (3) Differences in its input and output data, while its function is preserved; (4) Different data structures and implementations due to differences in the available resources; (5) Differences in the size of data input and output; (6) And the need of some users of only a subset of features provided by the software.

Hence, according to Parnas, software can be considered **general** if it can be used, without change in a variety of situations; whereas it is considered **flexible** if it is easily changed to be used in a variety of situations (Parnas 1978). Our notion of versatility is based on this original definition of flexibility, and incorporates additional design properties that are important to current publish/subscribe infrastructures. Parnas observations, even though still current and valid did not explicitly mention nor predict other kinds of concerns such as runtime (dynamic) change, module (or component) distribution and usability. The first two issues are central to distributed systems and publish/subscribe middleware, whereas the latter is essential for the acceptability and usefulness of the proposed approaches. Based on this motivation, we proceed to present our concept of versatility.

6.1 Versatility framework for publish/subscribe infrastructures

According to the Cambridge Advanced Learner's Dictionary, versatility is the ability "to change easily from one activity to another" or to be "able to be used for many different purposes." In the context of software engineering, versatility can be defined as the ability of a computational system to serve multiple purposes or to accommodate the requirements of different use situations.

On the light of the above discussion, we proceeded to research ways of providing and maintaining good software engineering qualities that allows the customization, expansion and contraction of publish/subscribe middleware in a usable way. In one sense, the set of properties we survey has been the ultimate goal of software engineering since its beginning: to build software that is easy and cheap to evolve and change. For such, we adopted the term *versatility* in order to embrace that extensive set of qualities. Moreover, we sought a new term that could imply that that these qualities applied not only to technical needs but to the varying needs of human stakeholders and application workplace settings. Hence, from a software engineering perspective, and more specifically in the context of middleware and publish/subscribe architectures, versatility comprises the following requirements.

Techniques for Software Evolution. These techniques allow a piece of software to incorporate changes due to (functional and non-functional) requirements evolution. Techniques in this category accomplish their goals by promoting extensibility, programmability or reuse of software. They are focused in three main sub-areas:

- ***Extensibility (or enhancement) techniques***. Encompasses all classes of enhancements that can be made in the system without changing the existing functionality, for example, techniques as script, macro languages and composition primitives found in UNIX (Notkin and Griswold 1988), or programming language supported capabilities such as OO class extensions. Extensibility is obviously not enough to support software evolution, which usually requires fundamental changes in software functionality. In the context of publish/subscribe middleware, extensibility implies the addition of new functional behavior, such as advanced event processing, or non-functional properties such as reliability and fault tolerance, which adds to the current subscription language, while maintains backward compatibility with existing publish API and the subscription language.

- ***Programmability techniques***. They allow the customization and modification of the behavior of existing software. Programmability (or programming) implies deeper changes in the software, without necessary backward compatibility to existing requirements. For example, in software programmability strategies such as open implementations, strategic pieces of software can be changed or modified providing new behavior to the whole system. This new behavior may not be compatible with previous existing requirements. In the context of publish/subscribe infrastructures programmability allows the reconfiguration of the publish/subscribe middleware to support different event representations (as records, objects, tuples), and to permit deeper changes in the subscription and notification languages, allowing, for example regular expression queries and different event delivery mechanism. Programmability can also be used to define new federation and interaction protocols with the notification service.

- ***Reuse techniques*** (Krueger 1992). Those techniques allow the modularization of certain aspects of software, permitting the incorporation of existing functionality, wrapped as special software pieces (or components), in the construction of new software. These modules permit the transport of functional or non-functional requirements, from one application to another. Reuse allows the built of new systems out of the combination of

new and existing parts, a characteristic that can dramatically reduce the software development costs, since software can be built out of existing parts with minor adaptation effort. In the context of publish/subscribe, for example, existing subscription filtering functions can be used to implement more advanced filtering and event processing commands. A new sequence detection command can use the filtering capabilities already implemented in the system, for example. In another example, existing pull notification mechanisms can be integrated with roaming protocols, implementing more complex mobility functionality.

Techniques for Software Variability (or flexibility). Those techniques are used to manage the contraction and expansion of software in order to support different functional and non-functional requirement sets. For example, to accommodate different hardware resources, application domains, data formats and other reasons mentioned by Parnas (Parnas 1978). They may also be applied to redistribute the processing throughout the distributed system (in our case, between clients and servers). Those techniques may be applied statically, before the software is built and deployed, or dynamically, in the field, after the software is deployed.

- ***Static variability techniques*** are applied at software build time, as the example of conditional compilation, or at design time, such as those applied for the creation of product-line software architectures.
- ***Dynamic variability techniques*** are usually applied at invocation-time, when the deployed software is restarted, or at runtime, while the software is in execution. Examples of such techniques include plug-ins and dynamic architecture approaches described at (Clarke and Coulson 1998)

Usability Techniques for Software Engineering. In order to be useful, and fulfill its purpose, software must be usable by those who will use it. In its definition of usability, Nielsen (Nielsen 1993) proposes a set of characteristics software must have in order to be usable. Those characteristics include: Learnability, efficiency, memorability, few errors and satisfaction. The cost associated to learning, and applying those techniques must not exceed the total cost of developing an application-specific application. Because of that, usability is a key feature in the adoption of the other two sets of techniques.

Besides the above qualities of versatility, publish/subscribe infrastructures need to support the essential middleware requirements of **scalability, interoperability, heterogeneity, network communication and coordination** (Emmerich 2000) which must co-exist with the versatility properties we propose.

After defining our concept of versatility, the next section presents existing software techniques and their applicability to this versatility properties we propose.

7 Survey of existing software versatility approaches

This section surveys existing techniques that have been used in related middleware areas or have a good potential to be used in the publish/subscribe domain in order to provide some or all of the versatility properties we propose. The techniques are presented in different sections, according to the versatility properties that we propose in section 6.1.

As noted by Brooks, programs are complex to design and visualize, while easy to change (Brooks 1987). This flexibility is usually confused with generality, leading the false impression that software is easy to evolve. The reality, however, is that software is usually designed to solve a specific problem and, as such, it passes through many specializations and simplifications during its design. This simplification, while decreases the design complexity, the development time and

the initial software cost, usually promotes a lack of generality, which hinders the software ability to be extended and contracted to accommodate changes. Hence, when changes in the solution domain start requiring modifications in the software, the initial design decisions allied with the inherent complexity of software stand as obstacles for the software evolution. In this context, modifications in software usually results in architecture drift (the destruction of software conceptual integrity as mentioned by Brooks), and the intrinsic complexity of software imposes high maintenance costs. This change and drift cycle usually persists up to the point it becomes economically prohibitive for new changes to be made in the software resulting software. The techniques described here are in their large part, a result of years of research and practice in the fields of software maintenance, design and programming languages. They directly or indirectly represent approaches and techniques that strive to tame the fundamental complexity and changeability characteristics of software.

It is also important to mention that those techniques are not “silver bullets” (using Fred Brooks’ analogy (Brooks 1987)), they are nor universal nor definite solutions to all the software versatility problems as a whole, but represent small steps in the overcoming of some of those problems. Moreover, their use have an associated cost, mainly related to their learning curve, the way those techniques affect the design of software (generalization usually requires a more thorough software analysis phase), and how usable they are when applied in a day-to-day programming discipline. Hence, trade-offs must be observed when deciding which approach to choose, since the versatility gains they provide may come with steep learning curves and lack of usability.

In this section, we introduce a set of approaches that have been used to provide the versatility properties to middleware in general (especially RMI-oriented such as CORBA), as well as other promising software techniques that we think can be applied to publish/subscribe domain. The surveying of those related areas and systems is important since there are very few research systems that strive to address the problem of versatility in the publish/subscribe domain, a fact that limits the number of publish/subscribe infrastructures surveyed, but opens the opportunity to study the application of some approaches in related middleware infrastructures such as CORBA ORBs. Whenever mentioned in the literature, strengths and limitations of each approach in addressing the versatility requirements are discussed.

This list of techniques is by no means exhaustive. They were chosen due to their potential in addressing one or more of the versatility requirements proposed, or for being representatives of promising approaches for the area of middleware versatility, more specifically publish/subscribe. Whenever possible, a list of publish/subscribe infrastructures that uses the proposed approach is presented. It is important to mention, however, that the thorough comparison between those approaches, for the sake of determining which approach is better for publish/subscribe infrastructures is not the goal of this survey. Those techniques are presented here as valid or promising approaches to address the requirements imposed by the versatility properties. Ultimately, it is up to the reader to compare them and decide which technique to use.

7.1 Classification framework

In general, it is hard to separate techniques according to simple criteria as programmability, configurability and reuse. Those characteristics are usually inter-related, being a common goal of many techniques surveyed in this section. For example, the concept of software reuse usually requires ways of efficiently separating concerns into components or reusable blocks. Those blocks require mechanisms to allow their specification, implementation and future composition, which render them as good configurability and reuse techniques. Hence, in this section, we strive to present the software versatility techniques according to a more or less logical order, addressing the most popular ones first, and going to the most promising ones at the end. After a brief description of the approach, we classify them according to the following framework:

- **Short introduction and description**: an introductory set of paragraphs explaining the approach.
- **Strengths**: a list of the main positive points and remarks of the approach
- **Limitations**: a list of negative points or limitations
- **Examples**: a list of publish/subscribe infrastructures or related middleware implementations where the approach has been used
- **Applicability to publish/subscribe**: a brief description on how it can be used to address one or more of the versatility requirements in this domain.

7.2 Object-oriented programming languages

One of the most straightforward and adopted techniques to extend, configure and program new requirements into software is the direct modification of the source code. As a consequence, the most effective way to address the versatility requirements is to empower the programming languages with commands and concepts that allow the taming of software evolution and configurability. Throughout the years, especially in the 80's and early 90's, the object-oriented paradigm and programming languages as C++ and Java became mainstream. Those languages provide native support for the concept of objects (abstract data types), extension, inheritance, generalization, polymorphism, information hiding and late binding of objects.

Applicability to publish/subscribe. In fact, many of the approaches discusses in the next sections are directly or indirectly built upon modern object-oriented concepts, provided by object-oriented programming languages. Those approaches include: software frameworks, software patterns, computational reflection, aspect-oriented programming, and others. Besides the idea of objects as implementation of Abstract Data Types, that encapsulate data and processing below a public API, mechanisms such as late binding, inheritance (extension and generalization) and method overload are responsible by the versatility of OO languages as follows.

- **Late binding of objects**. Allows the decision of which object type (or subtype) to create, to be performed at runtime. It is the basis for different runtime change mechanisms such as template method and factory patterns (Gamma, Helm et al. 1995).
- **Inheritance, (extension and generalization)**. This mechanism allows the separation of concerns that are common to two or more objects, to be componentized in a super class (or parent object). This allows code reuse and facilitates maintenance. It also allows the punctual specialization of objects and the extension of software at object-level.
- **Method overload**. This mechanism also copes with specialization of software, allowing the redefinition of methods in sub-classes, being largely used during extension of software, since it preserves the object contract, in other words its API and usage protocol.

Table 2 OO Programming summary

General description	**Approach/technique**	Object-Oriented (OO) programming
	Pros	Provide a generic programming model based on abstract data types (objects) that cope with reuse, information hiding and extension of software.
	Cons	The information hiding benefit of objects do not scale well, requiring new techniques for their composition. The model does not allow encapsulation of non-functional requirements.
	Examples	OO Programming languages including Java and C++
Versatility	**Extensibility**	Inheritance and method overload
	Programmability	Inherent to the programming language
	Reuse	Concept of objects, inheritance and generalization, associations and aggregations
	Static variability	The concept of interfaces (Java), abstract classes (Java and C++) and inheritance allows objects to be interchanged at program time
	Dynamic variability	Late binding of classes allow the implementation of dynamic variability mechanisms
	Usability	Not enforced by the languages. However, the concept of abstract data types (objects) and their information hiding allows programmers to use objects as black boxes, improving usability of APIs.
Publish/subscribe applicability		Provide the basic mechanisms and concepts used in the implementation of many publish/subscribe infrastructures

7.3 Software Frameworks

A more elaborated form of extensibility is provided by software frameworks. Frameworks separate commonalities from variability in an application domain. They are implemented as skeletal groups of software modules that can be tailored for building domain-specific applications. They provide reuse in the form of pre-programmed logic that can be customized to specific needs in that application domain. Current Software frameworks are usually implemented in object-oriented languages, being described in terms of concrete and abstract classes and a set of variation points or hotspots that together collaborate for the overall software implementation (Johnson and Foote 1988). Users adapt the framework for their need by providing new implementations to its hotspots. Hotspots are parts of a program which are likely to change from application to application. From an OO point of view, hotspots are usually implemented using abstract classes, template methods and interfaces. Frameworks are not limited to object oriented languages, in fact, recent research on Aspect Oriented frameworks is in progress (Constantinides, Bader et al. 2000).

Strengths. As studied by Roberts and Johnson, the use of frameworks can reduce the cost of developing an application by an order of magnitude since it promotes the reuse of both design and code. Moreover, they have been adopted in a large set of applications and, for being built upon existing object-oriented programming languages and techniques, they can rely on existing extensibility and polymorphism features from these languages (Roberts and Johnson 1996). Frameworks also promote reuse. They can be used to consolidate the domain knowledge acquired during earlier projects so it can be reused in future projects to realize the application goal (Codenie, Hondt et al. 1997). Finally, frameworks also hide internal application details, and provide a general domain model, allowing their users to concentrate in customizing the hotspots for their particular needs, instead of being required to understand all the aspects of the program.

Limitations: A disadvantage of the framework is in its high initial development cost, which requires a thorough understanding of the domain being automated and its requirements, knowledge that may take years to be crystallized in the form of a framework. Hence, the design and implementation of software frameworks is not a trivial task, a balance between the number of features provided by the framework and the extension points must be reached. An ideal framework includes all common features of a domain, and leaves all variability to be implemented as extensions. If the framework includes too many features, it can become complex and less flexible; whereas, if it omits common functionality, its generality gets compromised and different applications will need to implement the missing functionality, which may result in code replication (Codenie, Hondt et al. 1997).

Usability may also become an issue. If not well documented, users can start making wrong assumptions about the process the framework automates Fact that can result in wrong implementations and steep leaning curves.

Frameworks are also limited in their ability to evolve in order to address new requirements imposed by the application domain evolution. This evolution usually issues in class complexity and continuous refectory, which issues in documentation inconsistencies, architectural drift, and proliferation of versions (Codenie, Hondt et al. 1997). If a framework gets adopted in the built of many projects, backward compatibility may also impose some restrictions in its evolution.

Example: Recently, frameworks have been used in the development of configurable middleware, as the example of the TAO ORB (Schmidt and Cleeland 2000). The TAO ORB implements a CORBA ORB as an extensible framework. It is modeled in terms of its basic components, allowing the static configuration of services and the runtime change of its strategic components. TAO can be configured to cope with different real-time constraints of applications by selecting the appropriate implementation of each component of the ORB. It also allows the definition of configurations where only necessary components are present, which addresses small footprint requirements of mobile devices or special real-time constraints.

Applicability to publish/subscribe. A publish/subscribe system, as defined in section 5.2 can be designed in terms of different dimensions, which concerns can be generalized, having its main concerns implemented as adaptation points and hotspots. In fact, this idea is used in YANCEES to implement composition filters; and in ADEES to define new subscription commands. In fact, frameworks are largely used to support other approaches such as open implementation and plug-ins, or even aspect-oriented and reflection.

Table 3 Software frameworks summary

General description	**Approach/technique**	Software Frameworks
	Pros	Separates commonalities from variability in an application domain. Reduced end user development costs, reuse of domain knowledge, design and code, high extensibility, allowing its customization to different domain variations.
	Cons	High initial development costs and poor programmability (changes in the main application logic crystallized in the framework are not allowed). High evolution costs: deep changes in the application domain may produce architectural drift.
	Examples	TAO ORB, YANCEES, ADEES
Versatility	**Extensibility**	Provided by hotspots and adaptation points
	Programmability	Limited to the features that can be customized in the adaptation points. The main program logic is not easily programmable
	Reuse	Of code and of domain logic (or solution)
	Static variability	Provided by hotspots and adaptation points
	Dynamic variability	Supported by dynamically loaded adaptation points and hotspots. Generally implemented by the use of OO late binding approach.
	Usability	Information hiding and hotspots allow the extension of software without the full knowledge of the code. Design by contract.
Publish/subscribe applicability		Can be used to model a publish/subscribe system as an extensible core around of new features are implemented.

7.4 Software patterns

Software patterns are recurring sets of relationships between classes, objects, methods and other programming language constructs, that define preferred solutions to common programming problems (Gamma, Helm et al. 1995). Software patters are programming language independent, but became popular in the context of object-oriented programming. They were originally developed by the observation of recurring solutions in existing software frameworks. For such characteristic, software patters have been largely used in software development as elements of reuse and, when combined in the development of new frameworks, can largely improve the extensibility, the understanding and documentation of software, coping with its maintainability. Software Patterns can also be seen as building blocks of large-scale software systems, helping in the discipline and composition of system's subparts. This allows the evolution of software in more predictable ways and leverage the design of software frameworks.

Strengths. The main benefits in the use of software patterns are in the areas of reuse and usability. Software patterns bridge the gap between frameworks and system libraries by providing higher level solutions to common problems. One of the main contributions of design patters is a catalog where researchers and practitioners can refer to common solutions to problems. Moreover, when a solution is non-trivial, patterns work as reference implementations, allowing users to learn form optimized solutions to the problem. In a design pattern catalog, examples, counter-examples and trade-offs are presented, allowing the choice of the pattern to each solution. These high-level concepts also help in source code documentation, improving its understanding and design that, by using this approach, can be expressed in terms of higher-level concepts (instead of mere classes) (Gamma 2001). In a further step, software patters are also small-scale solutions, at the software development level, that are used as building blocks to implement higher-level software components at software architecture level (Beck and Johnson 1994).

Limitations. In spite of all the benefits involved in the use of software patterns, they are not universal solutions. According to Gamma: "Patterns have costs: indirection and complexity, and therefore one should design to be flexible as needed, not flexile as possible" (Gamma 2001). In other words, the excessive use of software patterns, when applied to simple problems, may over-complicate the software, instead of simplifying its comprehension and documentation. A right balance must be achieved.

Examples. Over the last 10 years, software patterns have been applied in the construction of all sorts of software, including virtually all modern publish/subscribe infrastructures such as Siena, YANCEES, FACET and other sorts of middleware such as CORBA ORB frameworks such as TAO (Schmidt and Cleeland 1999).

Publish/subscribe applicability. They have become a common practice in modern object-oriented programming, coping with its benefits to design and maintenance of software. The idea of patterns is not restricted to object oriented programming. They have recently been applied to AOP and other advanced programming techniques too. Patterns that solve common variability, reuse, extensibility and programmability approaches have been defined. Some of them include the strategy design pattern, chain of responsibility, the observer (publish/subscribe), filter, and others (Gamma, Helm et al. 1995). Hence, software patterns and frameworks are largely used techniques that largely improve the maintainability, documentation, reuse and evolution of software. They currently represent building blocks with which other approaches are implemented.

Table 4 Software patterns summary

General description	**Approach/technique**	Software Patterns
	Pros	Brings software reuse in terms of recurrent design solutions, improve extensibility and understanding of software
	Cons	May complicate simple programs with unnecessary generality
	Examples	Most OO-based systems: Siena, YANCEES, FACET, TAO ORB and others
Versatility	**Extensibility**	Some patterns such as the strategy pattern, factories, filters and chains of responsibility represent common implementation-level solutions for extensibility problems.
	Programmability	Some patterns such as the strategy pattern and chains of responsibility can also be used to improve software programmability (for example, open implementations (section 7.8) can be seen as an instance of the strategy software pattern).
	Reuse	Reuse of design and recurring solutions
	Static variability	Some patterns such as the component configurator software pattern, for example, can be used to select between existing implementations.
	Dynamic variability	Some software patterns such abstract factories represent recurring programming-level solutions to the problem of dynamic allocation of components and objects
	Usability	They improve software comprehension, allowing design in terms of higher-level constructs. Pattern catalogs provide common solutions and establish a vocabulary that improves the understanding of software designed with this approach.
Publish/subscribe applicability		Provide the basic mechanisms and concepts used in the implementation of many versatility techniques used for publish/subscribe infrastructures

7.5 Program transformations

As pointed out in 7.3, the adaptation capability of software frameworks is limited. They do not tolerate changes or variability other than that possible to be accomplished when using its variabil-

ity points (hotspots, hooks and adaptation points). As new application requirements get produced, there is a need for evolving the main framework logic and design. Program transformation techniques[1] such as source code refactoring address some of those problems by managing and automating the evolution of the source code.

7.5.1 Source code refactoring

Automated source code refactoring techniques are behavior-preserving program transformations that automate design-level changes (Tokuda and Batory 2001). By applying successive transformations in the program source code (such as add, remove, promote object methods or attributes; inherit, un-inherit, substitute classes; push up and push down methods and so on.), a program can be transformed in order to more easily incorporate new functionality, improve design or be more permissible to changes and extensions. Those transformations must be assisted by software tools, which are usually available as part of Integrated Development Environments (or IDEs). The same techniques can also be used to perform improvement on legacy code by, for example, generalizing it, and making it more amenable to changes.

Strengths. Automated source code refactoring techniques address the complexity involved in modifying existing software such as software frameworks or other complex systems. They manage the consistency of complex software projects updating references, types, names and other program aspects, as necessary, maintaining the original code behavior.

Limitations. Source code refactoring techniques, however, are limited in scale and scope. They provide small-scale, source code level changes (class, method and variable-wide changes), which hinders their scalability to very large projects. When deeper changes are required, such as component-level or system-wide changes, for example, current transformations cannot handle such abstraction level. In other words, they are not fit for changes in higher-level software abstractions. Another problem with source code refactoring is the inability to support behavior changes in the code, such as algorithmic and semantic changes, which requires programmer's assistance, for example, changes in the software environmental assumptions, protocols and others.

Examples. Program transformations provide a set of techniques that support source code changes. As a consequence, they can be used in different programming languages and models in virtually all application domains, including publish/subscribe middleware. An example of IDE that provides refactoring capability is Eclipse (International 2003).

Applicability to publish/subscribe. As other programming techniques, they indirectly allow the development of more versatile publish/subscribe infrastructures by allowing the reorganization of current implementations in order to better accommodate changes and extensions.

[1] More information about other program transformation techniques is available here: http://program-transformation.org.

Table 5 Software refactoring summary

General description	**Approach/technique**	Source code refactoring
	Pros	Help in the evolution of software by automating source code modifications, managing code consistency.
	Cons	Supports source code changes preserving reference consistency. Provides very limited semantic changes such as promote, subclass, combine objects, but lacks more advanced semantic transformations.
	Examples	Object-oriented source code refactoring transformations supported by the Eclipse IDE.
Versatility	**Extensibility**	Not directly supported but provides the ability to perform structural changes in source code that helps in the extension of existing software functionality
	Programmability	Not directly supported but provides structural changes in source code that helps in the addition of new functionality
	Reuse	Operates over existing code, helping in its reuse by allowing its modification and adaptation, which can help existing code to be reused in other contexts.
	Static variability	Helps in directly changing source code, but provides no configuration management capability.
	Dynamic variability	Not supported
	Usability	The use of IDEs is a key characteristic of this technique. This integration improves its usability, resulting in the current popularization of refactoring tools and algorithms.
Publish/subscribe applicability		Is a technique that can be used to support software evolution and maintainability in general, which includes publish/subscribe infrastructures

7.6 Component-based software development approaches

In its seminal paper on software reuse, McIlroy (McIlroy 1968) proposes the componentization of software. Inspired on the componentization existing in hardware, he proposes the creation of a library of reusable software components and automated techniques that would allow their customization to different degrees of precision and robustness, and their subsequent application to build all sorts of software systems. According to McIlroy, component libraries could be effectively used for numerical computation, I/0 conversion, text processing and so on. This seminal idea inspired many of current Component-based Software Engineering (CBSE) approaches.

Currently, the software component concept is built upon concepts such as software modularization and information hiding (Parnas 1972), functional decomposition, abstraction and Abstract Data Types (ADTs) (Guttag 2001), and reuse (Krueger 1992). Components in general are modules as defined by Parnas, that can be decomposed in sub modules, obeying a require/provide relation. They apply the concept of recursive composition: Modules in a lower-level of abstraction provide services to higher-level ones, at the same time that they can require services from other modules. Hence, the concept of software components manage complexity by recursively composing encapsulated pieces of software under well-defined interfaces, obeying common communication protocols or styles. As such, components provide the basis for more advanced strategies such as software architecture and plug-ins that will be further described.

7.6.1 Classical component-based approach – Toolkits and component libraries

The classical idea behind CBSE is the building of software out of a family of generalized components that can be slightly configured and combined in different ways for the implementation of different systems. In order to be applied in an application domain, components are usually de-

signed following pre-defined communication and encapsulation models, generally known as a component models. An example of a component model for the building of applications on top of object-oriented middleware is the CORBA Component Model (Group 2002), which uses RMI over a common CORBA ORB, as its communication mechanism, and IDLs (Interface Definition Languages) as their interfaces. In this model, components can also reside in special containers that provide common services such as transactions, persistency, and life-cycle.

Based on a common component model, components are usually packed in the form of libraries or toolkits. Component libraries provide specialized sets of simple components for the building of software. A classical example is a set of mathematical functions, and generalized algorithms such as those provided in Java packages such as: java.math or java.util. A toolkit is a library of more specialized components, usually tailored as specific purposes that can be used to the build of different applications in a domain. For example, network communication protocols, user interface widgets and so on, as the example of the Java abstract window toolkit (AWT).

Modern component models apply a more elaborate concept of a container. A container is a generalized framework that manages the basic aspects of the component life-cycle such as activation, deactivation, persistency and provides basic communication services. A container can provide more advanced features such as transactions, distribution transparency, load balancing and other policies. An example of a framework that provides this capability is the J2EE[2] and their different implementations such as JBoss [3].

Strengths. The classical CBSE approach achieves high-level degrees of problem decomposition and reuse by applying the concept of components. This approach is usually based on standards, called component models that, by restricting the interaction and encapsulation mechanisms of those modules, strive to improve their ability to be integrated and composed into different kinds of software. A consequence of this model is the improved reuse and configurability of different parts of a software system.

Currently, standardized component models are being supported by application containers, that provide optimized life cycle and communication services, besides non-functional requirements such as security. They provide services such as logging, secure communication channels, persistency, and transactions, many times supported by AOP implementations.

Limitations. The traditional component libraries and toolkits are designed with specific application domains in mind, which usually limits the use of those components in different application domains. Moreover, they are usually provided with limited configurability and adaptability.

The more recent use of application containers may solve this problem by providing all sorts of non-functional requirements to the application. However, they are usually very complex to use and configure and may overcomplicate the implementation when what is needed is a simple or small solution. In other words, the one-size-fits-all approach may not fit all systems afterwards, due to its high footprint.

Example. The Quarterware architecture (Singhai, Sane et al. 1998) defines a middleware construction technique (called software RISC) and a set of generic components (provided as a toolkit), that can be specialized and combined to construct different middleware implementations. Such toolkit includes components for: data marshaling and unmarshaling, object references, data transport, dispatching, invocation policies and wire protocols. Experiments performed using this component library, showed that one can build infrastructures such as CORBA ORBs, Java RMI and MPI (Message Passing Interface) standards. In the Quarterware architecture, the components

[2] Java 2 Entreprise Edition: http://java.sun.com/j2ee/1.4/docs/index.html
[3] JBoss Application Server: http://www.jboss.com/

are implemented as generic OO classes that can be composed associations and can be customized through extension mechanism of the language.

Applicability to publish/subscribe. As exemplified by the Quarterware architecture, that builds a simple MPI application, many parts of a publish/subscribe infrastructures can be modeled using distinct components, and can be customized to different needs using this approach. For example, the subscription-event matching algorithms, the communication protocols, the input and output event queues, or event language commands. The challenge is in modeling and separating the system into replaceable components. Moreover, since versatility is our goal, a good deal of time may be spent on modeling generic interfaces that can absorb the evolution of those components. Non-functional requirements can be provided by application containers.

Table 6 Classical component-based summary

General description	**Approach/technique**	Classical component-based
	Pros	Provides high modularization and reuse of functional components; non-functional requirements are usually provided by standardized application containers.
	Cons	Are usually based on application-specific components; components provide limited internal configurability and programmability. Application containers are very powerful but have a steep learning curve and not all its services are needed by simple applications.
	Examples	The Quarterware architecture
Versatility	**Extensibility**	Extension happens in the component-level, and may be possible by applying OO techniques such as sub classing, generalization, method overriding; ore more advanced approaches such as open implementations and AOP.
	Programmability	Provided by regular OO or AOP languages. Programs must comply with a predefined component model, which may restrict communication styles.
	Reuse	Achieved at the component-level by the use of common interfaces and protocols prescribed by the component model
	Static variability	Provided by the component model and its services. The application containers may provide mechanisms such as manifest files that help in achieving this goal.
	Dynamic variability	Dependent on the component model and auxiliary tools/mechanisms. The application container may provide dynamic loading mechanisms that help in implementing this facility.
	Usability	Information hiding and standards help in creating a consistent programming environment that copes with usability. Application container models may be hard to learn. Advanced features are usually not used due to the steep learning curve of the model.
Publish/subscribe applicability		Main parts of a publish/subscribe system can be componentized; especially event routing mechanisms, protocols and event filters. Application containers can be used to provide non-functional requirements. Language extensibility is not addressed by this model.

7.6.2 Plug-in based software development

Traditionally, plug-ins have been used as optional (as opposed to required) application-specific components which can be used to extend existing applications. In other words, they were defined as small tools or modules, not known at built time, that were used to extend an existing application. Their success in enabling application extensibility has inspired their use as a fundamental mechanism for building whole new applications originating pure plug-in architectures (Birsan 2005). A great success example of this new approach is the Eclipse IDE (International 2003), which are entirely built upon this paradigm.

In this context, plug-in based software development can be seen as a special case of component-based software development which supports the evolution and customization of the features of the application by the use of plug-ins. Plug-ins rely on configuration management provided by the system runtime environment (or kernel), rather than the user, allowing graceful upgrading of systems over time without requiring application restart (Chatley, Eisenbach et al. 2003; Mayer, Melzer et al. 2003). The runtime environment manages: (1) plug-in activation and deactivation; (2) plug-in registry, a list of installed plug-ins; and (3) inter-plug-in dependencies management. Optionally, the kernel can support other services such as logging, security, and so on.

As components, they must also implement a predefined interfaces and communication styles, defined by the plug-in model of its target application. They usually can access a sub-set of environmental resources using an API provided by the application kernel. Plug-ins can be of different granularities. They can be implemented as light-weight modules, or more complex components that can be used to extend software. They can also depend on one another and their interdependencies can be established through extension points defined in the plug-in interface, which allow their composition into complex applications. The dynamism aspect of plug-ins allow their installation after the target application it is released.

Plug-in oriented development versus frameworks. The basic difference between plug-in oriented programming and software frameworks is that: (1) plug-ins are usually much more complex modules than regular framework extension points, which are usually implemented as one or few objects; (2) for being complex, plug-ins can be whole sub-systems, which make them similar to components. (3) As such, plug-ins can depend on one another, which allow their composition to create full-scale applications (for example, the Eclipse IDE is a full application built upon this paradigm).

Strengths. Plug-ins leverage the idea of components with dynamic loading capability, which can be used to reduce application footprint. For being developed for a specific environment, they usually rely on existing environmental APIs and services, which makes their development easier than generalized components. For such characteristics, plug-ins have been used to:

- Modularization and footprint control: To decompose systems in smaller optional parts that are loaded only when necessary, which copes with functional configurability.
- Extensibility: To provide (third party) extensions to existing software after it is deployed.
- Runtime change and upgrade: To allow upgrade of software parts without restarting, which provides dynamic change capability.

Limitations. Plug-ins are usually not designed to provide non-functional requirements to software. In other words, their control over the application they extend is limited by the environment and API the application provides, mainly due to security policies and the lack of access to the application source code. This makes it difficult to use approaches such as AOP, that requires access to the whole application code (break of encapsulation). They also requires extra effort in the original development of the system, which requires the built of an extensible plug-in model, a runtime environment where plug-ins can be activated and deactivated, an API that allows plug-ins to communicate with the application and its main resources and data structures. Another problem that comes from the ability of plug-ins to be composed is the management of their interdependencies, versions and possible incompatibilities (a.k.a. “plug-in hell”). Security management is another issue since plug-ins can many times be dynamic downloaded and installed. Finally, scalability can be an issue, if too many plug-ins are installed at the same time.

Examples. Plug-ins have been widely used on the web to implement extensions both to the server side, as the example of Apache[4], and in the client side, as the example of Netscape Communicator[5]. The Apache web server uses a pluggable architecture where modules providing different functionality can be added. These modules or plug-ins can be installed with the help of hooks and an internal API, over the different stages (request reception, request translation, authentication, resource handling (using MIME types), response generation, logging, response) of the internal dataflow-based architecture of this HTTP server. Examples of extensions supported include protocols such as WEBDAV and SSL, and externally-invoked applications such as CGI scripts.

Recently, the development of software based on plug-ins has gained momentum. One of the drivers of such popularity is the Eclipse environment (International 2003). Eclipse is an IDE (Integrated Development Environment) built entirely on plug-ins. Eclipse plug-ins are not only small programs (or tools) designed to augment the IDE, but also the main building blocks of this tool. The architecture of the system is presented in Figure 4 below. Small tools are usually implemented as a single plug-in, whereas larger tools can comprise many plug-ins. In Eclipse, a plug-in can contribute new functionality to the platform by using extension points declared by other plug-ins. Plug-ins are dynamically loaded when necessary, which reduces the application memory footprint and load time. In a small level, plug-ins are implemented by extending specific interfaces that adhere to a non-preemptive multitasking protocol from the eclipse environment.

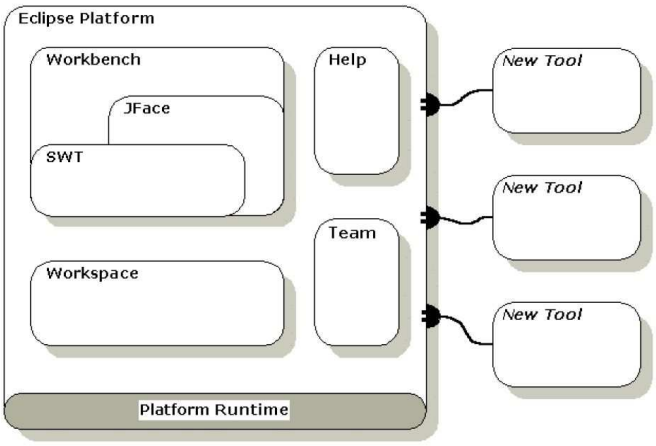

Figure 4 Eclipse Platform Architecture (extracted from (International 2003) Figure 2)

Finally, a publish/subscribe system that uses the concept of plug-ins is YANCEES (Silva-Filho, deSouza et al. 2003; Silva-Filho, Souza et al. 2004). In YANCEES, plug-ins are used to augment the subscription, notification and protocol languages, being dynamically loaded in response to subscriptions that use commands implemented by those plug-ins. YANCEES also allows plug-in composition in the implementation of more complex subscription commands, which improves reuse, copes with footprint control, since plug-ins can be installed or removed as necessary, and provides extensibility.

Applicability to publish/subscribe. As exemplified by the YANCEES notification service, plug-ins can be used to extend the subscription, notification and protocol models, by defining new commands and features in these models. The runtime characteristic of plug-ins allows their load

[4] Apache web server: http://www.ics.uci.edu/~fielding/talks/apache98/
[5] Netscape Gecko Plugin API: http://devedge.netscape.com/library/manuals/2002/plugin/1.0/

when necessary and the evolution of the infrastructure at runtime. It also copes with configurability, allowing the installation and un-installation of plug-ins in order to cope with requirements such as application foot-print.

Table 7 Plug-in based software development summary

General description	**Approach/technique**	Plug-in based software development
	Pros	Plug-ins provide a model for modularization and application footprint control; they implement third-party extensions to software, plug-in runtime provides dynamic load and upgrade capabilities.
	Cons	Are limited in their ability to support the modularization of non-functional requirements.
	Examples	Eclipse and YANCEES
Versatility	**Extensibility**	Provided by extension interfaces and supported by OO programming languages.
	Programmability	Almost any application can be defined with this model, where new plug-ins and their dependencies can be programmed.
	Reuse	Supported in some approaches that allow plug-ins to be dependent on other plug-ins
	Static variability	At load time, when plug-ins are dynamically loaded and composed according to predefined interdependencies
	Dynamic variability	Supported by the intrinsic ability of the model to install (load) or uninstall (unload) plug-ins at runtime
	Usability	The usability of the model is a function of the component model (plug-in model) and the application API the plug-ins can use to extend the application. The ability to expose only the necessary API for the development of plug-ins can improve the learnability of the application, by hiding unnecessary details.
Publish/subscribe applicability		As demonstrated by YANCEES, plug-ins can be used to implement subscription, notification and protocol model extension, configurability and evolution.

7.7 Extensible programming languages

Finally, a recent trend that may improve the programmability of different systems, including publish/subscribe infrastructures is the use of extensible programming languages. According to Gregory Wilson (Wilson 2004), the next generation programming systems will allow their users to define entire new kinds of programming languages and control how they are processed. They will be able to accomplish this by the use of:

- Compilers, linkers, debuggers, and other tools that can be extended by the use of plug-ins;
- Programming languages that allow end users to extend their syntax;
- Programs that are stored as XML documents, that can be processed uniformly.

One of the main examples of how a programmable language can be customized to different application domains is the LISP programming language. It allows users to define their own functions and use them in their programs as first-class entities of the language. Another example is the Java syntactic extender (JSE) (Bachrach and Playford 2001), which allows programmers to define parameterized macros that are parsed into full snippets of Java code. Another example is JSP (Java Server Pages)[6] , which provides a preprocessor that converts XML tags into JavaScript code that gets embedded in web pages implementing their visual dynamism. This language allows new tags to be defined, with their correspondent JavaScript code. Finally, the Jakarta Tool Suite (JST)

[6] Java Server Pages (JSP): http://java.sun.com/products/jsp/

(Batory, Lofaso et al. 1998) provide a set of domain-specific languages and component-based generations based on mixings.

In fact, an emerging approach to versatility, being used in different application domains, is the combination of extensible languages such as XML (the Extensible Markup Language), and plug-ins. This approach is usually motivated by the need to cope with different languages, tailored at different application domains, that share a common infrastructure. Example of systems that use this approach include the Aspect Oriented Markup Language and its aspect plug-ins (Lopes and Ngo 2004), the XADL extensible architecture and its application-specific extensions and tools (Dashofy, Hoek et al. 2005), the xMonVe language for event-based software monitoring (a.k.a as MonArch) (Dias and Richardson 2003) and the YANCEES publish/subscribe infrastructure. All those systems are driven by the need to provide a generic infrastructure that can be customized for specific application domains. Another commonality is the existence of configuration languages, used to install and declare dependencies of plug-ins (or components) used in the infrastructure, and interaction languages (architectures in XADL; monitoring languages in xMonVe; subscription, notification and protocol languages in YANCEES; and domain-specific aspect languages in AOML), that allow end-users to interact with the system.

Strengths. Allows the implementation of domain-specific commands and abstractions into regular or domain-specific programming languages. Besides the benefits inherited from the plug-in based development, this approach allows the combination of the extensibility of a language, which may be provided by extensible languages such as XML.

Limitations. Besides the limitations of the plug-in based development, previously described, this approach requires an understanding of both the extensible language and the plug-in model of the infrastructure. There is also an overhead of the infrastructure that needs to deal with the adequate matching (or parsing) of the extensions in the language with the extensions implemented by the plug-ins.

Another disadvantage of providing language syntax programmability is in the cognitive gap between what programmers write and what they debug. Moreover, as what happens with components, those macros or plug-in components may not encode or perform exactly what is needed by the programmers, forcing their customization of those commands/extensions to particular needs (Wilson 2004).

Example. In the publish/subscribe domain, YANCEES is an example where different subscription, notification and protocol languages need to be supported in order to cope with heterogeneous application domains. For each set of commands in YANCEES, a new plug-in can be defined to implement its functionality. The infrastructure is then responsible for matching the language with installed the plug-ins. In YANCEES, the matching between plug-ins and the extensions in the language is coordinated by parsers, a plug-in registry, and factories.

Applicability to publish/subscribe. This approach is particularly attractive to publish/subscribe infrastructures since it combines the extensibility of languages such as XML with the runtime change and dynamic characteristics of plug-ins, allowing a combined evolution of the subscription, notification and protocol languages with the components that implement those features in the infrastructure.

Table 8 Extensible programming languages summary

General description	**Approach/technique**	Extensible programming languages
	Pros	Allows the implementation of domain-specific commands in different languages. The use of XML and plug-ins work together in providing extensibility and dynamism to the model
	Cons	Need consistency mapping between extensions in the language and their plug-in implementations. Cognitive gap between extensions and actual implementation.
	Examples	YANCEES as the publish/subscribe example, as well as AOML, xMonVe and XADL.
Versatility	**Extensibility**	Provided by a combination of languages and infrastructure implementations, usually provided by XML and plug-ins on both applications and tools
	Programmability	Provided by the plug-in based software engineering.
	Reuse	Reuse of the extensions in the language or the infrastructure is promoted by the plug-in oriented architecture
	Static variability	Supported by the ability to configure the system with plug-ins and extensible languages at load time.
	Dynamic variability	Supported in the plug-in level by the intrinsic ability of the model to install (load) or uninstall (unload) plug-ins at runtime. May be supported in the language level through the use of XML, for example.
	Usability	The usability of the model is a function of the component model (plug-in model) and the application API the plug-ins can use to extend the application.
Publish/subscribe applicability		As demonstrated by YANCEES, plug-ins and extensible languages can be used to promote subscription, notification and protocol model programming and extension.

7.8 Open Implementations

Open Implementation, as proposed by Maeda et al. (Maeda, Lee et al. 1997) goes against the wide spread software engineering principle of “black boxes”, or modules, proposed by Parnas. According to Parnas, modules should expose only the necessary interface to allow its operation, and hide its implementation (the secret) as much as possible from its users (Parnas 1972). This principle, for example, is embedded in the concept of Abstract Data Types, which is one of the fundaments for modern OO languages, and is the basis of CBSE (Component-based Software Engineering). As noted by Garlan et al.(Garlan, Allen et al. 1995), and restated by Kiczalles (Kiczales 1996), the reason black box abstraction does not always work is that the best implementation for a module can only be determined if the developer knows, before hand, how the module will be used. In order words, it is hard to predict all possible uses of a module. Fact that makes modules be designed with special purposes in mind. Hence, according to Kiczalles, if a generic approach is adopted, and modules are implemented in a generic way, they may not fit completely with the software specification this module will be part of.

Open implementation approaches this problem by designing modules that can be adapted or changed to accommodate the requirements imposed by different applications they may be used at. It strives to reach a compromise between the advantages of the “black box” principle and the total access to the module internals. The idea is then to provide modules with alternative implementations that can be tuned according to their use. This is accomplished by a separation of control: in addition to the usual (primary “black box”) interface or API, the component should provide a tuning (select strategy) interface that allows the change of its internal strategies according to the problem to be solved (in other words, the context the component will be used at). Additionally, if the available strategy implementations are not adequate for the problem, an optional interface is

defined that allows the developers to provide their own strategy implementations, thus adapting the component to their needs. A methodology for the analysis and design of systems according to this approach is described here (Maeda, Lee et al. 1997). A range of mechanisms that can be used to implement this strategy is described by Kiczalles at (Kiczales, Lamping et al. 1997).

Strengths. Open implementations cope with configurability and extensibility by allowing the selection and modification of existing implementations to cope with different environmental requirements. They can be seen as small-scale frameworks, in the component-level.

Limitations. Some drawbacks of this approach are the additional costs to design configurable modules (or components), and the fundamental impossibility to design highly customizable components. As happens in larger-scale frameworks, only certain parts of a component can be made configurable. For example, using pure object-oriented programming, non-functional requirements cannot be easily componentized such that those aspects can be added or removed form the system. As will be latter discussed, some of those problems can be addressed by current AOP techniques.

Example. An example of a RMI-oriented middleware that uses open implementation is OpenCorba (Ledoux 1999). It uses computational reflection as a way to instrument the existing implementation and change the internal implementation of the ORB. An example of a publish/subscribe system that uses open implementation is FULCRUM (Boyer and Griswold 2004),which allows the implementation of different context-aware strategies for the commands available in its subscription language, allowing the support for different domain semantics. YANCEES also supports the replacement of certain parts of the system in order to cope with different strategies. In particular, it allows the use of different event dispatchers, permitting the change of the event routing strategy (for example, from content-based to channel-based), by selecting a different core at load time, or by switching from one to another at runtime.

Applicability to publish-subscribe. Systems such as FULCRUM and YANCEES illustrate how open implementation can be useful in the context of publish/subscribe infrastructures. It can be applied not only to the subscription language but also to the notification and protocol models, which intrinsically have the notion of languages or commands, that can be added or removed as necessary, or can be implemented in different ways by the extension of the language. The same approach can be used to support different routing algorithms, as exemplified in YANCEES. When it comes to non-functional requirements as security, reliability and others, this approach is limited.

In the open implementation seminal paper (Maeda, Lee et al. 1997), computational reflection was seen as a preferred mechanism to allow the selection and installation of new strategies in open implementations. They were regarded as good strategies for the implementation of the control interface in such modules. Recently, however, many of the limitations of computational reflection have been addressed by AOP, which improved the design of open implementations.

Table 9 Open implementation summary

General description	**Approach/technique**	Open Implementation
	Pros	Allows the adaptation of software components to different environmental requirements
	Cons	Extra effort to design open components, impossibility to modularize all aspects of a component. No support for non-functional requirements.
	Examples	Open CORBA, FULCRUM and YANCEES
Versatility	**Extensibility**	New implementation strategies can be provided by the users, while preserving the module interface.
	Programmability	Limited since the actual logic of the component is not designed to be changed, only specific points.
	Reuse	The component logic is reused in different contexts, when tuned to different needs
	Static variability	At component load time, when component is tuned.
	Dynamic variability	Provided the component configuration interface that supports dynamism.
	Usability	If well programmed, allow users to customize certain aspects of a component without the need to understand its whole implementation, which improves the learning curve.
Publish/subscribe applicability		In a component-based publish/subscribe system, it can be used to implement different subscription, notification and protocol commands/strategies, or even to allow different routing strategies to co-exist or be replaced as necessary.

7.9 Computational reflection (meta-level programming)

Computational Reflection (or meta-level programming) is a programming technique that allows a program to maintain information about itself (meta-information) and use this information to adapt or change its behavior. In other words, a reflective system is one that is capable of reasoning about itself. This implies that the system has some representation of itself in terms of its runtime programming structures. Reflection also provides access to the basic execution mechanisms of the system through the Meta Object Protocol (or MOP). Using this protocol, meta-programs can intercept and adapt the base-software execution environment, which may include middleware mechanisms such as remote method invocation, marshaling and un-marshaling of messages, thread creation and so on (Costa, Blair et al. 2000).

Reflection is supported by different programming languages in different levels. *Introspection* allows read-only access to the program structure; whereas *structural reflection* enables dynamic alteration of the program structure (for example custom implementation of serialization and de-serialization of programs). Finally, computational reflection allows not only structural but also *runtime control customization* (for example, through the use of smart class loaders). An example of language that allows introspection is the Java language. In this language, commands such as *instanceof* and methods such as *.getClass()* can be used to access the basic building blocks of a software system, its classes. Anotehr language that allows introspection and structural reflection is Guaraná (Oliva and Buzato 1999), permitting the definition of whole meta-level programs. Both can be used in the implementation of middleware.

Strengths. Reflection is a powerful mechanism that allows the fine-grained extension of applications by means of meta-programs, allowing, for example, the implementation of mechanisms to assure real-time constraints, perform logging, enforce security policies, collect performance data, and so on.

Limitations. A drawback of such approach (especially structural reflection), however, is the potential performance and software integrity side effects. In terms of performance, there is a cost associated to the meta-level protocol, and the instrumentation of code to allow the interception of method calls. When used in a generalized way, in many parts of the code, the MOP can slow down the application. Violations of software integrity are another problem (Venkatasubramanian 2002). This can happen when software behavior (or contracts) is accidentally altered by meta-programs or even due to incompatibilities between those meta-programs (composabilty issues). Since the debugging of such applications is harder, those errors may also be hard to track. This is a consequence to the fact that the reflection mechanism itself does not impose restrictions on when and how to extend the system: every point in an application is a potential extension point. Hence, in order to improve its usability, it must be supplemented by architectural restrictions in the system or even software patterns as proposed by (Gutierrez-Nolasco and Venkatasubramanian 2001). Another problem with computational reflection is the need for deep knowledge of the software internals. One must know which points of the software to instrument, according to their function. Finally, in general terms, meta-programs operate at a level of abstraction above programming languages, which often makes it difficult to write correct, easily, readable, maintainable code.

Example. One example of a middleware system that uses reflection is Open ORB (Costa, Blair et al. 2000). Open ORB uses introspection and structural reflection, allowing the ORB to be static and dynamically configurable, as well as programmable and extensible. The basic idea is to provide a bare implementation that can be extended with new features as needed by different applications. This is performed by intercepting and modifying the connections between the main components of the ORB, an approach similar to that used by Aspect-Oriented programming (that will be further discussed).

A publish/subscribe system that uses reflection to be adaptable is ADEES (Vargas-Solar and Collet 2002). It supports introspection by keeping information about the installed components (subscription operations), allowing their composition at runtime in order to perform different sorts of event filtering, which is orchestrated by the Event Manager, a special component of the system.

Another example of use of reflection is YANCEES. In YANCCES, a special component, the architecture manager supports introspection; it keeps a registry of all installed components, allowing the installation of new plug-ins at runtime by the use of the factory design pattern (differently than structural reflection). This information is used to load plug-ins at runtime, by the subscription, notification and protocol parsers.

Applicability to publish-subscribe. Examples such as ADDES and YANCEES shows the ability to use of introspection to help in the dynamic composition of commands in subscriptions. Since many aspects of the strengths of computational reflection are now found in AOP (Aspect Oriented Programming), the use of meta-object programming (especially structural reflection) for middleware extensibility have declined on the last years on behalf of AOP. In fact, more recent systems such as FACET use AOP instead of reflection to achieve the extensibility characteristic that would be otherwise implemented by using structural reflection. More details on the use of AOP will be further discussed.

Table 10 Meta-level programming summary

General description	**Approach/technique**	Computational reflection (or meta-level programming)
	Pros	Allows fine-grained extension of applications, and implementation of cross-cutting concerns such as real-time constraints, logging and security.
	Cons	Performance degradation, potential to break of software integrity, deep knowledge of the application is required which may reduce the maintainability of software
	Examples	Structural reflection: Open ORB; introspection: ADEES and YANCEES
Versatility	**Extensibility**	Extensions to the base-program can be implemented as meta-programs, extending existing software behavior
	Programmability	New behavior can be implemented in the meta-level, using the existing base-level application as a start point
	Reuse	Base code reuse: the same base code can be augmented with different meta-programs. Meta-program reuse: Meta-programs can also be modularized to allow their use in different context, fostering reuse.
	Static variability	Supported by compilers: Meta-programs can be statically combined allowing the addition and removal of functional and non-functional requirements to applications, allowing the implementation of variability policies.
	Dynamic variability	Dynamic meta-object protocols exist that allow the instrumentation of code at runtime by loading/unloading meta-programs and linking them to the base code.
	Usability	One of the problems of this technique is the lack of usability of some implementations. The strategy must be supported by compilers and tools to allow the proper instrumentation and testing of the base code. Meta-level programs can be hard to understand.
Publish/subscribe applicability		Can be used to implement new extensions to existing software by intercepting the communication between modules in a publish/subscribe infrastructure. Introspection is used by applications such as YANCEES and ADEES to load and combine different subscription commands.

7.10 Feature-oriented programming

The concept of feature-oriented programming encompasses techniques that allow the modularization of different non-functional requirements, allowing their selection and composition. Featrure-oriented programming techniques stem from the observation that in the development of applications, separate concerns such as security, logging, persistency and other “ilities” are hard to modularize. They are hard to be implemented as a single module (or component) that can be added or removed from an implementation as necessary. In paradigms such as pure object-oriented programming (OOP), different “cross-cutting” concerns become entangled in code across many modules, which in many cases are forced to implement the same concern over an over. As a consequence, the maintainability of software is highly jeopardized since the addition of a new concern such as security, implies in the update of different parts of the code, residing in many objects throughout the implementation. As a result, whenever new crosscutting concerns need to be implemented in the system, the costs associated to code understandability, and maintainability increase (Lopes 2002).

7.10.1 Aspect-oriented programming (AOP)

AOP complements OOP and leverages concepts such as open implementation and meta-level programming, by providing a programming paradigm that allows the modularization of cross-cutting concerns and the integration of these aspects to the program. One of the main contribu-

tions of AOP was to allow the separation of concerns, which was not possible in traditional Object-Oriented Programming. AOP solves the software entanglement problem by modularizing individual concerns in what was called aspects. Aspects are modular units of cross-cutting concerns which are associated with a set of classes of objects. Aspects can be composed or weaved to the main software on specific joint points. Joint points are well defined points in the software static structure and its dynamic execution control flow. Examples of joint points include method calls (invocations) and field accesses (read/write). Advices are method-like implementations that are executed whenever a joint-point is reached in the program execution. Thus, aspects comprise both the joint points and advices. In other words, aspects are defined in a specific programming language (the aspect language) and are interwoven in an application with the help of special compilers.

For example, AspectJ (Kiczales, Hilsdale et al. 2001) is an aspect weaver (or compiler) that allows the definition and weaving of aspects in Java programs. An example of aspect defined with AspectJ is provided in Figure 5 as follows. In this example, the aspect forces the refresh every time one of the methods in Line or Point objects are invoked. The aspect, which otherwise would be part of the methods in both classes, can be defined separately, and implemented as an advice, improving clarity and reducing code duplication.

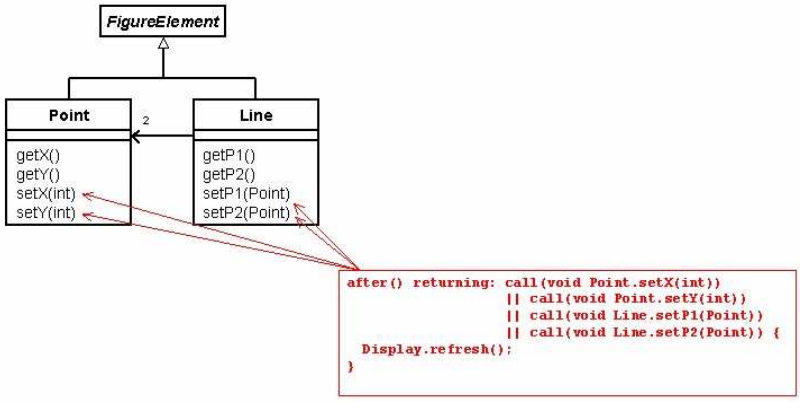

Figure 5 Example of an Aspect defined in AspectJ [7]

With the recent availability of aspect-oriented languages, parsers and weavers such as AspectJ, AOP has become very popular, being used in the implement non-functional requirements and cross-cutting concerns such as: logging, debugging, security, as well as software-wide policies and rules such as architectural constraints, coding conventions and so on. Its potential is still to be fully exploited in developing OO systems.

Strengths. AOP techniques allow the encapsulation (or modularization) of cross-cutting concerns in the form of Aspects. In aspect-oriented languages, aspects are first-class entities that can be weaved in and out a base implementation as necessary. Hence, besides allowing a much better separation of concerns, and the solution of implementation entanglement problem, AOP provides improved modularity and reuse of those non-functional concerns. Configurability also comes as a consequence, as aspects can be weaved to the base code as necessary.

Limitations. As with the reflexive middleware, aspects can be defined to extend virtually part of whole application, allowing the modification of every aspect of the system. Due to this white box approach (more open than that proposed by open implementations), the extension of the sys-

[7] Image source: http://www.theserverside.com/talks/videos/GregorKiczalesText/Figure.jpg

tem requires extra knowledge of its implementation details, which contrasts with object-oriented frameworks, that externalize only specific parts of the software. Moreover, AOP must rely on additional tools and methods to regulate the definition and combination of the different application concerns (or aspects) in a target application (Constantinides, Bader et al. 2000). Due to the generality of aspect languages, and their ability to define pointcuts and advices in virtually every part of the software, architectural constraints and dimensions must be observed in order to restrict and guide their use. For using a fine-grained approach (to the level of specific method invocations), performance degradation is also an issue that must always be managed in such approaches.

Additionally, for allowing point cuts to be inserted virtually at everywhere in the code, AOP should be carefully used. A non adverted use of this technique can break some good software qualities enforced by OOP, such as decoupling, cohesion and textual locality. There is no protection such as available in strongly typed languages and on information hiding, when users are protected from themselves. Hence, its use is recommended for cases where OOP does not produce good modularity.

Examples. In a recent work with RMI-oriented middleware, Zhang and Jacobsen (Zhang and Jacobsen 2004) showed how to extract, model and implement the different functionalities of an ORB as AOP cross-cutting concerns. The middleware was decomposed using in an approach called Horizontal Decomposition (HD). The HD approach proposes the use of conventional OOP software engineering techniques to provide a generalized implementation that provides a minimal core on top of which aspects, implementing domain-specific features, are weaved. His work also proposes some principles to be followed when decomposing such systems in aspects, and how to apply aspects to produce product line families.

FACET (Hunleth and Cytron 2002) is an extensible and configurable and extensible implementation of the CORBA Event Service that was initially designed to allow the customization of this service to address the strict footprint and real time embedded systems applications. FACET also uses horizontal decomposition: a bare-bones implementation of CORBA-ES is augmented with features modeled as aspects. Those features are weaved with one another in order to achieve a customized implementation of the service.

An important problem faced by FACET is the management of conflicting aspects, i.e. aspects that cannot be weaved together, in the final system. In FACET, one aspect can be incompatible with another one, whereas a second aspect can depend on the installation of other ones. This dependency and mutual exclusion problem is addressed by the use of a configuration manager.

Applicability to publish/subscribe. As demonstrated by FACET, and the HD approach, AOP can be used to model and compose different non-functional requirements on top of a basic publish/subscribe infrastructure. The generalized application of AOP, however, including functional requirements may have configuration and performance problems that can be better addressed with other techniques such as composition filters, or even frameworks.

Table 11 Aspect-oriented programming summary

General description	**Approach/technique**	Aspect-Oriented Programming
	Pros	Modularization of cross-cutting concerns and the configurability of those concerns into aspects (pointcuts and advices)
	Cons	May degrade performance and, if not properly used, may damage the software integrity.
	Examples	FACET
Versatility	**Extensibility**	Aspects can be defined to augment the behavior of the base implementation
	Programmability	Aspects can be used to add new behavior to the software, bypassing existing implementation if necessary
	Reuse	Non-functional requirements can be modularized as aspects, an aspect can depend on another one, improving reuse.
	Static variability	Horizontal Decomposition can be used to strip-off or add new features into a base implementation, using aspect weaving mechanism. Thus, one can choose which aspects to add or remove in an implementation, achieving configurability.
	Dynamic variability	Not currently supported buy languages as AspectJ but is being planned for next versions of this tool.
	Usability	The modularization of non-functional requirements can reduce code entanglement improving software understandability.
Publish/subscribe applicability		Can be used to modularize many non-functional aspects of publish/subscribe infrastructures. Also copes with extensibility and configurability of current implementations but does not address the interaction language extensibility (notification, subscription and protocol models).

7.10.2 Composition Filters

Composition filters (or CF in short) (Bergmans and Aksit 2001) is an aspect-oriented programming technique where different aspects are expressed by the use of Filters. Filters implement declarative and orthogonal message transformation specifications that (Aksit and Tekinerdogan 1998). In other words, they extend conventional OO programming model, permitting the association of a special function to one or more method calls in an object. Filters are used to manipulate messages sent and received by an object, specifying conditions where those messages are accepted or rejected. They can also perform user-defined actions before and after a message is received. Since the object observable behavior is determined by the messages it receives and sends, filters associated to those methods are able to express a large set of concerns such as inheritance and delegation, synchronization, real-time constraints, and inter-object protocols.

Filters are programmed using a uniform message manipulation language that can control whether a message is delivered or not to an object. This language operates in terms of runtime conditions or on the messages content. Different collections of filter types can be defined for each application domain, implementing different domain-specific crosscutting concerns. Different filter instances can be associated to more than one object at the same time and can be composed one after another for the expression of filter priorities. Filters provide strong encapsulation and represent modules that can be reused in different applications. New filter types can be implemented, allowing their specialization to different application domains. They can also be automatically generated and instrumented in the classes with the help of compilers such as ComposeJ (Wichman 1999) . Filters can be composed both at runtime or at compile time. This is due to their

declarative structure, uniform language and interaction mechanisms, that make them independent building blocks.

For example, filters can be used to enforce access control policies, permitting certain users to execute some methods in a file class, such as *open()* and *write()*, while forbidding other users from invoking *open()*, *read()* or *write()* methods. As a consequence, the code that would perform the access rights check, and otherwise would be replicated in each method, becomes part of the filter, which is associated to those three different methods and is, by itself, a software module that can be added or removed from the application as necessary.

If compared to AOP, filters provide an alternative way to implement pointcuts, advices, and aspects that do not require any special protocol or modification in the target programming language. Pointcuts are the methods being filtered and the regular expression on what methods to allow and what to block. Advices are the operations that the filter object can before and after a method call. On the same token, if compared to reflection, it provides an OO way to implement the meta-object protocol, that intercept object messages, and the meta-level program, which is embedded in the filter itself.

Strengths. Filters provide well-defined interfaces, and are implemented as orthogonal components, that can be composed in any order. Those two characteristics increase their reuse and adaptability. Filters can be implemented as regular objects in an object-oriented language, which makes this method attractive for the implementation of AOP concerns in OO programming languages. They also have the advantage of supporting runtime change and composition in a more principled way than aspects in AspectJ, since they implement predefined interfaces and can be expressed in terms of OO software patterns such as wrappers, filters and chains of responsibility.

Limitations. It suffers from the same limitations of AOP, but with the potential of imposing more restrictions to its use, for example, filter capability can be allowed only in certain points of the program (by their use in frameworks for example), preventing its indiscriminate use.

Example. The same idea of controlling communication with filters that is applied in the small in the case of individual objects, can be applied in the large, as the case of middleware components. In fact Filman et al. (Filman, Barrett et al. 2001), for example, demonstrated how to extend ORBs with non-functional requirements such as security and fault-tolerance using filters inserted in the CORBA stubs and sleketons. This insertion is performed automatically using a modified IDL compiler. Non-functional requirements are defined using OIF (Object Infrastructure Framework), an extension to IDL that supports the expression of such concerns. With this approach, the use of filters becomes transparent for the end user.

In the context of publish/subscribe infrastructures, filters are used in YANCEES to control the input and output of events in the general publish/subscribe architecture, and in FNF to program to add new services to the event queues. They can be used to enforce type checking, to provide persistency to events, to enforce security policies and so on.

Applicability to publish/subscribe. As demonstrated by YANCEES, FNF and also by OIF, filters are an interesting approach for publish/subscribe system, especially for the implementation of policies, such as type checking, event ordering, persistency, and non-functional requirements such as security.

Table 12 Composition filters summary

General description	**Approach/technique**	Composition filters
	Pros	Provide well-defined interfaces in OO languages. They can be used to implement cross-cutting concerns, and can be statically or dynamically composed.
	Cons	Same of AOP with the advantage of limiting its use to specific points of software.
	Examples	YANCEES, FNF, OIF
Versatility	**Extensibility**	Filters can be used to compose policies and implement aspects, keeping compatibility with the existing system infrastructure.
	Programmability	Filters can also extend the behavior of software by controlling the communication between, its parts, and implementing new functional and non-functional features
	Reuse	The composition characteristic of filters copes with reuse in different contexts as shown by the FNF system.
	Static variability	By adding and removing filters, at object load time, one can configure the behavior of the system.
	Dynamic variability	Supported by OO techniques such as filter design pattern and chain of responsibility.
	Usability	The modularization of non-functional requirements reduces entanglement improving software understandability. The use of software patterns and the native support in OO languages improves its comprehension.
Publish/subscribe applicability		Can be used to modularize many non-functional and functional requirements of publish/subscribe infrastructures. Also copes with extensibility, programmability and both dynamic and static configurability of current implementations. Can be used in conjunction with approaches that allow the extensibility of the subscription, notification, protocol and event languages.

7.10.3 GenVoca (Stepwise refinement)

GenVoca (an integration of two projects Genesis and Avoca) (Batory and O'Malley 1992) is a methodology of program construction inspired in the step-wise refinement methodology (Wirth 1971). It defines components called Layers that encapsulate a complete implementation of a single design feature. A layer is implemented as a set of modularized classes. Those layers can be composed to implement different applications that provide the features expressed in each layer. The JavaLayers compiler, for example, implements the GenVoca model using the concept of mixins (Cardone, Brown et al. 2002). Mixins are mini extensions in the form of types (or classes) which super types are parameterized in a way similar to C++ templates or java and C# generics. With this approach, a mixin can be used to extend any existing class (which is provided as a parameter), with a shared implementation (or characteristic). For example, a Lockable mixins which provides methods *lock()* and *unlock()* could be defined as follows:

```
Class Lockable<T> extends T {
     Public lock() {…}
     Public unlock() {…}
}
```

The class Lockable extends the parameter class T to include the methods *lock()* and *unlock()*. This approach is also known as parametric polymorphism. Using this approach, the code for *lock()* and *unlock()* methods are defined only once, in the mixin. Moreover, it permits existing code to be extended without modifying it. Examples of layers include:

```
class TCP implements TransportIfc {…}
class Secure <T implements TransportIfc> extends T {…}
class KeepAlive<T implements TransportIfc> extends T {…}
```

Hence, mixins introduces super class variability to traditional OO approaches. It also allows features to be mixed and combined with one another, so new types can be defined. They defer the definition of parent/children relation to the composition time, instead of the specification time, as in the traditional OO approaches (Cardone and Lin 2001).

Based on this approach, GenVoca defines the concept of Realms, or modular sets of mixins, representing usually cross-cutting and modular features in a system. These realms can be composed by type equations defined in the JavaLayers language. Hence, in order to compose different layers, JavaLayers provide the following syntax:

```
KeepAlive <Secure<TCP>> trans;
class TP extends KeepAlive<Secure<TCP>> { …}
```

The variable *trans* above is declared as a composition (or generation) of types; whereas the class TP is an instance of this composed type.

AOP and GenVoca, even though differ in their approach and technology, are both designed with the same intent, solve the entanglement problem, and the duplication of code and concerns in OO software. Both allow the easy composition of features in software as a single component. A comparison between GenVoca and AOP is beyond the scope of this paper, and comparison between those approaches can be found here (Cardone 1999).

Strengths. GenVoca allows the encapsulation (or modularization) of cross-cutting concerns in the form of mixins sets, or Realms, allowing their composition in the implementation of software, coping with configurability, separation of concerns and reuse.

Limitations. The use of mixins has some drawbacks. The class hierarchies produced by the use of mixins can generate deep class hierarchies; super class initialization is challenging since the super class of a mixins is not previously known. Since incorrect use of mixings may happen, compositions must be checked for correctness and, since recursive composition of types may also happen, checking can become a challenging task.

Example. GenVoca has been used to re-implement the ACE (Adaptive Communication Environment) ORB, and adaptive ORB that uses software patterns as its functional configuration mechanism. The ORB was decomposed in layers that were afterwards composed to reconstruct a full ORB. The use of mixings reduced the number of lines of code in the software and allowed the definition of different ORB configurations. More details are described here (Cardone and Lin 2001). The results show improved scalability and better support for the evolution of the software if compared to a more traditional framework approach.

Applicability to publish/subscribe. As the case with AOP and composition filters, the approach can be equally used to implement a large part of publish/subscribe system features. However, the extensibility of the subscription, notification and protocol languages is not addressed by this and the other three approaches.

Table 13 Mixins summary

General description	**Approach/technique**	GenVoca mixins
	Pros	Provide an approach for separation of concerns, modularization of non-functional features. Allows composition and reuse based on the concept of mixins and realms
	Cons	Mixins can generate deep class hierarchies, which may jeopardize the understanding of the software. It also requires an extra checking for correctness and compatibility.
	Examples	ACE ORB reimplementation using mixings
Versatility	**Extensibility**	The parameterized sub-typing promoted by mixin allows the extension of software while keeping compatibility with existing classes
	Programmability	Programmability is more challenging in this approach since subclassing implies compatibility with existing classes. Mixins should be used in combination with other approaches to achieve this goal. Cross-cutting concerns, however, may be programmed and waved into the base code using this approach.
	Reuse	Non-functional requirements are easily modularized and can be reused across compatible implementations i.e. Implementations that support the super-classes specified as realms.
	Static variability	Supported by the composition of realms into applications.
	Dynamic variability	Not supported
	Usability	Modularization of non-functional requirements improves software understandability.
Publish/subscribe applicability		Can be used to modularize many non-functional requirements of publish/subscribe infrastructures. Also copes with extensibility and static configurability of current implementations. Does not directly supports extensibility of the subscription, notification and protocol languages

7.10.4 Discussion

A potential problem in most of those approaches is the extreme freedom that they provide to the application developer (especially in AOP and GenVoca). Even though the flexibility of these approaches makes them suitable for a large set of domains, which is indeed a desirable feature, they must be applied according to some principles and restrictions in order to fit each application domain and prevent their misuse. According to John Guttag, *"too muck knowledge is dangerous, and encourages programmers to engage in global modifications and changes that are not local"*. Besides that, this may also issue in loss of performance since the mechanisms used to compose cross-cutting concerns are many times costly in terms of runtime performance.

Additionally, one aspect that is not well addressed by these approaches, in the context of publish/subscribe infrastructures, is the extensibility and configurability of the subscription, notification and protocol languages. All those models focus on the configuration, extension and programming of functional and non-functional requirements at source-code level, and do not handle extensions in the interaction level, which in the publish/subscribe context, is usually implemented by subscription, notification and protocol languages. Hence, other approaches should be used to address the interaction language extensibility.

7.11 Software Usability techniques

Usability as proposed by Nielsen (Nielsen 1993) is not a single attribute but a set of attributes a system must have. Those attributes are:

- **Learnabilty**: a system must be easy to learn so the user can start working with it with few or none previous tutoring;
- **Efficiency**: the system should be fast in accomplishing the work it is supposed to perform;
- **Memorability**: the necessary steps to operate the system should be easy to memorize, so that a casual user is able to return to the system after some period of time, without needing to learn everything back again;
- **Errors**: the system should have a low operational error rate and, if an operational error happens, it should be easy to recover. Catastrophic unrecoverable errors must not occur;
- **Satisfaction**: the user should have a pleasant experience using the system, they should like it.

This definition proposed by Nielsen is focused on GUI-based applications, usually designed to be operated by non-programmers. When it comes to software engineering, however, the concept of usability is slightly different. It not only refers to the software tools used throughout the development of software such as code and document editors, CM systems, CASE tools and so on, but also refers to the artifacts being produced, especially the source code, which middleware systems are part of. In the specific case of middleware, usability must be especially considered with respect to its APIs (Application Programming Interfaces), through which the infrastructure externalizes its services to distributed applications programmers. Hence, for software developers' point of view, usability can be described in terms of internal attributes of a system that affect the developers' performance and productivity in understanding, maintaining and evolving the software.

In the scope of this paper, in terms of middleware infrastructure for publish/subscribe, two main usability issues arise. One is the usability in terms of the system designers which, based on the properties described in the versatility definition section, need to define, configure, maintain and deploy a versatile publish/subscribe middleware. The other aspect is of the systems users, in this case the middleware API clients, that will rely on the services of the versatile middleware to support their application. Since we are dealing with middleware, both classes of users are developers and, at some cases, the same group of people can be at the same time middleware programmers and clients of the system. In this context, some approaches to usability include API evaluation and design techniques following described.

7.11.1 API evaluation techniques

Some usability design and evaluation techniques, focused on libraries, and their APIs, as well as programming languages are defined here (Clarke 2004). According to Steven Clarke, one of the main problems in designing APIs are wrong affordances, or misleading assumptions about the expected outcome of an interface call. For example, a not so usable API would not indicate by its signature (or afford according to psychological jargon), that the passing of a null parameter instead of a file would create a default file instead. A proposed remedy to this problem is user-centered design. In order words, the API must reflect the tasks the users must accomplish and not the implementation details behind its interface. To address those issues Clarke proposes a cognitive framework expressed as a set of good properties an API must have. Those properties are formulated in terms of questions that must be answered during a cognitive walkthrough session involving users and the API. In other words, Steven Clarke proposes a set of 12 measures a good API design that allows the determination of what users expect from the API and what it actually provides. The closer those two factors are, the better the API is in meeting the user's needs.

The results of this approach vary according to the expertise and type of the developer. Another issue is that it may not always be easy to determine the user's needs with respect to the API. In other words, this technique requires a well know user model in order to produce good APIs. One possible solution to this problem is to define different APIs, one for each kind of user (occasional user, more frequent user, expert) and so on, which requires extra implementation work.

7.11.2 API design guidelines and principles

According to Norman (Norman 1988) for the design of usable systems, some principles must be observed such as adequate *visibility*, *conceptual model*, *mapping* and *feedback*. The good visibility principle states that the software must externalize its possible actions, states and alternatives to the accomplishment of the tasks it performs; an adequate conceptual model is a set of abstractions that help the users understand the system and its behavior. It must be complete and consistent with what the system really does. The mapping principle states that actions performed in the system must be followed by a natural (expected) result; and finally, the feedback principle states that the system must provide continuous feedback in response to the actions being performed in the system.

According to Jacques (Jacques 2004) *"A usable API reduces the time it takes to learn how to program against it (learning time), it reduces the number of lines of code to write (execution time), the number of wrong interpretations and misuses (error rate) and allows the easy understanding of code although written a long time ago (knowledge retention). Consequently, the programmer is happy (user satisfaction!) "*. Based on those principles, he proposes some API design guidelines as follows: emphasizes the visibility of important things; minimize the number of visible elements, avoid the use of modes (actions dependent of system states), use of simple functions; employ user domain concepts; consistency; use of the right abstraction level; avoid randomness, use natural mapping, fail as close to the error as possible; provide detailed debug messages; and improve feedback. More details and examples are described here (Jacques 2004). Some other API design suggestions for middleware usability are described here (Bernstein 1996).

7.11.3 Usability aspect of software versatility techniques

The success of many of the software versatility techniques previously described in this survey can be attributed to their ability to achieve a balance between expressiveness and abstraction. In special, they improve the comprehension of the source code, by reducing source code entanglement, summarizing design concepts or encapsulating functionality in standardized ways in order to be reused. In order to be reused, however, software must be well documented and designed, in other words, it must not be useful but usable for its purpose (McLellan, Roesler et al. 1998).

AOP. One of the motivation factors for the development of AOP and AspectJ, was the awkwardness of using meta-object programming to implement aspects. In other words, the lack of usability of such approach. According to Lopes (Lopes 2002), the power of AspectJ as a programming language is in its richer set of structural and temporal referencing that follows that of natural languages. AspectJ allows those constructs to be defined in a way that seems useful to practitioners, allowing the encapsulation of modules in such a way that allows their addition and removal from programs by the use of conditional compilation. According Lopes, *"writing a tracing aspect is like writing a different chapter in a book"*.

Software patterns. Software patterns bridge the gap between frameworks and system libraries by providing higher level solutions to common problems. One of the main contributions of software patters is a catalog where researchers and practitioners can refer to common solutions to problems, such that, when a solution is non-trivial, they learn form optimized solutions to the problem. In a software pattern catalog, examples, counter-examples and trade-offs are presented,

allowing the choice of the pattern to each solution. These high-level concepts also help in source code documentation, improving its understanding and design, that now happens in terms of higher-level concepts (instead of mere classes) (Gamma 2001). The idea of patterns have been applied not only to object-oriented programming but also to AOP and other advanced programming techniques.

Component and Plug-in oriented software development. From the usability perspective, the use of components forces the design for extensibility and programmability, hiding irrelevant application details from the software developer, and decomposing a system into composable modules. The use of components, and especially plug-ins allows the application to be customized and incremented in small and modular steps. In contrast to other kinds of modularization, the development of plug-ins usually produce systems that do not require changes in different parts of the application, which facilitates their development.

7.12 Versatility techniques summary

A summary of the surveyed techniques that highlight the main versatility dimensions that they address is presented in Table 14 as follows.

Table 14 Summary of versatility techniques

Technique	Extensibility	Programmability	Reuse	Static Variability	Dynamic Variability	Usability
OO Programming	Inheritance and method overload	Object-based programming	Inheritance, associations and aggregations	Interfaces, abstract classes and specialization	Late binding	Abstract data types
Software Frameworks	Hotspots and adaptation points	Hotspots and adaptation points	Of the framework logic	Hotspots and adaptation points	Dynamic hotspots and adaptation points	Information hiding
Software Patterns	Extensibility patterns	Programmability patterns	Of design solutions	Software configuration patterns	Dynamic variability patterns	Catalog of common solutions
Software Refactoring	Not directly supported	Not directly supported	Of old source code	Direct source code modification	Not supported	Provided by automated tools
CBSE	Supported by OO techniques	Supported by OO techniques	Of components	By component composition and linking	By component containers that provide this feature	Information hiding, component model and containers
Plug-in based software development	OO programming languages and extension interfaces	Supported by OO techniques	Of plug-ins	At load-time plug-in composition (interdependencies)	By the plug-in runtime that allows dynamic loading/ unloading and upgrade	Modularization, reuse and information hiding
Extensible programming languages	Provided by plug-ins and extensible languages such as XML	Provided by plug-ins	Of plug-ins and language extensions	Provided by the plug-in model	Provided by the plug-in model	Function of the abstraction, composition and extensibility of language

Open implementation	Of components by allowing new strategies	Limited since the module interface is preserved	Of components main logic and interface	At load time when component is tuned/ configured.	Provided by the configuration interface	Partial information hiding
Meta-level programming	Of individual or group of objects, by the meta-program	Of cross-cutting concerns, by the meta-program	Of base code and meta programs	Compilers can link together different meta-programs to the base code	Meta-object protocols allow load/unload and dynamic link of meta-programs	Low usability: meta-level programs can are hard to understand.
AOP	Of objects and programs with aspects	Of cross-cutting concerns with aspects	Of non-functional requirements modularized as aspects	By the use of aspect composition techniques	Not currently supported by AspectJ but planned for next versions.	Reduces software entanglement and modularizes non-functional requirements
Composition Filters	Of objects by intercepting method calls using OO programming	Of cross-cutting concerns by intercepting communication using OO.	Of filters and of cross-cutting concerns	By adding or removing filters at load time for each object/ method	Supported by the filter, chain of responsibility and other software patterns	Natively supported by OO languages
GenVoca	Of collections of objects based on mixins and OO programming	Of non-functional concerns using OO programming with mixins and realms	Of non-functional requirements that can be easily modularized into mixins and realms	Of non-functional concerns using mixings and realms	Not supported	Modularization of non-functional requirements.

Once identified the main techniques that can be applied to provide versatility, a next step would be to compare them with one another. The comparison between those techniques, however, requires a detailed comparative study of their use in the publish/subscribe domain, which is beyond the scope of this paper. On the absence, in some cases, of systems that apply those techniques to publish/subscribe, we can only present them here as alternative approaches and exemplify them, as we did, with their use in related middleware fields, such as RMI-oriented middleware.

8 Other versatility approaches

This section present alternative approaches to the versatility problem, especially addressing issues related to configurability and variability. They take a more holistic (application-wide) view instead of just focusing on the infrastructure alone.

8.1 Model-driven approaches

Model driven approach strives to separate program specification from the technology that implements it. This is achieved by using refinements and mappings to transform specifications in actual implementations. In this context, middleware becomes a component in the overall system, that must be configured to attend the needs of the application.

OMG promotes the use of Model Driven Architectures (MDA) as a way to decouple the application specification from its particular implementation on different middleware platforms. The approach maps platform independent models defined in UML to middleware-specific implemen-

tations. The idea is to better isolate the application specification from the specifics of different middleware, improving portability. The mapping from independent specification to middleware is automated, and performed with the help of platform-specific models.

A similar idea is proposed by Ckarnecki et al. (Czarnecki and Eisenecker 1999). He defines Generative Programming as an approach for generating customized components and systems. It combines techniques such as AOP, Domain Specific Languages (DSLs), Generic Programming and Configuration knowledge to achieve separation of concerns (AOP). The idea is the creation of generic source code libraries that can be automatically customized and composed in order to implement domain-specific systems. Domain Specific Languages are used to program the application in a domain-specific language, using domain constructs (for example mathematical programming); Generic Programming allows the definition of parameterized data structures by the use of templates and generics (for example STL – standard template library -- from C++); and AOP to allow the implementation of cross-cutting concerns. The whole idea is to automate the code generation from the DSL to the final program using the generalized implementations that, with the help of domain knowledge, are automatically configured and assembled together in order to build the solution to that particular problem.

An example of a publish/subscribe service configuration model, used to support the communication of CORBA components is descried in (Edwards, Deng et al.). It defines a model-driven approach to deploy event-driven applications, and configure the publish/subscribe service, with respect to distribution and QoS, to support those applications. The whole idea is to automate the configuration of the publish/subscribe service in terms of how the channels are federated, and which parameters are selected (push versus pull model, persistent channels, and other CORBA-ES options) on those services.

8.2 Service-oriented architectures

Service-oriented architectures (SOA) are used to implement complex applications by the integration of distributed services, as the case with web services. A service is an application externalized through standardized programmatic interfaces, a façade in the software patterns jargon, or a component in a software architecture point of view. Services hide the implementation of more complex systems behind well defined interfaces, which should be operated according to a richer semantic protocol. Hence, service-oriented approaches are not middleware extensibility approaches, but a composition and integration strategy that combines distributed applications. For example, web services[8] externalize APIs for different e-businesses and applications through the use of standardized and HTTP-based protocols such as XML-RPC, and SOAP, published in an standardized way using the web service description language WSDL. The interconnection of services is helped by the use of other services such as UDDI

9 Survey of existing publish/subscribe infrastructures

After presenting the concept of versatility and surveying existing software approaches that help in addressing those properties, we proceed to present existing publish/subscribe infrastructures, presenting them according to the generalized design framework from section 5.2. In do doing, we expect to make explicit the dimensions those systems address and the variability they require in each one of those dimensions. The goal is not only to classify existing systems according to this new framework, but also to illustrate the diverse set of requirements publish/subscribe infrastructures need to support due to their use in different application domains.

[8] http://www.w3.org/2002/ws/

In the next section, we present a table relating all those systems and how they can be used to address the proposed software versatility properties.

9.1 CORBA-NS

The CORBA Notification Service (CORBA-NS in short) (OMG 2002) is an extension to the CORBA Event Service (CORBA-ES in short) (OMG 2001) that allows the definition and management of different event channels between CORBA distributed objects. It supports topic and channel-based routing, as well as content-based filtering of events. Events can be typed or untyped, persistent or non-persistent. Subscriptions allow sequence detection and content-based filtering. The event delivery can be performed using pull and push approaches. Secure channels can be established between publishers and subscribers. Scalability is addressed using federation of servers. The CORBA-NS provides a very comprehensive set of features since it is designed to support the largest set of applications as possible. A summary of its features are described in the following table.

Table 15 Design dimensions for the CORBA Notification Service

	CORBA Notification Service
Event Model	Record-based
Subscription Model	Topic-based, channel- based, and content-based filtering
Notification Model	Push and pull
Timing Model	Total order when using channel and topic-based subscriptions.
Resource Model	Centralized on the server(s). All event processing and filtering is performed in the server-side
Protocol Model	**Interaction protocol**: besides the conventional publish/subscribe API, it provides secure connections, and polling protocols to retrieve persistent events. **Infrastructure protocols**: server federation and fault tolerance mechanisms are used to connect distributed servers, a time synchronization protocol can also be provided.
Versatility Model	The CORBA-NS standard does not specify versatility mechanisms. Some implementations discussed in section 7, propose different versatility approaches to CORBA ORBs

9.2 Java Message Service (JMS)

The JMS standard from Sun (Sun Microsystems 2003) is based on the OMG CORBA-ES standard, being especially designed for Java and EJB (Enterprise Java Beans) integration and communication. It supports topic and channel-based event subscription models; events are represented as records with predefined set of attributes. Hence, event routing is performed through distributed queues between event producers and their consumers. Some implementations of this standard support event persistency and transactions as additional properties of the channels. Different notification policies such as pull and push are also supported.

Table 16 Design dimensions for the Java Message Service

	Java Message Service
Event Model	Record-based
Subscription Model	Topic- and channel-based
Notification Model	Push and pull
Timing Model	Total order of events between producers and consumers guaranteed by the event queue
Resource Model	All processing is performed in the server side
Protocol Model	**Interaction protocols**: besides the regular publish/subscribe, it allows transactions to be defined **Infrastructure protocols**: not specified
Versatility Model	OO Programming – direct source code modification, or application level implementation of extensions

9.3 READY

The READY (Reliable Available Distributed Yeast) notification server is a general-purpose service based on YEAST event action system (Krishnamurthy and Rosenblum 1995). READY incorporates most of the functionality of Yeast (further described), with the ability to handle compound event matching (and other event constructs), subscriptions that matching over both single and compound event patterns; communication sections that manage quality of service (QoS) and event ordering; as well as group subscriptions. Event abstraction – the creation of events based on the combination of attributes of other events is also provided. The system also supports the temporary disconnection of sections, coping with mobile applications and fault tolerance. READY supports event types and subtypes, allowing users to specify their own hierarchy of events.

Table 17 Design dimensions for the READY Notification Service

	READY
Event Model	Recod-based with support for event typing and hierarchies
Subscription Model	Content, topic and channel-based subscription models are supported as well as advanced event processing features (sequence, abstraction)
Timing model	Total order of events
Notification Model	Push and pull
Resource Model	Provides both: server-side and client-side subscription evaluation
Protocol Model	**Interaction protocols**: Mobility support (disconnection and reconnection primitives), authentication and polling protocols **Infrastructure protocols**: Server federation and migration of event processing from server to client side
Versatility Model	OO Programming – direct source code modification only

9.4 Siena

The Siena content-based router (Carzaniga, Rosenblum et al. 2001) provides a Internet-scale event notification network implemented as a federation of servers; The subscription model provides content-based filtering, and event sequence detection. The event model is tuple-based and

the notification model implements a best effort event delivery, implying in no event delivery guarantee. Siena applies advanced subscription advertisement and event routing algorithms to allow events published in one side of the network to be routed to interested parties that post subscriptions in different nodes (servers) of the network. Current version guarantees partial event ordering.

Table 18 Design dimensions for the Siena Notification Service

	Siena
Event Model	Tuple-based
Subscription Model	Content-based
Timing model	New implementations support partial order of events when federated configurations are used, older implementations apply best-effort approaches
Notification Model	Push
Resource Model	Server is federated and all processing is performed in the server side
Protocol Model	**Interaction protocols**: Supports only the common publish/subscribe interaction **Infrastructure protocols**: Server federation and advanced routing protocols for servers interconnection, as well as partial event ordering
Versatility Model	OO Programming: open-source distribution, allowing modification of the source code; Provides a minimal publish/subscribe core that can be used by the client application to implement more advanced features such as event abstraction ex: used by YANCEES and FULCRUM.

9.5 Herald

The Herald project from Microsoft (Cabrera, Jones et al. 2001) implements a distributed event routing network that provides Internet-scale content-based routing. The scalability is achieved by the federation of servers. Resilience to failure, self configuration and administration, timeliness (human acceptable delays); support for disconnection of publishers and subscribers are provided; security (access control and authentication) and partition communication are also supported. These characteristics are mainly accomplished by a design on loosely-coupled components called rendezvous points which are federated. Each component is designed to handle failure conditions and to rely as less as possible on others. Replication of rendezvous points cope with scalability (uses load balancing) and fault-tolerance (uses store and forward of events when connection is reestablished). Herald does not support filters or advanced query languages. It also does not guarantee event ordering as Siena.

Table 19 Design dimensions for Herald

	Herald
Event Model	Tuple-based
Subscription Model	Content-based
Timing Model	Best effort, no guaranteed event ordering.
Notification Model	Push
Resource Model	Servers are federated and all processing is performed in the server side
Protocol Model	**Interaction protocols**: Supports only the common publish/subscribe interaction **Infrastructure protocols**: Transparently provides fault tolerance, replication and composition between rendezvous points
Versatility Model	API provides a simple pub/sub core, and closed source code (commercial distribution) requiring implementation of new features in the client application.

9.6 Elvin

The Elvin notification server (Fitzpatrick, Mansfield et al. 1999) was initially designed as an event routing infrastructure to support the development of awareness applications. The use of content-based routing and a subscription mechanism with quenching (which optimizes subscriptions and discards published events that are not of subscriber‘s interest), together with federation mechanisms, enabled its use in large-scale applications. Elvin provides a relatively simple but optimized set of functionalities, with the ability to efficiently process a large amount of events based on content-based routing, of tuple-based events. The subscription model does not support simple event pattern detection as Siena, but allows filtering based on regular expressions in the content of the events.

Table 20 Design dimensions for Elvin

	Elvin
Event Model	Tuple-based
Subscription Model	Content-based with support for content-based regular expressions filtering
Timing model	Total order of events guaranteed in the centralized implementation only
Notification Model	Push
Resource Model	Servers support federation; all processing is performed in the server side
Protocol Model	**Interaction protocol**: Pure publish/subscribe interaction with the end user **Infrastructure protocol**: federation of servers
Versatility Model	Simple core, and closed source code (commercial distribution) requiring implementation of new features in the client application.

9.7 Gryphon

The Gryphon (Jin and Storm 2003) publish/subscribe system strives to combine the strengths of database systems with the timely delivery of notifications from publish/subscribe infrastructures. It allows the use of SQL relational queries (continuous queries over event streams) performed over a distributed event routing network, which allows, for example, the realization of joins involving events from many different sources that can span a given period of time. A key feature of this system is the ability to perform queries on histories of events, in what is called “stateful” middleware. The use of SQL allows richer queries, which can combine data from different sources at different times, calculating totals, summarizing content, and group information into new data. The relational ability provided by Gryphon is applicable to many application domains, in special, it is important for context-aware applications, mobility and other applications where history information is important.

Table 21 Design dimensions for Gryphon

	Gryphon
Event Model	Tuple-based
Subscription Model	Content-based through a relational query model
Timing Model	Total order of events with synchronized clocks
Notification Model	Periodic pull (queries are checked at every time interval)
Resource Model	Servers are federated and all processing is performed in the server side
Protocol Model	**Interaction protocols**: Supports only the common publish/subscribe interaction, but with a relational (SQL-based) subscription language **Infrastructure protocols**: federation, clock synchronization, guaranteed delivery of events
Versatility Model	SQL query capability provides query flexibility. Infrastructure provides an API to the client. Source code is not available, requiring implementation of new features in the client application.

9.8 JEDI

The JEDI (Java Event-Based Distributed Infrastructure) (Cugola, Nitto et al. 2001) was designed to cope with the special requirements of scalability and mobility. Scalability is achieved by server federation. Event ordering is guaranteed by the system. One distinct characteristic of Jedi is the support for mobile applications. As such, roaming and special primitives (move-in and move-out) for client migration are provided. Event sequence detection with regular expression is also supported. To cope with mobility, push and pull delivery policies are supported. JEDI uses subject-based filtering where events are represented as method invocations: each event is labeled with a subject, the method (or function name), and a list of attributes representing the parameters.

Table 22 Design dimensions for JEDI

	JEDI
Event Model	Record-based. Events represent method invocations.
Subscription Model	Subject-based
Timing model	Use of logic clocks to achieve partial event ordering
Notification Model	Push and pull to support mobility
Resource Model	Servers are federated and all processing is performed in the server side
Protocol Model	**Interaction protocols**: Provides mobile applications support, with the help of protocol primitives such as move-in and move-out. **Infrastructure protocols**: federation and roaming protocols
Versatility Model	OO programming: Source code modification or implementation of advanced features by the client applications only

9.9 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 ability to model information source hierarchies, allowing end-users to browse through and subscribe to those hierarchies. The level of disruptiveness of the notifications can also be configured according to dif-

ferent awareness styles. It allows the definition of content-based or type-based subscriptions. Support for mobile applications is made possible through the use of the ubiquitous HTTP protocol and by allowing information consumers to store events in the server during periods of disconnection. Cassius uses a record-based event model, which its own set of fields.

Table 23 Design dimensions for CASSIUS

	CASSIUS
Event Model	Record-based
Subscription Model	Topic-based and Type-based
Timing Model	Total order of events guaranteed by the central server
Notification Model	Push, pull and periodic pull
Resource Model	The server is centralized. The client-side provides support for pull delivery and sequence detection
Protocol Model	**Interaction protocols**: Provides an API for managing event sources and hierarchies, and for reading persistent notifications. Also supports user authentication protocol. **Infrastructure protocols**: supports HTTP and a special pull protocol
Versatility Model	OO programming: New features require direct source code modification or application-side implementation of the required functionality. Fixed record-based set makes it difficult to use the event format in different application domains.

9.10 KHRONIKA

In the same way as CASSIUS, KHRONIKA (Lövstrand 1991) is a notification server designed to support awareness in collaborative settings. It allows the filtering and delivery of information coming from different event sources. Each user of the system can specify sets of pattern-action subscriptions (in the form of Event-Condition-Action – or ECA rules). The notification mechanism is configurable allowing different delivery mechanisms such as sounds, messages, starting of applications and so on. Khronika also allows the direct browsing of the event repository. Events have expiration times and remain on the server database as specified in their validity (days, hours or brief intervals). They are represented as attribute/value pairs and can be grouped in class hierarchies. The event language allows queries by time interval, event types and substring matching. Access control lists and user groups are used. These restrictions are however, made simple for usability purposes.

Table 24 Design dimensions for KHRONIKA

	KHRONIKA
Event Model	Tuple-based
Subscription Model	Content-based supporting ECA rules (active subscriptions).
Timing Model	Guaranteed event ordering due to central server. Allows event expiration check
Notification Model	Push and Pull with programmable notification styles
Resource Model	Centralized server with distributed daemons: all event filtering is performed in the server side but daemons execute notification actions
Protocol Model	**Interaction protocols**: User authentication, Hierarchy browsing **Infrastructure protocols**: unknown
Versatility Model	Programmable notification language: Parts of the server can be programmed with Lisp, allowing new notification styles to be provided.

9.11 GEM

The GEM (Generalized Event Monitoring) system (Mansouri-Samani and Sloman 1997) implements a generalized event language for real-time distributed systems monitoring. It provides an advanced language where ECA (Event Condition Action) rules can be defined. Advanced sequence detection and specification rules can be specified, executing actions such as: the activation or deactivated of other rules, the generation of higher-level events, summarization of events, detection of critical conditions and so on. For being designed for real-time monitoring, rules can include special time constraints such as specific delays between events and timers. It also allows the use of event order constraints in event expressions, such as the specific order events should occur and the acceptable delay between them. Events can be abstracted and generated based on contents of other events. The event model is record based: events are represented as records with variable attributes list, following the structure*: (event-id, [<source-id>], [<timestamp>], [(<attribute-value-list>)]).*

Table 25 Design dimensions for GEM

	GEM
Event Model	Record-based
Subscription Model	Topic-based: evens, are generally queried by their id (a topic in this case) or timestamps, when used in temporal expressions. The ability to mix temporal relations between events with complex sequence detection expressions are the most important constructs of the language. It also supports rules and actions.
Timing model	Total order is of events is guaranteed by a centralized implementation
Notification Model	Push, events are processed as they arrive and new events can be generated as notifications.
Resource Model	Centralized on the monitor that interprets the language
Protocol Model	**Interaction protocols**: GEM language **Infrastructure protocols**: unknown
Versatility Model	The interaction language, with ECA rules (active subscriptions), allows the programming of user-defined event processing rules, but is limited by the language vocabulary. The service itself can only be changed using OO programming techniques.

9.12 YEAST

The Yeast (Yet another Event-Action Specification Tool) (Krishnamurthy and Rosenblum 1995) is an event-action system used to automate tasks in a UNIX environment. Yeast allows actions to be performed when event patterns and environment changes are detected. It allows the association of temporal constraints to events, borrowing its syntax from the **at** and **cron** programs of UNIX systems. Sequential and out or order event pattern detection is supported. User-defined actions are executed whenever an event pattern match occurs. These actions can originate new events or start different applications. Yeast also allows the definition, activation and deactivation of rules at runtime. This flexibility is provided by a shell script interface that integrates the Yeast event processor with the UNIX environments. In short, YEAST works as an event-driven language, with some similarities to the UNIX **cron** rule interpreter.

Table 26 Design dimensions for YEAST

	YEAST
Event Model	Record-based
Subscription Model	Topic and type-based with rule-based expressions (ECA rules). Supports temporal and non-temporal event subscriptions (or specifications). Allows the definition of complex event patterns.
Timing Model	Event ordering is enforced by the infrastructure
Notification Model	Pull: users can define rules to query for the status of the subscriptions (rules). Push: actions can be associated to subscriptions, allowing the execution of applications or the generation of new events, which allows different notification strategies.
Resource Model	Centralized, all processing is performed by the local YEAST daemon.
Protocol Model	**Interaction protocols**: User authentication, rule status query **Infrastructure protocols**: unknown
Versatility Model	Event-driven language that allows the elaboration of advanced subscriptions with the existing vocabulary and the implementation of different notification mechanisms due to its integration with UNIX shell script and ability to invoke external applications. Further extensions need source code change (possibly C programming).

Other examples of systems that use rule-based subscriptions and specialized subscription languages are RUBCES and RUBDES (Sahingöz and Erdogan 2003) (Sahingöz and Erdogan 2003).

9.13 TSpaces from IBM

TSpaces (Wyckoff 1998) is a middleware for ubiquitous computing based on the tuple space model and principles established by the Linda system (Gelernter 1985). It provides group communication services, database services, URL-based file transfer services, and event notification services. TSpaces allows heterogeneous, Java-enabled devices to exchange data with little programming effort. In TSpaces, a tuple is a set of fields (attribute name, type, value) that represent sets of Java objects. Tuples are published in tuple spaces in specific TSpace servers. Information consumers subscribe to tuples using templates.

The basic primitive operations supported by the space are:

- *write(tuple)* Adds a tuple to the space, equivalent to a publish command.
- *take(template_tuple)* 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.
- *waitToTake(template_tuple)* 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.
- *read(template_tuple)* Same as the "take" command above, except that the tuple is not removed from the tuple space.
- *waitToRead(template_tuple)* Same as the "waitToTake" command above, except that the tuple is not removed from the tuple space.
- scan(template_tuple) Same as the "read" command above, except returns the entire set of tuples that match.

- *countN(template_tuple)* Same as the "scan" command above, except that it returns a count of matching tuples rather than the set of tuples itself.

Besides the primitives described above, the TSpaces API have evolved to support more advanced inter-process synchronization methods supporting regular expressions string matching, transactions, SSL communication, XML and others[9].

Table 27 Design dimensions for TSpaces

	TSpaces
Event Model	Tuple-based
Subscription Model	Type- (or topic-) based
Timing model	Central tuple space serializes operations and guarantees ordering of events
Notification Model	Pull, when take() command is executed
Resource Model	Event matching is performed in the server-side
Protocol Model	**Interaction protocols**: TSpace API as presented above **Infrastructure protocols**: Consistency algorithms to cope with multiple federated servers
Versatility Model	Minimal extensions can be programmed using the server API. For example, a distributed queue, by the use of write and take over a tuple type. OO programming: Other changes require direct source code change.

9.14 The Modular Event System

The Modular Event System (Fiege, Mühl et al. 2002) provides a configurable publish/subscribe architecture based on the formal concept of *scopes* and *event mappings*. Informally speaking, scopes are software components that implement a standard publish/subscribe interface. They communicate by publishing and consuming events as any other publish/subscribe system. Scopes can be recursively composed in publish/subscribe trees where a super-scope subscribes to events from sub-scopes. Using this approach, a standard publish/subscribe system can be assembled. The final behavior of the system is defined by the recursive composition of scopes, each one implementing a different concern. For each scope, *event mappings* can be defined. Those mappings apply sets of transformation to events allowing, for example, the implementation of content transformations (for interoperability between two event services for example), or the filtering of events based on security policies, or visibility rules, as another example. Hence, this approach provides a modular architecture for the implementation of different publish/subscribe infrastructures based on a common interface.

The current implementation, however, is based on the Siena event and subscription models, allowing simple content transformations in the event mappings. In other words, it focuses on event transformations for interoperability. Scopes allow the configuration of the functionality to include or exclude from the service, coping with static configuration. The idea, however, can be further extended to address other versatility issues.

[9] Source: http://www.almaden.ibm.com/cs/TSpaces/

Table 28 Design dimensions for the Modular Event System

	Modular Event System
Event Model	Attribute/value pairs, but supports transformations
Subscription Model	Content-based
Timing model	Can support different timing models implemented in different scopes.
Notification Model	Push
Resource Model	Centralized
Protocol Model	**Interaction protocols**: regular publish/subscribe API **Infrastructure protocols**: not speficied
Versatility Model	OO model and special components: Scopes can be used to implement different publish/subscribe policies and algorithms and Event mappings can be used to implement filters and event transformations. Scopes can be composed, excluded and included providing static variability of the system functionality.

9.15 Flexible Notification Framework (FNF)

Shen and Sun propose a flexible notification framework (or FNF for short) (Shen and Sun 2002) that allows the implementation and combination of different notification policies in the support of collaborative applications. This is accomplished by the use of programmable message queues, where different ingoing and ongoing notification mechanisms can be installed. It allows the manipulation of incoming and outgoing event queues by controlling their event granularity and event forwarding frequency. It also allows the definitions of transformations of notifications (or events), for the implementation of application-specific concurrency control mechanisms. Some of the notification policies supported are: instant propagation of messages (push), user collection of events (pull), deferred publication (send event only upon receiving of commit command) and deferred notification (scheduled pull). Since the frequency of events varies with different applications, for example: chats, desktop sharing, file sharing and so on, operations involving those events can be performed between the event generation and its receipt by another application. Hence operation transformations, specific to different application domain, can be programmed, installed on an event queue, and used to support the application interaction requirements for different application domains. For example, events can be abstracted or summarized in higher-level operations to support application sharing sections; or redundant events can be eliminated (or filtered out) in collaborative editing sections. Another advantage of this approach is the reuse of strategies and policies by the composition of queues (or buffers).

Table 29 Design dimensions for the Flexible Notification Framework

	Flexible Notification Framework (FNF)
Event Model	Record-based. Events represent operations from collaborative applications
Subscription Model	Channel and Topic-based (events are identified by their name). Message transformations can be associated to the channel.
Timing model	Event ordering is guaranteed by a centralized service and by the use of event queues
Notification Model	Push, Pull, periodic Pull.
Resource Model	Distributed: Queue filtering and event processing can be performed both in client or server sides
Protocol Model	**Interaction protocols**: User-defined, using the programmable queues **Infrastructure protocols**: queue composition protocols
Versatility Model	OO programming and use of special components called programmable queues: programmable support different policies and filtering mechanisms, allowing the customization of the notification model. They can be composed to implement more complex protocols.

9.16 FULCRUM

FULCRUM (Boyer and Griswold 2004) is a publish/subscribe system designed to support context-aware applications. The service is constantly evaluating properties (or subscriptions) such as user location, distances, and other runtime properties used by applications dependent on context to react to changes in the environment (for example, the proximity of peers given by the triangular distance formula: $(X_R\text{-}X_W)^2 + (Y_R\text{-}Y_W)^2 < D^2$, where X, Y and D are event attributes, and R and W peers). It uses the open implementation design technique in order to allow the customization of the system to the needs of different clients. Using this approach, FULCRUM allows the configuration and definition of different implementation strategies that exploit the domain's semantics to be used in the subscription language. It also allows the execution of subscriptions in the publisher brokers, called entry nodes, which copes with scalability by the reduction of event traffic between nodes. Reuse of subscription strategies is also achieved. Different implementation strategies can be combined in evaluating properties with similar semantics (for example, the notion of distance).

FULCRUM is built on top of Siena and Jabber (Jabber Software Foundation 2004) (a set of streaming XML protocols and technologies that enable the exchange of messages, presence, and other structured information between clients over the Internet). Hence, it borrows from Siena, its event model, and from Jabber, its protocols. It adds to those systems the concept of active subscriptions in the form of Java code that is executed when a subscription is matched, and enhanced client-side brokers supporting different strategies.

FULCRUM subscription language and the use of active subscriptions allow event aggregation (combination of data from multiple events). The also infrastructure provides optimizations such as suppression of events from the source in case they do not match a peer subscription, which reduces the number of messages in the system, and as a consequence the network traffic, an important requirement for context-aware systems where subscriptions combine events from multiple distributed sources that are constantly generating events.

Table 30 Design dimensions for FULCRUM

	FULCRUM
Event Model	Tuple-based as Siena
Subscription Model	Content-based with support for abstraction and context-aware operations.
Timing model	Partial order of events
Notification Model	Flexible and programmable: java programs are executed in response to subscription matching allowing the implementation of different strategies besides push and pull
Resource Model	Servers are federated and subscriptions are performed in the client side
Protocol Model	**Interaction protocols**: context-aware subscription language that supports active subscriptions **Infrastructure protocols**: Peer-to-peer federation of servers, dissemination of events based on Jabber protocol, protocols for reducing event traffic between peers based on subscription knowledge.
Versatility Model	Allows active subscriptions, support open implementation to add new subscription strategies. Allows reuse or the implementation of new strategies.

9.17 ADEES

The ADEES (Adaptable and Extensible Event Service) (Vargas-Solar and Collet 2002) is a client-side framework that allows the definition of different subscription and notification strategies. It supports different sets of notification and subscription operations, expressed in a meta-model (language). Operations are implemented by different components which can perform different transformations over the events. Operations can be combined in different ways, forming more complex expressions. The system is implemented as an event processing layer on top of CORBA-NS and therefore, inherits its record-based event model, and subject-based subscription model. The client framework, known as Event Manager, hosts the different components that implement the operations, and is also responsible for interpreting the subscriptions, expressed according to a meta-model. The Event Manager can be configured with a set of components that implement different subscription commands and filters such as: event sequence detection, event composition and client-side persistency of events. It also supports different notification policies such as push and pull or other user-defined policies. Each command can be a client to services provided by other commands, allowing their composition. The novelty of the system is its ability to select the set of client-side commands (or components) to have at a given moment, by providing a client-side framework where new commands and notification policies can be installed and used, and to allow their composition in more complex commands. This strategy provides a certain degree of client-side extensibility, programmability and configurability of the subscription and notification languages.

Table 31 Design dimensions for ADEES

	ADEES
Event Model	Record-based
Subscription Model	Subject-based with support for defining new commands. Supports composition
Timing model	Uses the CORBA-NS time model
Notification Model	Flexible and programmable, currently supporting push and pull
Resource Model	All advanced filtering is performed in the client side, but the basic event routing is performed in the server-side (CORBA-NS)
Protocol Model	**Interaction protocols**: supports only publication and subscription of events **Infrastructure protocols**: unknown
Versatility Model	Frameworks, models and components: Based on meta-models (subscription and notification languages descriptions) and client-side components which are used to extend those languages.

9.18 The programmable event-based kernel

A programmable event-based middleware that uses the concept of active subscriptions is presented here (Gazzotti, Mamei et al. 2003). The infrastructure was designed to support the interaction between mobile agents that are co-located in the same host. As agents migrate from one host to another, there is the need for obtaining local context information, from the host the agent migrated to. The idea is to allow mobile agents co-located in the same host to be notified about changes in the environment, and to communicate with one another using a publish/subscribe infrastructure. The proposed infrastructure relies on a simple publish/subscribe kernel, installed in each host, that allows the specification of active subscriptions in java. A subscription is defined by extending a generic subscription class and implementing an event filter and an action in that subclass. The event model is also object-oriented, i.e. events are objects of a generic type (or class), that define their own attributes and methods. Using this model, subscriptions are also object-based, using events as templates. A template is an event object, with fixed attribute values or wild cards, used by the kernel to match the events published in the infrastructure. The programmability is provided in the subscription model, that allows the execution of Java programs whenever events get matched, and the programming of customized filtering policies.

Table 32 Design dimensions for the programmable event-based kernel

	A programmable event-based kernel
Event Model	Object-based
Subscription Model	Object-based: objects used as templates for matching events represented as objects. Support for active subscriptions and filter customization
Timing model	Event ordering is guaranteed by the local event queue
Notification Model	Programmable due to the active subscription approach: notifications are programmed in Java
Resource Model	Centralized, all the processing is performed in the local host, in the scope of the local publish/subscribe infrastructure
Protocol Model	**Interaction protocols**: supports only publication and subscription of events **Infrastructure protocols**: unknown
Versatility Model	Programmable and based on active subscriptions and filter customization that use the Java programming language.

9.19 FACET

FACET (Hunleth and Cytron 2002) is an extensible and configurable implementation of the CORBA Event Service. The extensibility and configurability of features are implemented using Aspect Oriented Programming (Elrad, Filman et al. 2001), which allows the weaving of different features in the middleware. It was initially designed to provide specific configurations that can run on restricted conditions of embedded systems, and can support the real-time requirements of specific applications. Hence, performance and footprint are key design goals of this system (Even though implemented in Java, C++ implementations are part of the future work). In FACET, extensions, in the form of advices, are provided along a skeletal implementation of the standard CORBA event service (CORBA-ES). Configurations of those extensions, implemented as aspects can be defined in order to support different applications requirements (footprint, QoS and filtering capabilities). For being based on the CORBA-ES, the extensibility points go along the main components of this standard, which basically specifies an event channel and standardized push or pull supplier and consumer proxies.

Table 33 Design dimensions for FACET

	FACET
Event Model	Record-based according to CORBA-ES standard
Subscription Model	Topic and channel-based
Timing model	Total order of events guaranteed by the centralized implementation (event channels)
Notification Model	Push and pull, customizable
Resource Model	Centralized
Protocol Model	**Interaction protocols**: regular publish/subscribe through a programmatic API **Infrastructure protocols**: real-time guarantees
Versatility Model	Based on AOP, allowing the configuration, dependency check and static weaving of functional and non-functional components to provide the desired implementation. Focus on real-time applications and embedded systems.

9.20 YANCEES

YANCEES (Silva Filho, de Souza et al. 2003; Silva Filho, De Souza et al. 2004) is a versatile notification service designed to be programmable, configurable and dynamic. It uses a combination of plug-ins, extensible languages, open implementation and composition filters techniques to provide configurability, extensibility and programmability over the main design dimensions of a publish/subscribe system, including support for protocols. YANCEES provides a bare-bones implementation on top of which plug-ins can be added. Plug-ins implement extensions in the subscription, notification and protocol languages. Besides plug-ins, filters can be used to intercept the publication and notification queues of events, performing event transformations, type checking, persistency or other actions. Static services can also be installed in order to support the implementation of plug-ins and filters. Finally, the system uses open implementation to allow the replacement of the event dispatcher with different event routing strategies. Those components are managed and combined together with the help of a configuration language. Reuse is achieved by the dynamic composition of plug-ins by the use of dynamic process trellis architectural style (Factor 1990). Static variability is achieved by a configuration language that allows the installation of plug-ins, filters and services; whereas dynamic variability is possible by the ability to dynamically install plug-ins.

YANCEES was initially designed as a way to allow the customization, extension and implementation of new functionality on top of existing publish/subscribe infrastructures, especially content-based notification servers such as Siena and Elvin, with special focus on collaborative software engineering applications. In fact, the system can be used as an event processing layer on top of those systems. The main focus of the project is its configurability, the expressiveness of the models, the interoperability, integration of applications and the support for many application-specific services. Even though performance and footprint are important, they are not the main concern of this project (since it uses XML and Java).

Table 34 Design dimensions for YANCEES

	YANCEES
Event Model	Extensible, supports record, tuple and others.
Subscription Model	Extensible and programmable according to the installed plug-ins
Timing model	Total order of events in centralized implementation and partial order with specialized time synchronization plug-ins (can be extended to support more advanced features).
Notification Model	Extensible and programmable, supports: push, pull and others
Resource Model	Allows the evaluation of subscriptions in the client and server sides. Also supports filters in both sides.
Protocol Model	**Interaction protocols**: programmable and configurable, current extensions support CASSIUS protocols. **Infrastructure protocols**: programmable and configurable, current extensions support P2P connection
Versatility Model	Applies a combination of software versatility strategies: plug-ins and extensible languages, open implementation and composition filters strategies.

9.21 Other publish/subscribe infrastructures

Besides the systems mentioned here, many other research prototypes and commercial products exist, most of them providing specific functionality for different classes of problems such as Internet-scale notification systems, peer-to-peer networks, mobility, awareness, software monitoring, distributed processes communication and so on. The survey of all those systems is beyond the scope of this document, which main focus is to show the limitations of current publish/subscribe infrastructures with respect to their versatility, and to propose some research venues in the area by enlisting promising and existing approaches to the problem.

Information about publish/subscribe infrastructures can be found in the following surveys (Baldoni, Contenti et al. 2003; Eugster, Lausanne et al. 2003). A survey of event-based systems for software monitoring is also presented here (Dias 2002).

10 Analysis of publish/subscribe infrastructures according to their versatility

Based on the systems previously surveyed, we built a table presenting a more in depth classification of the systems with respect to the proposed versatility dimensions. For each one of the system, Table 35 presents the main strategy used to the problem.

It is important to mention that usability is a comprehensive set of properties as mentioned in 7.11, which in the case of software development, is highly influenced by the versatility approach

used in its development. Hence, when populating the usability column as follows, we mention the interaction mechanisms of the users with the system, which includes the mechanisms the versatility approaches provide to achieve the other versatility dimensions.

Table 35 List of publish/subscribe infrastructures and their versatility approaches

System / Versatility	Extensibility	Programmability	Reuse	Static variability	Dynamic variability	Usability
CORBA-NS	OO model	OO model and channel QoS API	As a pub/sub API	n/a	n/a	pub/sub API and subscription language
JMS	OO model	OO model and channel QoS API	As a pub/sub API	n/a	n/a	Pub/sub API and filters
READY	OO model	OO model and channel QoS API	As a pub/sub API	n/a	Allows move of subscription from client to server	Complex pub/sub API and subscription language
Siena	OO model	OO model	As a pub/sub API	n/a	n/a	Pub/sub API and filters
Herald	OO model	OO model	As a pub/sub API	n/a	n/a	Pub/sub API
Elvin	OO model	OO model	As a pub/sub API	n/a	n/a	Pub/sub API and subscription language
Gryphon	OO model	OO model	As a pub/sub API with SQL-like language	n/a	n/a	SQL-lique query language
JEDI	OO model	OO model	As an extended pub/sub API	n/a	n/a	Pub/sub API and filters
CASSIUS	OO model	OO model	As an extended pub/sub API	n/a	n/a	API and query language
KHRONIKA	LISP	LISP	LISP	n/a	n/a	API and programming language
GEM	OO model infrastructure and Rule-based language	OO model infrastructure and Rule-based language	As an event processing language	n/a	n/a	Rule-based language
YEAST	C language and Rule-based language	C language and Rule-based language	As an event action language integrated to UNIX shell	n/a	n/a	Shell script programming
TSpaces	C language and Programming API	C language and Programming API	As a tuple space API	n/a	n/a	Programming API
The Modular Event System	OO model with scopes and event mappings components	OO model with scopes and event mappings components	Of scopes and event mappings	Based on scopes and mappings composition	n/a	Component-based abstraction

FNF	OO model and programmable queue components	OO model and programmable queue components	Of queues and filtering policies	Based on queue composition	n/a	API and OO programming
Fulcrum	Open implementation	Open implementation	Of subscription strategies	Based on Open implementation	n/a	Open implementation and subscription API
ADEES	Frameworks, meta-models and components	Frameworks, meta-models and components	Of event processing components	Configuration of client-side components set	n/a	Frameworks, meta-models and component model
FACET	AOP	AOP	Of aspects	configuration of aspects	n/a	AOP programming
YANCEES	Plug-ins and extensible languages, filters and open implementation	Plug-ins and extensible languages, filters and open implementation	Of plug-ins, filters, and third-party components	Configuration of plug-ins, languages, filters and other components	Plug-in oriented	Plug-ins and extensible languages, filters and open implementation

Based on our concept of versatility, it is clear that many of the systems previously surveyed marginally address the versatility *requirements* we propose. Instead of being designed for evolution and configuration of their features set, those systems strive to support different application domains and requirements by applying simpler but limited strategies. The main strategies employed are: (1) to build the exactly set of required functionality for the application domain the system will serve; or (2) to provide a minimal set of features and let the extension to the application level (minimal core); or (3) to support the larger set of features as possible (also known as one-size-fits-all or monolithic approaches); or (4) to provide a more flexible (i.e. programmable) subscription/notification languages in the context of the variability of the application domain; and finally, more recently, (5) to allow the general adaptation, configuration and programming of the major dimensions of the system. This last category of systems strives to provide versatility and support for different application domains.

10.1 Specialized notification servers

A largely used strategy, and the less versatile of all, is the development of specialized solutions, in order words, to “build the right tool for each job”. Using this strategy, notification servers have been developed from scratch, supporting different domains, ranging from workflow management systems and mobility, as the example of the JEDI; context-aware applications, as Gryphon; awareness applications, as the case of CASSIUS; and so on. In all those cases, the basic publish/subscribe mechanism is implemented together with a specialized set of features tailored to the needs of the application they intent to serve.

As “right tools for each job”, they tend to perform better than more generalized approaches. For not being designed to support different application domains, their implementation can be simplified and focused on the problem they are solving. It is common, for example, the use of record-based event models and context-specific subscription languages. A side effect of this specialization, however, is the lost of generality and the natural incompatibility with other publish/subscribe networks. Finally, since versatility techniques are generally not used, their evolution and adaptation to new domains is difficult.

10.2 Minimal core infrastructures

A more recent category of notification servers which includes Siena, Elvin, and Herald, provide a relatively simple but optimized set of functionalities, with the ability to efficiently process a large amount of events and scale to Internet-wide proportions. For such, they adopt the content-based subscriptions and a flexible event model based on events defined as variable length attribute/value pairs. Scalability is achieved by the federation of servers. In fact, the simplified, but generalized, subscription language is a result of the striving for scalability, which limits the expressiveness of the subscription language in favor of routing performance (Carzaniga, Rosenblum et al. 1999). TSpaces also falls in this category, for providing a minimal API for tuple space manipulation, permitting the implementation of more advanced primitives.

In spite of a current trend for Internet-scale routing requirements, all those systems provide a basic set of primitives (typically only the publication and subscription of events), that can be used to implement more advanced features in the client applications. In fact, systems as YANCEES and FULCRUM use Siena as a component of their systems, which simple API is used as a basic publish/subscribe mechanism on top of which new functionality is implemented. Additionally, the use of content-based routing and tuple-based events provides compatibility with other subscription models such as channel-based and topic-based. In spite of this generality, this approach also presents a low degree of versatility in their infrastructure. The implementation of more advanced features such as security, mobility and advanced event processing, for example, require deeper design changes and/or new commands in their subscription and notification languages, which is not natively supported by those systems.

10.3 One-size-fits-all implementations

One-size-fits-all notification servers such as the CORBA Notification Service, the JMS standard from Sun or even READY, adopt a different approach. They strive to satisfy the large number of application requirements as possible, by implementing a broad spectrum of event, subscription, notification and resource options and policies.

This strategy allows the development of systems able to support a large set of application domains. For using the same infrastructure, systems can more easily interoperate. A common problem of those approaches, however, is their bulkiness. The large set of features those servers provide usually implies in excessive consumption of resources, both in the client and server sides. As stressed by Wirth(Wirth 1995), this approach leads to software that expands to fit all available system resources.

10.4 Domain-specific versatile notification servers

The need for variability demanded by some application domains, resulted in more flexible infrastructures. Systems as FULCRUM, the Shen and Sun FNF, ADEES and KHRONIKA are some examples. They are designed to support the variability within specific application domains, in this case, context-aware applications (FULCRUM), concurrency control in collaborative settings (FNF), inter-agent communication (ADEES), interoperability and heterogeneity (The Modular Notification Service) and Awareness (KHRONIKA). For such, those systems apply more advanced strategies such as open implementation, in the case of FULCRUM; programmable queues in the case of FNF, software frameworks (ADEES), components (The Modular Notification Service) and programmable notification styles as KHRONIKA.

Also included in this category are rule-based systems as YEAST and GEM. They provide domain-specific languages that support the concept of rules (or active subscriptions). Such approach provides some programmability in the way events are processed and notifications are generated,

which allow their integration with system tools, as the case of YEAST, or the detection of critical conditions in software applications, as GEM. Those systems, however, are created for specific proposes such as UNIX application integration (YEAST scripting and action language) and distributed systems monitoring (GEM), and their lack of use of more advanced versatility techniques hinders their portability to different application domains.

Hence, a common strategy that permeates most of those approaches is the combination of the concept of active subscriptions (subscriptions that specify a set of actions as a result of an event matching). The action part of the subscription is implemented with different approaches such as frameworks or open implementations.

While being able to support the variability inside their application domains, those systems are not designed to support other application domains, having their variability defined in the exact points where the application they support requires this variability. Another consequence is the lack of interoperability in most of those solutions.

10.5 Generally versatile notification servers

Finally, a new category of publish/subscribe infrastructures that strives to support various levels of versatility properties mentioned in 6.1 have lately received increased research attention. Those systems apply current software engineering techniques such as Aspect Oriented Programming (FACET) and extensible languages, plug-ins and composition filters (YANCEES), improving their ability to expand and contract their set of features in order to support variety of application domains.

We can also classify ADEES in this category since it provides a versatile subscription and notification model, that is not bound to any specific application domain. Even though it does not provide a flexible event or protocol models (as the case of FACET also), it represents an effort towards of a fully versatile publish/subscribe system.

Systems as YANCEES and FACET allow the customization of many aspects of the infrastructure including the set of features supported (its footprint) and the notification strategies provided. YANCEES goes beyond FACET in providing a flexible event model, an extensible subscription language that uses plug-ins and a protocol model that allows the implementation of other sorts of interaction with the service.

A constant challenge of those new approaches is their usability. The use of new techniques, not well known in the practitioners' community usually requires a steep learning curve, and the understanding of the models behind those systems is not usually a trivial task, both factors can compensate the gain in cost associated to the extensibility, reuse and adaptability of those systems.

A summary of the herein described approaches is presented in Table 36 as follows.

Table 36 Summary of most popular versatility approaches for publish/subscribe infrastructures

Versatility Strategy	Example systems	Description	Strengths	Limitations
Application specific	CASSIUS, JEDI, Gryphon	Provide a fixed set of requirements demanded by a specific application domain	Right tool for the domain, requiring low or no adaptation	Limited interoperability and portability to different application domains
Minimal core	Siena, Herald, Elvin, TSpaces	Provides a minimal an optimized API to cope with fast event routing or tuple-spaces manipulation	Useful for applications that demand fast and simple publish/subscribe services or tuple manipulation	Limited subscription capability, with a minimum set of features requiring extensions to be made in the client side
One-size-fits-all	CORBA-NS, READY, JMS	Support a large set of features to address the requirements of the majority of applications	Can support a larger set of applications, coping with interoperability and application evolution	Bulkiness (memory footprint), may not support some application-specific requirements, complex API
Domain-specific versatile	FULCRUM, KHRONIKA, FNF, Modular Event System, ADEES, GEM, YEAST	Provide a flexibility around the variability points of the application domain (the variability is implemented through different approaches)	Right tool for domains that require a certain degree of variability	Limited interoperability and portability to different application domains Support only domain-specific applications
Generally Versatile	FACET, YANCEES, ADEES,	Provide generalized support for the extension, addition and selection of new features	Address many of the versatility properties we propose	Steep learning curve to understand extension mechanisms, not so clear system models.

11 Promising research topics

One next step would be the realization of a comparative study between versatility approaches, comparing their effectiveness in addressing the versatility properties in the publish/subscribe domain. The first step towards the identification of methodologies and approaches to apply those techniques is their understanding. The next step would be to research and identify successful cases where those approaches are used in the solution of the versatility problem, and identify patterns, and guidelines that lead to their proper use.

Another venue is to study and compare the combined use of such techniques in the solution of the publish/subscribe problem. How could the benefits of one or another approach be combined in an application without inheriting its weaknesses and without overcomplicating the implementation? For example, one can try to combine extensible languages for representing subscriptions and events, aspect-oriented programming for implementing cross-cutting concerns, plug-ins for extensibility, programmability and dynamism, software patterns for source code clarity and extensibility, active subscriptions for advanced notifications, and composition filters for implementing additional protocols, all under a common component model, using the services provided by an application container. The challenge then would be to combine those strategies, using their strengths without inheriting their weaknesses and without overcomplicating the design. Our experience in the design of YANCEES showed that the combination of those approaches, even though may achieve hither degrees of extensibility, programmability and configurability, may result in an implementation that is not usable, mainly because of the steep learning curve associated to the versatility approaches and to the understanding and use of the system.

Hence, in direct opposition to the previous idea, one could conceive a simple but generic model that would be the basis for the construction of more complex systems; a model that uses composition of simple standardized components (or kernels), for example that could be easy for practitioners to understand and apply. Simplicity is the key to the success of many approaches that, even though may not address all the proposed versatility requirements, do a good job in capturing the essence of the problem. Usability is an important aspect in the selection of the extensibility technique but, ironically, is many times ignored by practitioners or even researches that build those approaches. In other words, solving the problem through a simple and comprehensible way must be the goal of a solution.

12 Conclusions

In the context of software engineering, versatility can be defined as the ability of a computational system to serve multiple purposes or to accommodate the requirements of different use situations which, in terms of software qualities can be defined in its ability to support extension, programmability and reuse; besides of being able to be dynamically and statically configured to different purposes in usable ways. The survey of existing publish/subscribe infrastructures shows that most of existing research and industrial publish/subscribe infrastructures are not versatile enough to address the requirements of new application domains, or to be adapted to different application requirements.

More recently, domain-specific versatile and generally versatile approaches have been developed moved by the need for variability in the application domains. In the case of the domain-specific versatile solutions, systems as ADEES, FULCRUM and FNF provide ingenious ways to address their domain variability requirements, which is usually addressed by modern approaches such as components, open implementations and extensible languages. In generally versatile approaches such as YANCEES, a set of techniques are used to address many of the proposed versatility qualities. Those approaches, however, are still insufficient for the proposed set of qualities, failing short mainly in the usability requirement.

The use of multiple versatility approaches is an interesting idea. It strives to combine the strengths of multiple approaches in the addressing of versatility. In fact, systems as YANCEES, that employs different techniques in addressing the versatility requirements (open implementation, plug-ins, extensible languages and frameworks), achieves a much higher degree of versatility than those systems based on a single solution, such as FACET which mainly relies on AOP. The drawback, however, is the reduction of usability, since users need learn and use not only one but many different approaches in a meaningful way.

Another pitfall in the use of multiple approaches is the possibility of their misuse. If not used to the specific point they excel, they can jeopardize, instead of improve, the overall system design, impacting in other dimensions. For example, approaches such as computational reflection may slow down the whole system if generally used. In another example, software frameworks may actually diminish extensibility if the right adaptation points are not identified through a thorough domain analysis.

Another interesting observation is that the principles of reuse, extensibility, programmability and usability are not always compatible with one another. In fact, before applying one or another technique, both practitioners and researches need to account for the trade-offs from each approach. In other words, due to the complexity of those requirements and the availability of too many approaches to address the problems that they raise, it is very challenging to come up with an implementation that fits all provided properties in all situations. Hence, based on the strategies surveyed, we come to the conclusion that the choice of the right versatility approach for the right versatile implementation must be driven by the application domain and the foreseeable uses of the

application. In fact, we observed the following trade-offs, if combining the strengths and limitations of each versatility approach:

Decomposition and abstraction. According to Guttag (Guttag 2001), since good programs are those who persist over time and are able to evolve to address new improvements and requirements, no matter how good a program was originally, or how good it performs; ultimately, performance is determined by how easy it is to repeatedly modify and optimize the software. In other words, according to Guttag, *"programming is about managing complexity in a way that facilitates change, and there are two powerful mechanisms for accomplishing this: decomposition and abstraction. Decomposition creates structure in a program, and abstraction suppresses detail. The key is to suppress the appropriate details"*. Hence, on the construction of software, and in special publish/subscribe infrastructures, one should definitely apply techniques that allow the partition of the problem (decomposition), hiding unnecessary details (abstraction).

Freedom versus restriction. Abstraction is a key idea in handling with complexity. It is largely used by frameworks and component-based approaches to hide unnecessary implementation details from the user. This characteristic is also important in protecting the system from its users from modifying critical parts of the software. This approach, however, can make the system harder to change since the access to the implementation details is limited. Approaches such as open implementation, AOP and meta-level programming go in the opposite direction, permitting a better customization of the system, bypassing some of those classical restrictions. The power provided by those approaches and languages, however, may lead to solutions that instead of improving modularity, reuse and clarity of software. For example, one can try to completely define a system in terms of aspects, modeling every functional requirement as meta-level programs, instead of implementing only the non-functional requirements as aspects. This approach may result in very inefficient and not so clear source code. As a consequence, good design practices and possible restrictions need to be observed when combining those approaches in order to achieve a balance between encapsulation, abstraction and openness, designing a system that is easy to change and evolve at the same time that prevents end users from misusing the techniques it applies.

Usability versus usefulness. Another critical point in the process of design for evolution is usability. The system and the versatility mechanisms used must be relatively easy to understand and use. The versatility approach adopted must be such that the customization and evolution costs of customizing and extending the infrastructure is inferior or equal to the cost of designing and building a new system from scratch. In other words, the reason there are too many solutions to the same publish/subscribe problem is that, at least initially, the cost of producing a new publish/subscribes system is seductively low, and the reuse or customization of existing solutions is not usually as easy as desired. In other words, the use of a new versatility approach and its application to extend or customize a generalized solution may require a steep learning curve or may have high customization and extension costs. Hence, an approach must not only be useful but also usable.

Efficiency versus effectiveness. A common consequence of the application of some versatility techniques is performance degradation. It is a price to be paid on account of abstraction, modularization and, sometimes usability of a versatility approach. However, if used in a right way, the gain in versatility may compensate or even surpasses the long term cost of using not so versatile approaches. In fact, a proper use of a versatility technique, respecting its limitations, has shown comparable or even in some cases, slightly better in terms of performance, than those solutions using more traditional software engineering approaches. An example is given by (Zhang and Jacobsen 2004) where a CORBA middleware refactored with aspects, performed better and resulted in software with less lines of code than the original pure OO implementation.

Configurable versus inflexible implementations. The need for configuration management is another issue that comes as a consequence of modularization. As modularization addresses complexity and improves reuse and extensibility, inter-dependencies between these modules must be observed. Automatic incompatibility checks must be performed, protecting the developers from wrong module versions and configuration mismatches. Besides dependencies, variability becomes an issue. As parts of the system can be configured and exchanged, the need for automatic mechanisms to manage the software evolution becomes evident. This is an important aspect to be considered in the use of approaches such as plug-ins, components or even AOP (as exemplified by FACET).

Testing and debugging. Error handling and application debugging is another issue to be managed in versatile solutions. On designing for extensibility and programmability one usually introduces many variability points in the systems that will be usually implemented by inexpedient developers that do not want to know about the hidden parts of the system. A challenge in those approaches is then is to keep the system tolerant to bugs and customization errors coming from user's extensions. In the case of the versatility techniques surveyed, approaches such as CBSE, frameworks, plug-ins and open implementation are more sensitive to these kinds of interference, which may either refrain the adoption of such approaches in more generalized solutions, or impose a large amount of work in devising mechanisms that allow improved application debugging and fault isolation.

As a concluding remark, a general observation is that, as new techniques are devised and the fundamental software characteristics such as complexity and changeability are tamed, the need for versatility will be always a concern. This comes from the fact that, as new techniques and approaches are created and their impact to software engineering changeability produces more malleable software, the demand on software tends to grow, and with it, its complexity and need for generality. Today's software, with systems build of millions of lines of code, as modern operating systems, is a proof of that modern versatility techniques such as object oriented programming, software patterns, frameworks and component-based software engineering can gradually overcome the complexity of software. The problem however, is the vicious cycle that gets formed by the continuous growth of software complexity, motivated by those advances. This fact forces current approaches to their limit, demanding new techniques. In this context, the search for the "silver bullet" will always be part of the software engineering research.

Acknowledgements

This research was supported by the U.S. National Science Foundation under grant numbers 0205724 and 0326105, and by the Intel Corporation

References

Aksit, M. and B. Tekinerdogan (1998). Solving the Modeling Problems of Object-Oriented Languages by Composing Multiple Aspects Using Composition Filters. 12th European Conference on Object-Oriented Programming - AOP'98 Workshop, Brussels, Belgium.

Bachrach, J. and K. Playford (2001). The Java syntactic extender (JSE). 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'01), Tampa Bay, FL, USA.

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

Banavar, G., T. Chandra, et al. (1999). An efficient multicast protocol for content-based publish-subscribe systems. 19th IEEE International Conference on Distributed Computing Systems.

Banavar, G., T. Chandra, et al. (1999). "A Case for Message Oriented Middleware." Lecture Notes in Computer Science **1693**.

Batory, D., B. Lofaso, et al. (1998). JTS: tools for implementing domain-specific languages. Fifth International Conference on Software Reuse, Victoria, BC, Canada.

Batory, D. and S. O'Malley (1992). "The Design and Implmementation of Hierarchical Software Systems with Reusable Components." ACM TOSEM **1**(4): 355-398.

Beck, K. and R. Johnson (1994). Patterns Generate Architectures. Lecture Notes in Computer Science. Springer-Verlag. **821:** 139-149.

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

Bernstein, P. A. (1996). Middleware: a model for distributed system services. Communications of the ACM. **39:** 86-98.

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

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.

Cardone, R. (1999). On the Relationship of Aspect-Oriented Programming and GenVoca. 9th Workshop on Institutionalizing Software Reuse, University of Texas, Austin, TX.

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

Cardone, R. and C. Lin (2001). Comparing Frameworks and Layered Refinement. 23td International Conference on Software Engineering, Toronto, CA.

Carzaniga, A., D. S. Rosenblum, et al. (1999). Challenges for Distributed Event Services: Scalability vs. Expressiveness. ICSE '99 Workshop on Engineering Distributed Objects (EDO '99), Los Angeles, CA, USA.

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.

Carzaniga, A. and A. L. Wolf (2001). Content-Based Networking: A New Communication Infrastructure. NSF Workshop on an Infrastructure for Mobile and Wireless Systems.

Chatley, R., S. Eisenbach, et al. (2003). Painless Plugins. Technical Report - http://www.doc.ic.ac.uk/~rbc/writings/pp.pdf. London, Imperial College London.

Clarke, M. and G. Coulson (1998). An architecture for dynamically extensible operating systems. International Conference on Configurable Distributed Systems (ICCDS'98), Annapolis, MA, USA.

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

Codenie, W., K. D. Hondt, et al. (1997). "From custom applications to domain-specific frameworks." Communications of the ACM **40**(10): 70-77.

Constantinides, C. A., A. Bader, et al. (2000). "Designing an aspect-oriented framework in an object-oriented environment." ACM Computing Surveys (CSUR) **32**(1es).

Costa, F. M., G. S. Blair, et al. (2000). "Experiments with an architecture for reflective middleware." Integrated Computer-Aided Engineering Journal **7**(4): 313-325.

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 (1999). Components and generative programming (invited paper). 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, Toulouse, France, Springer-Verlag.

Dashofy, E., A. v. d. Hoek, et al. (2005). "A Comprehensive Approach for the Development of Modular Software Architecture Description Languages." ACM Transactions on Software Engineering and Methodology **to appear**.

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

Dias, M. (2002). Software Monitoring - A Survey. Irvine, UC Irvine.

Dias, M. and D. Richardson (2003). The Role of Event Description on Architecting Dependable Systems. Lecture Notes in Computer Science - Book on Architecting Dependable Systems. Spring-Verlag.

Dingel, J., D. Garlan, et al. (1998). Reasoning about implicit invocation. 6th International Symposium on the Foundations of Software Engineering (FSE-6), Lake Buena Vista, FL, USA.

Dourish, P. and S. Bly (1992). Portholes: Supporting Distributed Awareness in a Collaborative Work Group. ACM Conference on Human Factors in Computing Systems (CHI '92), Monterey, California, USA, ACM Press.

Edwards, G., G. Deng, et al. Model-driven Configuration and Deployment of Component Middleware Publish/Subscribe Services.

Elrad, T., R. E. Filman, et al. (2001). "Aspect-oriented programming: Introduction." Communications of the ACM **44**(10): 29-32.

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

Eugster, P. T., S. Lausanne, et al. (2003). "The Many Faces of Publish/Subscribe." ACM Computing Surveys (CSUR) **35**(2): 114-131.

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.

Fiege, L., G. Mühl, et al. (2002). "Modular event-based systems." The Knowledge Engineering Review **17**(4): 359 - 388.

Filman, R. E., S. Barrett, et al. (2001). Inserting ilities by controlling communications. Communications of the ACM. **45:** 116-122.

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.

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

Gamma, E. (2001). Design Patterns - Ten Years Later. Software Pioneers (Contributions to Software Engineering), Springer.

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

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

Gazzotti, M., M. Mamei, et al. (2003). A Programmable Event-based Middleware for Mobile Organizations. 11th EUROMICRO Conference on Parallel, Distributed, and Network Processing, Genova, Italy., IEEE CS Press.

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

Group, O. M. (2002). CORBA Components. OMG Document formal/2002-06-65, 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.

Gutierrez-Nolasco, S. and N. Venkatasubramanian (2001). Design Patterns for Safe Reflective Middleware. Workshop Towards Patterns and Pattern Languages for Object-Oriented Distributed Real-Time and Embedded Systems (OOPSLA 2001).

Guttag, J. V. (2001). Abstract Data Types, Then and Now. Software Pioneers (Contributions to Software Engineering), Springer.

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 Embeded Systems, Berlin, Germany, ACM Press.

IBM (2003). Websphere MQ Family, IBM. **2003**.

International, O. T. (2003). Eclipse Platform Technical Review. URL: http://eclipse.org/whitepapers/eclipse-overview.pdf, IBM Corporation.

Jabber Software Foundation (2004). Jabber: Open Instant Messaging and a Whole Lot More - http://www.jabber.org/.

Jacques, M. (2004). API Usability: Guidelines to improve your code ease of use - http://www.codeproject.com/gen/design/APIUsabilityArticle.asp, The Code Project.

Jin, Y. and R. Storm (2003). Relational Subscription Middleware for Internet-Scale Publish-Subscribe. International Workshop on Distributed Event-Based Systems (DEBS'03), San Diego.

Johnson, R. E. and B. Foote (1988). "Designing Reusable Classes." Journal of Object Oriented Programming - JOOP **1**(2): 22-35.

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.

Kiczales, G. (1996). Beyond the black box: open implementation. IEEE Software. **13:** 8,10-11.

Kiczales, G., E. Hilsdale, et al. (2001). Getting started with ASPECTJ. Communications of the ACM. ACM. **44:** 59-65.

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

Krishnamurthy, B. and D. S. Rosenblum (1995). "Yeast: A General Purpose Event-Action System." IEEE Transactions on Software Engineering **21**(10): 845-857.

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

Ledoux, T. (1999). "OpenCorba: A Reflective Open Broker." Lecture Notes in Computer Science **1616**: 197-214.

Lientz, B. P. and E. B. Swanson (1980). Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations (Ch. 27). Rading, MA, Addison-Wesley.

Lopes, C. V. (2002). Aspect-Oriented Programming: An Historical Perspective (What's in a Name?). Technical Report UCI-ISR-02-5. Irvine, Institute for Software Research.

Lopes, C. V. and T. C. Ngo (2004). The Aspect Oriented Markup Language and its Support of Aspect Plugins - UCI-ISR-04-8. Irvine, UC, Irvine.

Lövstrand, L. (1991). Being Selectively Aware with the Khronika System. European Conference on Computer Supported Cooperative Work (ECSCW '91), Amsterdam, The Netherlands.

Maeda, C., A. Lee, et al. (1997). Open implementation analysis and design. 1997 symposium on Software reusability, Boston, MA, ACM Press.

Mansouri-Samani, M. and M. Sloman (1997). GEM: A Generalised Event Monitoring Language for Distributed Systems. IFIP/IEEE International Conference on Distributed Platforms (ICODP/ICDP'97), Toronto, Canada.

Mayer, J., I. Melzer, et al. (2003). Lightweight Plug-In-Based Application Development. Lecture Notes in Computer Science. M. M. M. Aksit, R. Unland, Springer-Verlag Heidelberg. **2591:** 87 - 102.

McIlroy, M. D. (1968). Mass Produced Software Components. In Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee. P. Naur and B. Randell (eds.), Garmisch, Germany.

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

Microsoft (2003). Building Distributed Applications with Message Queuing Middleware.

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.

Nielsen, J. (1993). What is Usability? Usability Engineering (Chapter 2). J. Nielsen, Morgan Kaufman**:** 23-48.

Norman, D. (1988). The design of everyday things.

Notkin, D. and W. G. Griswold (1988). Extension and software development. 10th international conference on Software engineering, Singapore, IEEE Compter Societ press.

Oliva, A. and L. E. Buzato (1999). The Design and Implementation of Guaraná. 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99), San Diego, CA.

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

OMG (2002). CORBACos: Notification Service Specification v1.0.1, Object Management Group.

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.

Paton, N. W. and O. Diaz. (1999). "Active Database Systems." ACM Computing Surveys **31**(1): 63-103.

Patterson, J. F., M. Day, et al. (1996). Notification servers for synchronous groupware. ACM conference on Computer supported cooperative work (CSCW'96), Boston, Massachusetts.

Roberts, D. and R. Johnson (1996). Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. Pattern Languages of Program Design 3. A. Wesley.

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.

Sahingöz, Ö. K. and N. Erdogan (2003). RUBCES: Rule Based Composite Event System. XII. Turkish Artificial Intelligence and Neural Network Symp. (TAINN'03), Turkey.

Sahingöz, Ö. K. and N. Erdogan (2003). "RUBDES: Rule Based Distributed Event System." Lecture Notes in Computer Science **2869/2003**: 284-291.

Sarma, A., Z. Noroozi, et al. (2003). Palantír: Raising Awareness among Configuration Management Workspaces. Twenty-fifth International Conference on Software Engineering, Portland, Oregon.

Schmidt, D. C. and C. Cleeland (1999). Applying a Pattern Language to Develop Extensible ORB Middleware. EEE Communications Magazine. L. Rising, Cambridge University Press. **37:** 54-63.

Schmidt, D. C. and C. Cleeland (2000). Applying a Pattern Language to Develop Extensible ORB Middleware. Design Patterns and Communications. L. Rising, Cambridge University Press.

Shen, H. and C. Sun (2002). Flexible notification for collaborative systems. ACM conference on Computer supported cooperative work (CSCW'02), New Orleans, Louisiana, USA, ACM.

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., C. R. B. De Souza, et al. (2004). Design and Experiments with YANCEES, a Versatile Publish-Subscirbe Service - TR-UCI-ISR-04-1. Irvine, CA, University of California, Irvine.

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

Silva-Filho, R. S., C. R. B. d. Souza, et al. (2004). Design and Experiments with YANCEES, a Versatile Publish-Subscirbe Service. Irvine, CA, Institute for Software Research.

Singhai, A., A. Sane, et al. (1998). Quarterware for middleware. 18th International Conference on Distributed Computing Systems, Amsterdam, The Netherlands.

Sommerville, I. (2001). Software Engineering (6th Edition).

Sonic (2003). Using SonicMQ® to Extend J2EE Application Server Capabilities, Sonic Software. **2003**.

SUN (2003). Java Message Service API, SUN. **2003**.

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

Taylor, R. N., N. Medvidovic, et al. (1996). A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering. **22:** 390-406.

Tokuda, L. and D. Batory (2001). "Evolving Object-Oriented Designs with Refactorings." Journal of Automated Software Engineering **8**(1): 89-120.

Vargas-Solar, G. and C. Collet (2002). ADEES: An Adaptable and Extensible Event Based Infrastructure. 13th International Conference, DEXA 2002 Aix-en-Provence.

Venkatasubramanian, N. (2002). Safe Composability of Middleware Services. Communications of the ACM. **45:** 49-52.

Wichman, J. C. (1999). ComposeJ: The Development of a Preprocessor to Facilitate Composition Filters in the Java Language, Master's thesis. Dept. of Computer Science. Twente, University of Twente.

Wilson, G. V. (2004). Extensible programming for the 21st century. ACM Queue. **2:** 48-57.

Wirth, N. (1971). Program Development by Stepwise Refinement. Communications of the ACM. **14:** 221-227.

Wirth, N. (1995). A plea for lean software. IEEE Computer. **28:** 64-68.

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 Collaboratice 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.