YANCEES Issues and Design Principles

Roberto Silveira Silva Filho
UC Irvine
Bren School of Information and Computer Sciences
Department of Informatics
5029 Donald Bren Hall
Irvine, CA 92697-3440

rsilvafi@ics.uci.edu

1 Versatility issues in YANCEES

The development of YANCEES, with its high flexibility and configurability demand has many benefits (ability to extend its different aspects, reuse of existing plug-ins, configurability). However, those benefits come with extra costs. Our experience with YANCEES revealed major issues that lead us to draw a set of observations in this domain.

1.1 Generality and late binding lead to interference and low reuse

The support for variability is in YANCEES was provided by the use of implementation strategies that delay extensions to as late as possible in the software process. As a consequence, binding was delayed to runtime and few assumptions were encoded in the SPL common assets. Another strategy was the use of a generalized common core (micro kernel architecture) with the move of specificity to the software extensions. The result is incompatibility between features in different variation points that leads to interference.

In other words, the more generic an API is, the looser are the contracts it provides, and the easier it becomes to unintentionally break the underlying assumptions of those contracts.

1.1.1 The variability dimensions in pub/sub are not orthogonal

The variability dimensions adopted in the design of YANCEES were inter-related by a set of fundamental problem dependencies. Those dependencies makes possible for changes in different dimensions to affect fundamental assumptions in other dimensions. As a consequence, the reuse is jeopardized. Those issues are aggravated by the generality of the solution, which creates weak contracts where those dependencies are not documented.

1.1.2 Generality issues

Separation of concerns into components presumes the integration of those components into a system. This integration requires well defined protocols and assumptions. Low coupling and cohesion may lead to generic interfaces with low context to their developers and users. Implicit dependencies and assumptions are not documented. Changes in different parts of the system driven by variability, may break those assumptions, leading to unexpected behavior (Silva Filho and Redmiles 2007).

1.1.3 Late binding issues

The use of late binding approaches, with the use of third party extensions suffer from the unpredictability problem. Integration testing is not always possible, and implementations make assumptions that may not be valid for all instances of the product line. The result is feature interference.

1.2 Documentation of dependencies becomes more important as generality increases

As a consequence of point **Error! Reference source not found.**, documentation is necessary to recover those assumptions and prevent interference. Those assumptions are translated into dependencies. This documentation, however, faces traceability problems (the representation of design concerns in code), and unpredictability (changes in those assumptions triggered by hidden dependencies.

1.2.1 The management of dependencies becomes essential

Hence, one of the costs of variability is the need for the management of hidden (or implicit) dependencies. Dependencies are the basic bounds between variation points, assumptions and feature implementations. If not documented, they break

2 Design principles

The development of flexible solutions need to achieve a balance between: Complexity, Usability and Variability. At the same time, it needs to address different issues that come with flexibility: Hidden or implicit dependencies; feature interference; lack of traceability from design elements to code, and unpredictability of new extensions.

They are called principles because they can be applied to flexible publish/subscribe middleware in general and not only to YANCEES scope and domain. In fact, most of those principles can also be applied to the implementation of other kinds of middleware.

Some of those principles are not new. They presented here to document the main design decisions in the implementation of YANCEES, being useful in the design of similar systems.

Principles: are means by which we achieve the characteristics demanded by the problem domain, in our case, these are: extensibility, configurability, reuse, usability and the publish/subscribe middleware qualities of service. However, those requirements need to face some issues. Therefore, trade-offs must be observed. The lessons learned in dealing with those issues are then translated into principles that should be followed in the implementation of versatile infrastructures.

2.1 The source of the Principles: Our experience

- Swirl project with protocol, fast routing and global filtering needs (Impromputu and Vavoom (DePaula, Ding et al. 2005)). This project faced some issues with variability, performance, and feature interference exposing many issues in YANCEES

- EDEM, CASSIUS and advanced subscription requirements. They motivated different extensions in the model.

2.2 Publish/subscribe specific principles

Those are principles applied to the publish/subscribe domain specifically.

2.2.1 The design of pub/sub infrastructures must support flexibility in different variation points.

One cannot predict all the variability an infrastructure will have. Hence, there is no one-size-fits all solution to the problem of flexibility in publish/subscribe infrastructure. Consequently, pub/sub infrastructures need to support the evolution and configuration of their different aspects. From a literature survey (Silva Filho and Redmiles 2005).

One must observe the trade-offs with respect to leanness and robustness. A microkernel approach, allows the development of leaner extensions, that utilize the existing infrastructure; whereas a component-based approach requires more robust components, that must support their use in many different combinations.

In YANCEES design, the common aspect of pub/sub: the process of receiving, routing and delivering events is uniform in the application domain, this lead us to the choice of a micro-kernel approach.

2.2.2 Co-evolution of infrastructure and languages (pub/sub specific)

The features implemented in the infrastructure must be externalized to the end-users in the way of subscription, notification and event format languages.

The addition of new commands in the subscription language must come with the adequate implementation of those commands in the infrastructure. Hence, both infrastructure and languages must be extensible.

In YANCEES, this is achieved by the use of subscription languages defined using XMLSchema, an extensible meta-model.

The co-evolution is also supported by dynamic parsers that allocate valid commands in terms of installed plug-ins in the infrastructure, into coherent event processing trees.

2.2.3 Reuse existing plug-ins in the built of more complex expressions

Event processing languages build upon existing functionality in the construction of more complex expressions. Event abstraction and rules are generally dependent on filtering and sequence detectors. This natural dependency allow their composition into an event processing hierarchy where existing functionality can be reused.

2.2.4 Use grammar rules to combine plug-ins in a coherent way (pub/sub specific)

The application of the principles in 2.3.4 and 2.2.2 allow the use of language grammars to guide the composition of plug-ins. This approach not only enforces the proper format of subscriptions but also supports the automatic allocation of plug-ins in response to the subscriptions.

2.2.5 Consider using programming language interfaces for non-interactive extensions

While the use of XML has its advantages in the subscription and notification languages, allowing the development of different languages, and the interaction with end-users of the system, the publication and protocol models no not benefit from this approach. Events are usually produced by software entities (for example: peers, repositories and interface components in impromptu, and require a fast and direct way to capture and represent this information. Moreover, the matching of events with subscription usually require direct access to their content, which must be optimized. The parsing of XML documents is a very computational and memory-intensive task that should be avoided. The same is true for the protocol model which extensions are usually applied to federate event routers and interact with other infrastructures.

Based on those facts, we changed our original YANCEES design, with pure XML interaction, to a hybrid XML/API interface.

2.2.6 Pay special attention to runtime configuration

As discussed in 2.3.6, configuration management is important in SPLs. In the particular case of pub/sub domain, subscriptions are posted and removed at runtime, which may use different routing capabilities and infrastructure features. In a configurable environment, special attention must be given for invalid syntaxes and capabilities that are not available. When freedom is given to end-users to choose implementation strategies, as the case with YANCEES, special care is necessary.

YANCEES solves those issues by the use of runtime parsers, and dependency checking.

2.3 Software product line principles

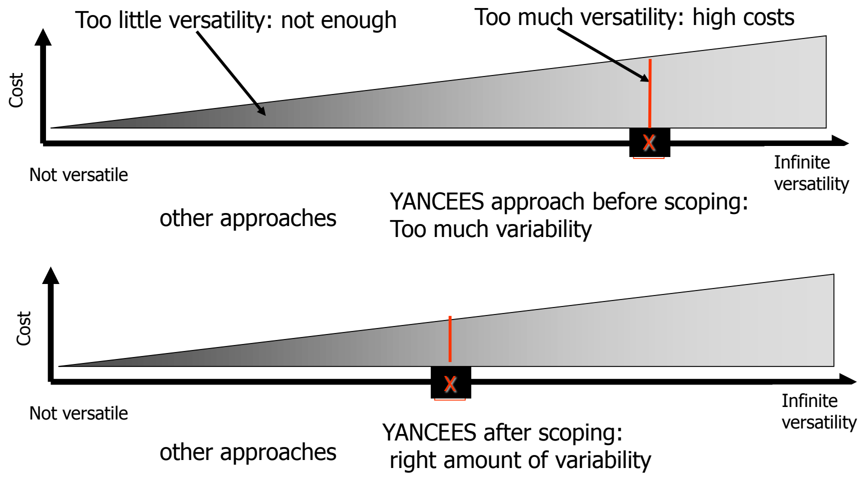
Those are principles, derived from our experience in designing YANCEES, that can be generalized to software product lines as a whole.

2.3.1 Scope to balance variability and reuse, limiting complexity and the impact of dependencies

Scoping is an essential activity to limit the complexity of the implementation and to achieve a reasonable amount of complexity that is cost-effective i.e. cheaper than building the system from scratch.

The broader the scope of a product line, more provision must be made for the negative aspects of variability, such as feature interference, and decrease in reuse of existing extensions. In particular, dependencies between extensions and configurations.

Scope's goal is to avoid over engineering and to guarantee a linear relation between the complexity of the extension and the steps needed to add that feature to the existing infrastructure. Over engineering not only increases complexity but also jeopardizes performance.



Too little versatility: not enough
Too much versatility: high costs
Cost
X
Not versatile
Infinite versatility
other approaches
YANCEES approach before scoping: Too much variability
Cost
X
Not versatile
Infinite versatility
other approaches
YANCEES after scoping: right amount of variability

Figure 1 YANCEES before and after scoping

2.3.2 Implement common core characteristics and layer out new functionality as necessary

In the publish/subscribe domain, the publication, routing (based on subscription) and notification of events represent a common behavior that can be represented in a micro kernel architecture. On top of this common code, extensions can be added to implement the specificities of each application domain problem.

The separation of core and extensions helps in keeping the core dependencies documented and constant throughout the different variants derived from the architecture and helps in the management of the complexity introduced by different extensions.

For example, the separation of filtering and event queue in YANCEES 0.8 strive to keep a simple core, where filtering functionality is plugged-in as necessary.

This principle is also a generalization of principle 2.2.3.

2.3.3 Combine different variability realization approaches, leveraging their benefits, preventing their weaknesses

Different variability realization approaches have strengths and drawbacks. The use of a single approach while may create a more consistent implementation, may not fit the different needs of the infrastructure variation points, increasing complexity (Silva Filho and Redmiles 2005).

For example:

YANCEES Uses extensible language and plug-ins, adapters, filters and configuration managers layered over a minimal core.

2.3.4 Modularize features in ways that allow their reuse in different contexts

Features may require extension in different variation points. The modularization of those components, allow their use in different contexts other than the features they were designed for.

Plug-ins are modular units representing the commands in the language. Those commands can be reused in different expressions.

In order to be successful, however, this modular reuse requires the documentation of the system-wide and environmental dependencies of these plug-ins

2.3.5 Make dependencies explicit

Documentation becomes more critical in low-coupling architectures, especially in software product lines. Many dependencies impact the complexity, variability and usability of a SPL (Fundamental, technological, configuration-specific and emerging properties). They must be appropriately documented and managed to prevent feature interaction (Silva Filho and Redmiles 2007).

In our prototype, those dependencies are provided in the code, through the use of annotations, in a global configuration file, where components are installed in different extension points, and in the subscription and notification languages in the form of grammatical plug-in composition rules.

2.3.6 Incorporate automated configuration management into the product line

One cannot adequately handle the combinatorial explosion of features, dependencies and extension mechanisms without automated support. Information about dependencies is the main source of information.

Configuration management has two aspects: runtime and load time. At load time, the required components that implement the specificities of a pub/sub infrastructure must be gathered and integrated. At runtime, those components must be used in a coherent way. In both cases, they require the analysis of dependencies.

2.3.7 Do not protect your software from changes but allow users to understand what changes are possible

Parnas' Information hiding principle (Parnas 1972) is expressed in the Open-Closed principle, which states that: software modules should be both "open (for extension and adaptation) and closed (to avoid modifications that affect clients)" (Larman 2001). This principle basically states that one must hide design decisions which are likely to change from other components in the system.

This principle can be harmful in SPL. We argue that users must know what dimensions are more prone to change in order to plan and design accordingly and prevent feature interaction. In special, our argument goes in line with the open implementation principle as proposed by Kiczales (Kiczales, Lamping et al. 1997).

2.3.8 Provide runtime context

In order to support the development of feature-rich plug-ins, one needs to provide the developers with enough runtime information. For example: subscriber interface, original subscription command, and other information that provides context, and allow developers to establish cause-effect relationships that can be used in their programs. That's the case with the subscription and notification plug-ins that need information such as the original subscription id, the original subscriber id, and the content of the message it represents. This additional information provides valuable information for the development of plug-in specific functionality.

2.3.9 Support architecture reflection

The dynamic and static configurability of the infrastructure, the implementation of a single feature by more than one component, creating the need of different components to interoperate, makes necessary the use of architectural reflection. For example, in YANCEES, the use of a plug-in registry allows the runtime discovery and binding of different plug-ins that implement the features of the infrastructure.

2.4 General software design principles

General design principles applied in the development of YANCEES that are broader than SPL and pub/sub middleware.

2.4.1 Allow users to control implementation strategies (Open implementation)

In order to properly operate the infrastructure, users must be aware of the resources installed in the infrastructure (subscription language), and should have ways to control the implementation strategies used in the system. The exposing of implementation concerns to end-uses and their ability to control the implementation parameters in the commands in the language is part of the open implementation principle (Kiczales, Lamping et al. 1997).

The system must also prevent end-users from performing illegal operations (for example, the posting of invalid or incoherent subscriptions). This is achieved in YANCEES by the use of grammars, and runtime parsers that enforce those rules in the implementation level.

2.4.2 Provide automated support for software engineers

The design for flexibility increases software complexity, making it difficult for software engineers not familiar with the infrastructure to extend the system for their purposes.

The automation of different aspects such as plug-in stub generation, runtime and load time configuration management and context annotations (that make dependencies explicit) help software engineers in adopting the infrastructure by supporting the basic configuration and extension activities.

2.4.3 Provide consistency by the injection of dependencies

One of the automatic supports in the infrastructure is provided by the contextual component framework, that enforces dependencies between components in the infrastructure.

Contextual Component Frameworks (Szyperski 2002) implement the automatic creation and composition of objects based on user-defined properties (or context). A component framework use the inversion of control (IoC) and injection of dependencies principle (Fowler 2004) to transparently provide user-requested services and properties to components.

2.5 Summary of design principles

The design principles document the main rationale in building YANCEES, providing examples where they were applied, supporting the hypothesis that a flexible pub/sub infrastructure can be build.

It also documents strategies used to overcome or minimize the effects of the problems we faced in the design and implementation of YANCEES. Those problems lead us to the design of the following experiment, in order to better understand the impact of variability in software complexity and usability.

References

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

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

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

Larman, C. (2001). Protected Variation: The Importance of Being Closed. IEEE Software. **18:** 89-91.

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

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

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

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

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