On the extensibility of plug-ins

Vanea Chiprianov, Yvon Kermarrec, Siegfried Rouvrais

On the Extensibility of Plug-ins

Vanea Chiprianov, Yvon Kermarrec
Institut Telecom, Telecom Bretagne
Université européenne de Bretagne
Technopole Brest Iroise, CS 83818 29238
Brest Cedex 3, France
UMR CNRS 3192 Lab-STICC
Vanea.Chiprianov@telecom-bretagne.eu

Siegfried Rouvrais
Institut Telecom, Telecom Bretagne
Université européenne de Bretagne
Technopole Brest Iroise, CS 83818 29238
Brest Cedex 3, France
Siegfried.Rouvrais@telecom-bretagne.eu

***Abstract*—There are software engineering tooling problems for which the solution benefits from being encapsulated as a plug-in. Among these problems, to ensure higher leverage, there are categories for which is important that their solution is extensible. However, extending a plug-in in practice often takes a long time, requires expertise, involves hacks and produces low quality code. In this paper, we advocate that assuring early in the design that a plug-in is extensible, by providing the necessary extension points, increases its re-usability, improves its evolution, and ultimately reduces the development time of the extender plug-in. We identify categories of software engineering problems whose solutions benefit from being extensible plug-ins, and review existing approaches to extending plug-ins. Finally, we report on our experience, with some of these approaches, in extending an Eclipse plug-in for a domain specific modeling language graphical editor.**

***Keywords*-Plug-in; extensibility; framework; software architecture; software design; design pattern; Domain Specific Language; modeling; experience report.**

I. Introduction

Our knowledge for solving software engineering problems is increasingly being encapsulated in tools. These tools provide the maximum of benefits when they operate in an environment that can provide integration with existing elements such as editors, compilers, debuggers, profilers and visualizers. A major challenge is to develop tools that can span different, heterogeneous and future environments.

A software plug-in is a set of software components that adds specific capabilities to a larger software application [1]. As an auxiliary ”client” module or expansion, it permits to add specific capabilities to a larger ”host” software application. For example, external capabilities may be functions, services, features, or support for handling a file format. The plug-in pattern, Figure 1, from [2], presents how to design an application in order to allow its extension at runtime by dynamically loaded modules or classes. The plug-in loader is part of what is called the framework.

Well-known examples of systems based on plug-ins include web-browsers (e.g., the add-ons [3] for Firefox), graphics editing programs [4], games (plug-ins are called mods [5]), integrated development environments (IDE) (e.g., Eclipse), tools for formal analysis and verification.

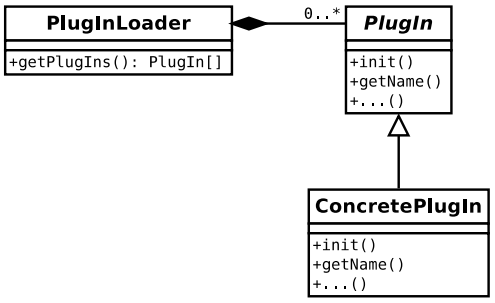

Figure 1. UML class diagram for the plug-in pattern, from [2].

Plug-in systems are developed in order to benefit from the following advantages [6]:

- Stability of system design. New features are added through plug-ins, independent of the core functionalities of the application.
- Reduced frequency of context switches. The user remains in the same integrated environment, experiencing a feeling of continuity.
- Increased usability. The user does not need to learn to use a new environment for the system functionality.
- Re-usability of framework functionality. Basic shared functionality is provided by the framework, so liberating the plug-ins from assuring it, reducing complexity and increasing modularity and understandability [2].
- High flexibility in tool customization. The user can select exactly the plug-ins tailored to her needs.
- Interoperability. In many research communities, all tools are developed using the same framework.
- Easy extensibility [7]. New tools can be added without the need to understand the framework code. Extendibility [8] is defined as the degree of usability and safety in contexts beyond those initially intended. Extendibility includes, but is not restricted to, extensibility.

Plug-in extension may be considered as a subproblem of the customizing libraries issue. Library customization con-

sists in adding new or modifying existing pieces of code. A recent comparative study [9] surveyed most of the techniques for library customization. Despite their differences, all techniques require some sort of hole/hook point/expansion in the ”host” code. The extensions have to ”hook” into a main application or framework environment [10]. For this, ”client” extensions must declare how they interact with the ”host” and the ”host” must provide interaction points. Because of this shared need of all extension approaches, we advocate that designers of the ”host” extended plug-in appear as good candidates to identify and provide these extension points, for the expansion approach of their choice.

However, plug-ins present drawbacks also:

- Difficult installation of new plug-ins. Compatibility issues with already present plug-ins, versioning problems, impede users and may even provoke reliability problems if the existing application stops.
- Restriction to the chosen framework. The framework may not support adequately all the necessary functionality and/or technologies (e.g., limitation to only a certain operating system).

To improve plug-in extensibility, and based on our experience report (Section IV), we defend in this paper that extension points should be included in the very initial design of certain plug-ins (Section II) software architecture in some manner (Section III). As such, developing plug-ins from scratch or refactoring them with those extension points in mind, opens the way towards easier plug-in interoperability, extensibility, integration, installation, increases openness to several frameworks, leading to more potential uses.

II. Categories of Extensible Plug-ins

It is quite safe to assume that plug-in providers would like to easily make their plug-in available in multiple frameworks. But experiments [11] on porting plug-ins to other frameworks suggest that only limited reuse is possible. Conceiving a plug-in as an extension of an adapter plug-in that takes care of framework particularities greatly increases reuse. So extension may be a great benefit or even a requirement for certain categories of plug-ins. In this section, we identify such categories of plug-ins.

One such category are tools for domain specific languages (DSLs). A DSL [12] is a language restricted to and focused on a particular domain. Implementing a DSL and its associated tools usually have as starting point an existing, more general purpose, base language and its tools. For example, SysML [13] (a modeling language for systems engineering applications) is defined as an extension of UML. Developing tools (e.g., editors, code generators) for such DSLs benefits from reusing base language tools. The base language tools are often part of a tool-set, an IDE, and implemented as plug-ins. Therefore, developing tools for DSLs based on another language often consists in extending plug-ins. The case study presented in Section IV is an example of an editor plug-in for such a DSL. Another example of plug-in extension for domain specificity is Ginga-NCL (Nested Context Language) [14], a declarative environment for IPTV services. An NCL application itself acts as a plug-in of another parent NCL application.

Tools for coverage, profiling and the collection of different kinds of runtime information are also particularly suitable for plug-in extension. Many of them are implemented as part of an IDE, as plug-ins, to assist the developer. Also, they may present several variants that share nonetheless common functionality, and so be suitable for extension. An example is InsECTJ [15], a framework which is also a plug-in. It is a set of Eclipse plug-ins for the collection of runtime information (coverage, profiling, and data values from specific points in a program execution). It defines a core plug-in, which implements the general framework and uses an extension point to expose its functionality to specific probe inserters. A probe inserter is an instrumentation module that implements the extension point in the core plug-in. Probe inserters are bundled in a second Eclipse plug-in.

In the Web development community, there are numerous web browser plug-ins that are extended at their turn. For example, Firebug [16] is a Firefox add-on for editing, debugging, and monitoring HTML, JavaScript and other Web languages. It has a number of extensions [17] that typically come in the form of Firefox add-ons. For example, Firebug Code Coverage is a Firefox add-on and Firebug extension that adds entry function code coverage for JavaScript code.

Another category consists of tools that implement different strategies (cf. the Strategy pattern [18]) as part of a toolchain. For example, the RDB2RDF Plugin [19] is an Eclipse plug-in that supports the standard relational database schema (RDB) to Resource Description Framework (RDF) Mapping Language (R2RML). R2RML mappings provide the ability to view existing relational data in the RDF data model, expressed in a structure and target vocabulary of the mapping author’s choice. New mapping algorithms can be added by the user through the implementation of an interface.

Plug-ins that are ported to other frameworks constitute another category for which plug-in extension is beneficial. To achieve more reuse when porting plug-ins to another platform, [11] propose to construct an adapter layer, written in a language supported by the framework, and conforming to the frameworks plug-in interface. The adapter in turn communicates with the plug-in through, for example, messages or remote procedure calls. We propose to construct the adapter as an extensible plug-in, so greatly simplifying the communication between the adapter and the plug-in. An adapter extensible plug-in will also reduce the *Restriction to the chosen framework* drawback of plug-ins (cf. Section I).

The need for extensible plug-ins is a real one, as shown by the numerous examples in different categories we have identified here. And even more categories should be identified. They can then be used by designers to decide if their

plug-in belongs to one of them. If it is the case, designers know there are high chances extensibility will be needed for their plug-in. So they can create an architecture accordingly.

III. Plug-in Extension Methods

To implement plug-in extension, common approaches, inspired from code generation techniques, are:

- Hacks. Though undesirable, many programmers use them. Hacks add code in many places, which reduces readability, maintenance, re-usability;
- C-style preprocessor directives. Simple #ifdef's can be used to include or exclude code, but readability is low and errors can be introduced easily;
- Object-Oriented Programming (OOP). Interfaces and (multiple)-inheritance provide variation/expansion points. They can be used at design time of the "host" plug-in to provide expansion points for the "client" code. Once the expansion points in the "host" are in place, the expansions consist of only adding new classes. This provides clear separation, facilitating readability, maintenance, extensibility. However, supplementary levels in inheritance hierarchies may result in loss of performance at run time;
- Feature-Oriented Programming [20]. A feature is an increment in program functionality. Feature interaction with core functionality and other features is defined in "lifters". At its essence, extending code this way is the same as OOP method overriding;
- Aspect-Oriented Programming [21]. An aspect is a supporting function, separated from the main logic. Aspects are added to main logic at various joint points. It allows adding functionality to an existing class transparently, which implies clean structuring of code. However, specifying point-cuts uniquely can be hard;
- Fragment-Oriented Program Generation. Pieces of code are combined to form a complete program. Pieces can be used by functions as regular input or output parameters. Composition of pieces is performed by plugging fragments into the holes declared in other fragments. Declaration of holes is needed.

Despite their differences, all these techniques require some sort of hole/hook point/expansion in the "host" code. Because of this shared need of all extension approaches, we advocate that designers of the "host" extended plug-in appear as good candidates to identify and provide these extension points, for the expansion approach of their choice. In this way, non-functional properties (e.g., re-usability, flexibility, extensibility) of the "host" plug-in are improved.

IV. Experience Report

In this section we report on our experience with extending a plug-in for Eclipse. Eclipse is an open source, extensible IDE, but also an extensible application framework upon which software, usually as plug-ins, can be built [10]. Using the OSGI framework to install, update or remove plug-ins on the fly, Eclipse can be easily customized. Moreover, it provides a mechanism to add features to a plug-in. In fact, there are numerous dependencies between plug-ins, some of them extending others. Eclipse allows building tools that integrate seamlessly with the environment and other tools.

A. Telecommunications Service Creation

As our research context is in the telecommunications area, we investigate domain specific models, meta-models and model transformations for service creation. We rely on a multi-layer, multi-view approach, as largely recognized in the Enterprise Architecture community. Recent efforts [22] [23] [24] of telecom operators (service providers) on defining meta-models for modeling services are indicative of the need for specific, dedicated modeling telecom languages and tools. Moreover, one of the service providers' requirements identified by [25] is to have an overall representation of service creation taking in all business, management, and technical activities. To meet these needs, we propose defining a graphical telecom DSL (with semantics implemented through code generation) as an extension of an Enterprise Architecture modeling language [26].

ArchiMate [27] is an Enterprise Architecture modeling language, a standard developed by the Open Group, with a large and growing user community. We propose defining our DSL as an ArchiMate extension. Archi [28] is a free, open source, cross-platform editor to create ArchiMate models. Archi is developed as a plug-in for Eclipse 3.6.1. To reuse existing tools for ArchiMate, we define a telecom DSL editor (Figure 2) as an extension of Archi. The editor presents the classical divisions of an Eclipse-based editor. At the left, there is the model navigator and an outline of the graphical model. The central window presents views (defined as tabs) of the graphical model. At the right, the palette offers the telecom specific concepts and relations, from which the designer can select, drag and drop the desired ones.

Our investigations and results for telecommunications [29] are out of the scope of this paper. Here we focus on tool design and development concerns [30] and especially report on our experience regarding the benefits of using extension points through three approaches presented hereafter.

B. Hacks

In the first phase, while still getting familiar with Archi's inner structure, we extended the editor by adding code in several methods from different classes. Adding a new concept in Archi means adding one class for the concept logic and two other classes for graphical purposes. Five other classes need editing. Three of these hacks are of the same type: adding a *case* statement in a *switch* instruction. Knowing all these distributed editing places requires in depth knowledge, which takes a significant time to acquire.

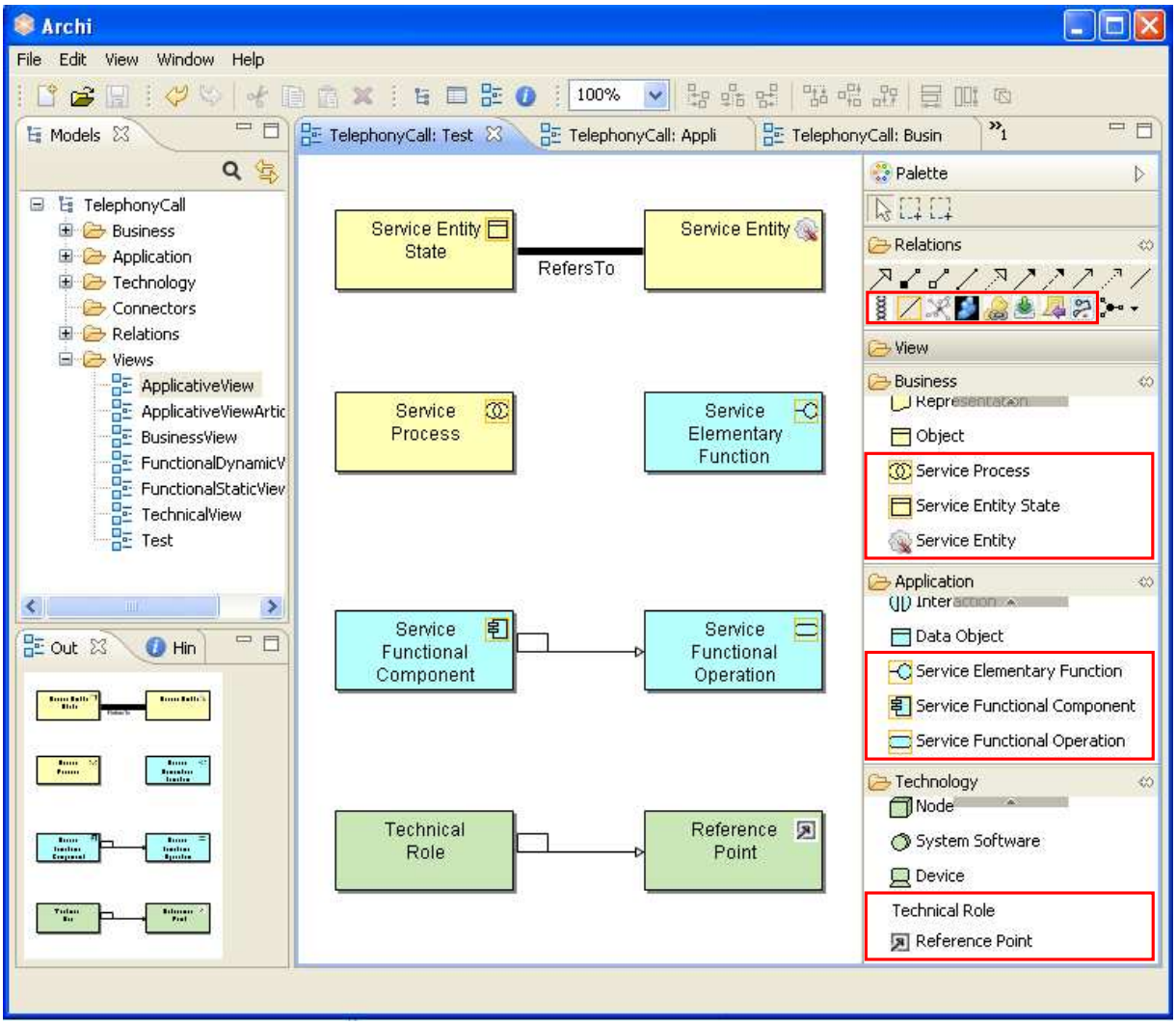

Figure 2. The Archi editor with the Telecom extension (showed in red boxes) in the palette.

After acquiring sufficient knowledge, we were able to propose a refactoring. The *switch* instruction may be replaced with an interface that is implemented by a class for each *case* statement (e.g., Strategy Design Pattern [18]). The advantage of this pattern is that code is added only in new classes, which impacts far less the existing code. However, refactoring takes a lot of time. An alternative solution would have been, for a good extensible plug-in, to have provided extension points from the original design.

C. Factory Design Pattern

Valid relations in Archi are listed in a hash-map. Adding a new relation implies updating this hash-map and the code that verifies if a particular relation has valid source and target entities. These verifications use conventions that are personal to the original developer (e.g., the pairs of valid source-target entities are coded as strings of letters). While the resulting code is small, its readability is very low. However, an alternative way of adding new relations consists in adding a new class that verifies the type of a relation: *ArchimateModelTelecomExtensionUtils*.

This new class implements an interface, also added by us: *IArchimateModelExtensionUtils*. The advantage is that when a new extension is desired, the relations introduced by this new extension are grouped in a new class which implements the interface *IArchimateModelExtensionUtils*. The instantiation of the class that implements this interface is chosen in another class, implementing in this way the Factory Design Pattern [18]. The disadvantage is that at any given time, only one extension can be activated.

D. Eclipse Extension Points

We may want to separate the newly added classes into a new plug-in. Eclipse offers the possibility, through ex-

tensions points, to define an ”extender” plug-in that adds functionality to a ”host” plug-in. However, it still requires the ”host” plug-in to call the extensions.

E. Lessons Learned and Insights

In all three methods presented for this case study, we had to provide expansion points in the ”host” extended plug-in. This implies detailed knowledge of the extended plug-in code, which takes a long time to acquire. That is why we advocate that the original designer of the ”host” plug-in should be the one that provides extension points. In this way, non-functional properties of the ”host” plug-in (e.g., re-usability, extensibility) are improved and the development time of the extender plug-in is reduced.

We also emphasize the iterative manner in which the extensions were written. In the discovery phase of the ”host” plug-in code, hacks were easier to use. When a more global understanding of the design was achieved, restructuring the design was envisaged and design patterns were employed. Finally, the added code could be separated in a new plug-in. We appreciate that a good extensible design for a ”host” plug-in should enable the ”client” plug-in developer to skip directly to what was, in our case, the third phase.

V. Related Work

ObjectTeams/Java [31] is an Aspect-Oriented Programming language which introduces the concepts of roles and decapsulation. A role class can declare a base class by which this role is played (using the keyword ”playedBy”). To visualize it, in Figure 1, consider ”Plugin” as the base class, ”ConcretePlugin” as the role class and replace the ”realize” relation between them with the ”playedBy” relation. A role class can adapt the behavior of its base class much like a sub-class, with the difference that roles are kept as separate entities at runtime. This results in base instances being kept intact, while roles can be added independently from each other. Of course, to have access to private data as sub-classes do, roles need decapsulation. Decapsulation means that a role class may access features of its base class even if the normal rules of encapsulation would prohibit it. Support for ObjectTeams has been added in the Eclipse OSGI framework. This generalizes the Eclipse plug-in extension mechanism with the introduction of joint points which can bee seen as unanticipated extension points. This makes it possible not only to add new behavior, but also to replace functionality of a plug-in. However, the introduction of decapsulation breaks one of the most important principle and advantage of the Object Oriented Paradigm. Programmers of the extending plug-in become encumbered with the responsibility of correctly using the decapsulated data, which goes back to the need of knowing in great detail the implementation of the extended plug-in. To counter negative effects like this, we propose that the original architect of the plug-in provides the necessary extension points.

Other proposals that enable reuse of plug-ins, although not through extensibility, include, for example, [32]. The authors propose the concepts of Task Based plug-in and work-flow of Task Based plug-ins. A Task Based plug-in is a plug-in that declares the functionalities that can be executed as tasks. Using tasks, IDE users can create work-flows that execute multiple tools and integrate tool results. In this way, Task Based plug-ins can be integrated and composed through pre-defined and user-defined task flows. However, this reuse approach does not allow specialization of the behavior of a tool, which extensibility does.

Another framework for service development, developed as an Eclipse plug-in, is jABC [33]. It also contains an extensible set of plug-ins, so that the jABC models can be analyzed, simulated, verified, executed and compiled. However, it deals with services in general, while the plug-in we developed is focused on Telecommunications services.

VI. Discussion and Conclusions

The importance of plug-in extensibility is intrinsically part of the bigger discussion on software architecture (good properties). It may be argued that extensibility, and even the existence of a software architecture, plans too much in advance, pushes too much on the anticipation side. This may lead to BUFD [34] (Big Up-Front Design), massive documentation, smell of waterfall, implementing features YAGNI [35] (You Ain’t Gonna Need It), huge future re-factorings because of architecture erosion. An alternative is that a metaphor should suffice, the architecture should emerge gradually sprint after sprint, as a result of a succession of small re-factorings, through an adaptive process. However, certain classes of systems, ignoring architectural issues too long, ”hit a wall” and collapse by lack of an architectural focus [36]. So, there are categories of systems for which an adaptive approach may prove more appropriate (e.g., small web-based socio-technical systems) or, conversely, categories of systems for which an anticipative approach may prove more beneficial.

In this paper we addressed problems related to plug-in extension. To reduce the development time of the extender plug-in and increase quality properties (e.g., extensibility, re-usability, flexibility) of the ”host” plug-in, we advocated including extension points from the start, in the original design of the ”host” plug-in. We have illustrated issues and investigated solutions for the case of extending an Eclipse plug-in for a domain specific modeling language graphical editor. Through this, we hope to raise awareness among plug-in designers for domains which are highly probable to make use of plug-in extension (e.g., domain specific languages). In the future, a full comparative study of extension methods will be useful in pinpointing limitations from which current plug-in development systems may suffer and help correct them.

VII. Acknowledgments

The authors would like to thank Sébastien Bigaret, Telecom Bretagne, for his helpful reviews and Phil Beauvoir, JISC CETIS, University of Bolton, for his helpful explanations, examples and suggestions.

References

[1] Wikipedia. (2011) Plug-in (computing). Accessed on 25.07.2011. [Online]. Available: http://en.wikipedia.org/wiki/Plug-in_%28computing%29

[2] J. Mayer, I. Melzer, and F. Schweiggert, "Lightweight plug-in-based application development," in *Intl Conf. NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, London, UK, 2003, pp. 87–102.

[3] Firefox. (2011) Add-ons. Accessed on 25.07.2011. [Online]. Available: https://addons.mozilla.org/en-US/firefox/

[4] Photoshop. (2011) The plugin site. Accessed on 25.07.2011. [Online]. Available: http://thepluginsite.com/knowhow/tutorials/introduction/introduction.htm

[5] Civfanatics. (2011) Customizing Civilization IV. Accessed on 25.07.2011. [Online]. Available: http://www.civfanatics.com/civ4/downloads

[6] S. Wagner, S. Winkler, E. Pitzer, G. Kronberger, A. Beham, R. Braune, and M. Affenzeller, "Benefits of plugin-based heuristic optimization software systems," in *Proc. of the 11th intl conf. on Computer aided systems theory*, Las Palmas de Gran Canaria, Spain, 2007, pp. 747–754.

[7] F. N. Paulisch, *The Design of an Extendible Graph Editor*. Secaucus, NJ, USA: Springer-Verlag, 1993.

[8] JTC1/SC7/WG6, *ISO/IEC CD 25010.3: Systems and software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Quality models for software product quality and system quality in use. Version 1.46*, Std., 2009.

[9] B. Aktemur and S. Kamin, "A comparative study of techniques to write customizable libraries," in *ACM Symposium on Applied Computing*, Hawaii, USA, 2009, pp. 522–529.

[10] D. Rubel, "The Heart of Eclipse," *Queue*, vol. 4, pp. 36–44, 2006.

[11] N. Sawadsky and G. C. Murphy, "Fishtail: From Task Context to Source Code Examples," in *[37]*, 2011.

[12] A. V. Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, 2000.

[13] OMG, *Systems Modeling Language, Version 1.2*, Std., 2010.

[14] M. F. Moreno, R. S. Marinho, and L. F. Gomes Soares, "Ginga-NCL Architecture for Plug-ins," in *[37]*, 2011.

[15] A. Seesing and A. Orso, "InsECTJ: a generic instrumentation framework for collecting dynamic information within Eclipse," in *Proc. of the eclipse Technology eXchange (eTX) Ws at OOPSLA*, San Diego, CA, USA, 2005, pp. 49–53.

[16] Firefox. (2011) Firebug. Accessed on 25.07.2011. [Online]. Available: https://addons.mozilla.org/en-US/firefox/addon/firebug/

[17] Getfirebug. (2011) Firebug/Extensions. Accessed on 25.07.2011. [Online]. Available: http://getfirebug.com/wiki/index.php/Firebug_Extensions

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[19] P. E. Salas, E. Marx, A. Mera, and J. Viterbo, "RDB2RDF Plugin: Relational Databases to RDF Plugin for Eclipse," in *[37]*, 2011.

[20] C. Prehofer, "Feature-oriented programming: A fresh look at objects," *ECOOP*, vol. 1241, pp. 419–443, 1997.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming," *ECOOP*, vol. 1241, pp. 220–242, 1997.

[22] E. Bertin, S. Bécot, and R. Nedelec, "Applying Enterprise Architecture Principles to Telco Services," in *14th Intl Conf. on Intelligence in Next Generation Networks*, 2010, pp. 1–6.

[23] A. Ahuja, J. Simonin, and R. Nedelec, "MDA tool for telecom service functional design," in *Proc. of the 4th Euro conf. on Software architecture*, Copenhagen, Dk, 2010, pp. 519–522.

[24] J. Simonin, E. Bertin, Y. L. Traon, J.-M. Jezequel, and N. Crespi, "Business and information system alignment: A formal solution for telecom services," *Intl Conf. on Software Engineering Advances (ICSEA)*, pp. 278–283, 2010.

[25] J. Hållstrand and D. Martin, "Industrial requirements on a service creation environment," in *Proc. of the 2nd Intl Conf. on Intelligence in Broadband Services and Networks: Towards a Pan-European Telecommunication Service Infrastructure*, Aachen, Germany, 1994, pp. 17–25.

[26] V. Chiprianov, I. Alloush, Y. Kermarrec, and S. Rouvrais, "Telecommunications Service Creation: Towards Extensions for Enterprise Architecture Modeling Languages," in *6th Intl Conf. on Software and Data Technologies (ICSOFT)*, Seville, Spain, 2011, pp. 23–28.

[27] Open Group, *Archimate 1.0 specification*, Std., 2009.

[28] Cetis. (2011) Archi. Accessed on 25.07.2011. [Online]. Available: http://archi.cetis.ac.uk/

[29] V. Chiprianov, Y. Kermarrec, and P. D. Alff, "A Model-Driven Approach for Telecommunications Network Services Definition," in *Eunice'09: Proc. of the 15th Open European Summer School and IFIP TC6. 6 Ws on The Internet of the Future*, vol. 5733, Barcelona, Spain, 2009, pp. 199–207.

[30] V. Chiprianov, Y. Kermarrec, and S. Rouvrais, "Meta-tools for Software Language Engineering: A Flexible Collaborative Modeling Language for Efficient Telecommunications Service Design," in *FlexiTools Ws at the 32nd ACM/IEEE Intl Conf. on Software Engineering (ICSE)*, Cape Town, South Africa, 2010.

[31] S. Herrmann, C. Hundt, and C. Pfeiffer, "Eclipse plugin adaptation with Equinox and ObjectTeams/Java," in *Eclipse Technology eXchange Ws at ECOOP*, Nantes, France, 2006.

[32] L. Mariani and F. Pastore, "Supporting Plug-in Mashes to Ease Tool Integration," in *[37]*, 2011.

[33] B. Steffen, T. Margaria, R. Nagel, S. Jörges, and C. Kubczak, "Model-driven development with the jABC," in *Proc. of the 2nd intl Haifa verification conf. on hardware and software, verification and testing*, Haifa, Israel, 2007, pp. 92–108.

[34] Wikipedia. (2011) Big Design Up Front. Accessed on 25.07.2011. [Online]. Available: http://en.wikipedia.org/wiki/Big_Design_Up_Front

[35] R. E. Jeffries. (2011) Youre NOT gonna need it! Accessed on 25.07.2011. [Online]. Available: http://www.xprogramming.com/Practices/PracNotNeed.html

[36] P. Abrahamsson, M. A. Babar, and P. Kruchten, "Agility and Architecture: Can They Coexist?" *IEEE Softw.*, vol. 27, pp. 16–22, 2010.

[37] J. Bishop, K. Breitman, and D. Notkin, in *1st Ws on Developing Tools as Plug-ins (TOPI) at the 33rd Intl Conf. on Software Engineering (ICSE)*, vol. (in press), Hawaii, USA, 2011.