Product Family Approach for Integration of In-house Software and Open Source Components

Janne Merilinna and Mari Matinlassi
VTT Technical Research Centre of Finland
P.O. Box 1100, 90571 Oulu, Finland
[Janne.Merilinna, Mari.Matinlassi]@vtt.fi

1. Introduction

Integrating software components - either components off-the-shelf or open source components - is a quite common practice in software development. This way, the role of the software developer has changed into that of an *integrator* who obtains components and integrates them to form a new product. Especially, using open source (OS) components in software development is expected to provide a lot of benefits such as faster time-to-market and cost-savings [1]. Especially in the realm of open source software (OSS) – depending on licensing[1] – cost-savings are evident and it has also been observed to provide a high quality of software [2].

While several advantages can be gained by employing OSS, integrators seem to face several kinds of integration problems [3]. The most common problems are related to vertical integration, i.e. platform incompatibility. Horizontal integration problems such as interface incompatibility are another common problem area; these horizontal problems may be solved with suitable integration architecture, i.e. interface adapters and component encapsulation. However, solving horizontal integration issues may be far from straightforward due to incompatible interaction policies [4]. This paper especially concentrates on horizontal integration problems of open source components and in-house software. Open source component (OSC) again denotes a component that is:

- available with its source code and
- under an open source license to study, change, and improve its design

One of the key tenets of open source software development is "release small, release often". Especially in immature components, frequent releases may cause laborious changes in integration architecture.

In addition to high frequency of changes, other special features of open source components are - obviously - the availability of source code and a possibility to participate in component development but also the lack of documentation.

Numerous studies have been made in the field to software component integration and integration architectures, e.g. [5] [6] [7], [8]. However, previous studies do not especially address the integrability of in-house software and OSCs. Addressing integrability of in-house software and OSCs would presume one to consider the special characteristics of open source software integration mentioned above.

This position paper claims that applying product family approach - see e.g. [9] - in developing open source component integration code i.e. glue code or glueware, would improve integrability of OSC by utilizing the benefits and even more important, compensating the drawbacks of OSS development. Furthermore, approach would benefit both integrators and open source communities.

2. Problem context

From the point of view of open source component integrators OSS development provides at least the following special characteristics and problems.

1. An opportunity to influence the development of OSCs. This is a unique and worthwhile opportunity and shall be grasped into. However, component integrators (e.g. SMEs) often experience contributing to communities too laborious [10]. Therefore, this paper would like to focus on solving problems such as:
   - How to lessen the effort to contribute your integration code to community?
   - How to affect integrability of OSC without being an active member?
2. The lack of documentation makes the components more difficult to integrate. This is one unavoidable and negative side related to the use of third-party components and, especially present problem in open source component integration.

[1] http://www.opensource.org/licenses/index.php

Thus, this paper would also like to focus on questions below:

- How to affect integrability of OSC without having extensive knowledge about the OSC?
- How to provide a bit better understanding of potential interaction mismatches?

3. A high frequency of changes distributed over the whole component life-cycle. Change management and maintainability are everlasting problem areas in software development. In open source component integration context, this paper would like to aim at answering the following problems.
   - How to share maintenance responsibility of integration gluecode?
   - How to enable reusability of gluecode?
   - How to implement the gluecode in a way that enables creating different variants of the component?

3. State of the Practice in In-house Software and OSC Integration

As far as our experience and other previous studies state [3, 10], current state of practice in OSC integration quite often follows the path illustrated in Figure 1. Here, we have two components communicating via *glueware*. Glueware is software that does not compute anything functional towards meeting the program's requirements, but instead serves solely to "glue together" different parts of code that would otherwise not be compatible. Part of the required *glueware* is implemented in the OSC (large rectangle surrounding OSC), i.e. OSC itself implements part of the features required from the integration architecture, while the rest is implemented within the in-house system (small rectangle between the components).

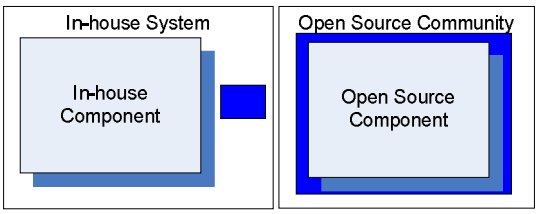

Figure 1. State of practice OSC integration.

Implementing the *glueware* as a separate part that is maintained by self is troublesome when the OSCs evolve. This is due to the fact that the *glueware* has to be adapted to the changed situation. This is problematic especially if the *glueware* is implemented as a monolithic block at least for four reasons:

- Contributing glueware to open source community would relief maintenance effort however, contributing monolithic glueware block to open source community is not feasible because block probably is not reusable for other users.
- Not all the features that the complex integration architecture provides are likely to be required by all the in-house components using OSC. For instance, synchronizing and buffering may not be required if there are no concurrent invocations. That is, different in-house components may require a different *variant* of the OS component.
- Components evolving rapidly require integration architecture that can easily be adapted to the changed situation. If the *glueware* is not designed for modifiability, keeping it up-to-date may be laborious.

When a part of the *glueware* is implemented *within the OSC*, there is also a problem whoever is maintaining the *glueware*: different components may require different *variants* of the OSC. This is the same problem as in implementing glueware as a separate monolithic part. Thus, there is no use to construct extensive integration architecture inside the OSC as although the component provider strives to please everyone, there will always be users that require different *variants* of the components.

Finally, the user of the component may also not be aware of exactly what kind of gluing solutions, e.g. synchronization, buffering, sequencing etc., are already implemented in the component. Although even the source code would be available, it is not always that simple to thoroughly understand the component, especially if the solutions that promote integrability are spread within several sub components. This creates a fundamental problem for integration – misassumptions about components [11].

4. Proposing Product Family Solution

The integration product family architecture is depicted in Figure 2 with small rectangles between the in-house components and the OSC. *Glueware* is now constructed from small parts and, it is (at least partly) contributed to the OS community. We may call this open *glueware* as *openware*. When integrator contributes glueware parts to open source community, the integrator becomes part of the developer community and therefore maintenance responsibility is only shared, not completely outsourced.

The key idea is that the OS community (including integrators who have contributed to development) provides a set of reusable *integration elements*, e.g. buffers, synchronizers, translators etc. that can be

applied as part of the complete integration architecture between the in-house component and the OSC. The user of the OSC can then combine part of the required integration architecture from provided *integration elements* and then complete the integration architecture by implementing the rest of the *integration elements* by self. By doing so, the user of the component can *configure* the OSC to be integrated with in-house system components by combining separate *integration elements* into complete integration architecture. Combination of the required integration elements is discovered by scrutinizing the problematic architecture interactions between components as described in [12].

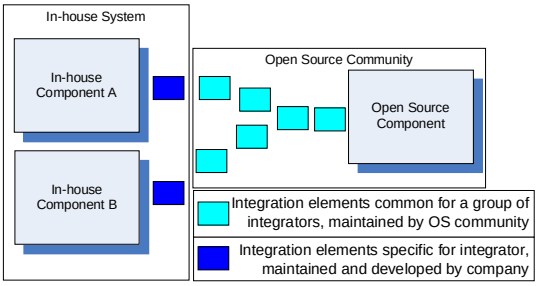

Figure 2. OSC integration with product family approach.

Openware technique also reduces the primary reasons integration problems occur, i.e. misassumptions about the components. The OSC itself contains only the main logic and the interaction logic is externalized into *integration elements*. Therefore, overlapping or unnecessary *glueware* between components is also reduced.

This technique is also beneficial for the community at least for the following reasons:

- integrability of the component will be increased,
- increased integrability and usability of the component may lure more users,
- more users means more developers, and
- therefore quality and attractiveness of the component is increased.

5. Segregator Product Family Micro Architecture for OSC Integration

The product family approach for the integration architecture can be implemented with a well-known design pattern, Decorator [13], when the intent of it is slightly broadened and structure modified. Decorator alone enables segregating *glueware* into discrete partial solutions that, as a whole, implement the complex integration architecture. Furthermore, the partial solutions can be used for constructing different *variants* of the components by stacking different kinds of *Decorators*. However, Decorator does not allow constructing any other than pipe-like integration architectures because the Decorator is not allowed to have multiple *Components*[2]. In addition, Decorator is originally not intended to function as any kind of *controller* [5]. In addition to these, implementing Decorator as it is causes the component to be integrated to inherit *Component*.

Thus, Decorator needs to be enhanced first by letting *Decorator* to refer to multiple *Components*, therefore allowing multiple output ports instead of merely one. By doing so, Decorator embodies Façade design pattern [13]. Multiple input ports can be provided by encapsulating each request as an object containing some identification information about the client that starts the transaction. This enables integration architecture to prioritize, schedule, filter the data for instance, i.e. playing a role of *controller*. In addition to these changes, component to be integrated is completely separated from the integration architecture. From now on, this enhanced Decorator pattern is called a Segregator (product family) micro architecture.

Figure 3 depicts the class diagram of the Segregator micro architecture. As it can be seen, the Segregator has somewhat the same structure as the Decorator. However, there are also some differences. Participants of the Segregator are as follows:

- ***Component*** defines the interface for objects participating in the integration architecture. The interface must enable accessing all required functions in *ConcreteComponent*.
- ***Segregator*** maintains reference to *Component* and conforms to its interface. *Segregator* can reference to multiple *Components*. *Segregator* functions as *extender*, *translator* or *controller*.
- ***ConcreteSegregator*** is an implementation of the *Segregator*.
- ***LeafSegregator*** functions as *extender*, *translator* or *controller*. It maintains reference to the *ConcreteComponent* and is the last integration element in the integration architecture.

Collaboration of the components in the micro architecture is as follows. The *Segregator* forwards requests to its *Component*. It may add additional behaviour before or after forwarding. A component being a *Segregator*, the request is forwarded again until a *LeafSegregator* is reached. The *LeafSegregator*

[2] Component is a participant in Decorator design pattern. See description below.

then invocates *ConcreteComponent*, which then implements the request.

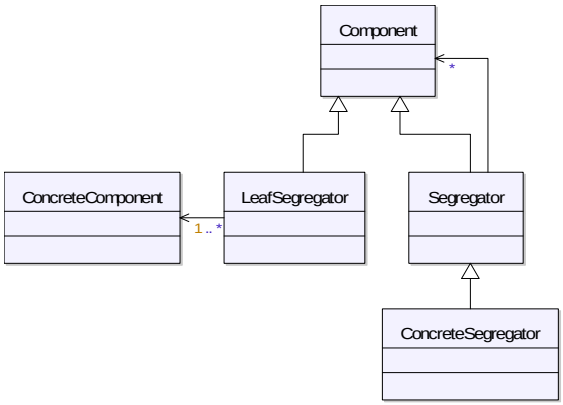

Figure 3. Class diagram of the Segregator micro architecture.

6. Conclusion

In this paper, we propose using product family architecture approach in open source component integration. Implementation of product family architecture is performed by using a slightly refined Decorator pattern, which we call Segregator micro architecture.

To sum it up, it is now possible to build integration architecture from multiple task-specific parts that can be stacked to form a variety of different *variants* of the component thus enabling easy configurability. By implementing integration architecture in this product family oriented way, it is possible for open source component providers to provide a separate list of *integration elements*, i.e. *openware*, that can be applied by the component user for configuring the integration architecture in a required way. Glueware implemented with product family approach also lessens integrators' effort to contribute their integration code into communities by following a predefined product family micro architecture. Integrators do not need any more to be active members in open source community or have extensive knowledge about the whole source code in order to affect integrability of OSCs.

Furthermore, task-specific integration code parts are reusable and maintenance responsibility may be shared with the rest of the community.

7. Acknowledgements

The publication of this paper has been supported by the Eureka ITEA research project COSI, funded by the National Technology Agency (Tekes) and VTT.

8. References

[1] B. C. Meyers, *Managing Software Acquisition: Open Systems and COTS Products.* Boston: Addison-Wesley, 2001, pp. 360.

[2] I. Samoladas, I. Stamelos, L. Angelis and A. Oikonomou. (2004, October). Open source software development should strive for even greater code maintainability. *Commun ACM 47(10),* pp. 83-87.

[3] J. Merilinna and M. Matinlassi. State of the art and practice of open source component integration. Presented at Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA).

[4] T. Trew and G. Soepenberg, "Identifying technical risks in third-party software for embedded products," in *5th International Conference on COTS-Based Software Systems (ICCBSS 2006),* 2006, 10 p.

[5] R. Keshav. 1999. Architecture integration elements: Connectors that form middleware.

[6] J. Davis, R. Gamble and R. F. Payton. (2002, The impact of component architectures on interoperability. *The Journal of Systems and Software 61*pp. 31-45.

[7] L. Davis and R. F. Gamble, "Identifying evolvability for integration," in *Proceedings of the International Conference on COTS-Based Software Systems,* 2002,

[8] L. Davis, R. F. Gamble and S. Kimsen. (2004, 02). A patterned approach for linking knowledge-based systems to external resources. *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics) 34*pp. 222-33.

[9] K. Pohl, G. Böckle and F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005, 467p.

[10] B. Lundell, B. Lings and E. Lindqvist, "Perceptions and uptake of open source in swedish organisations," in T*he Second Internatinoal Conference on Open Source Systems,* 2006, pp. 151-163.

[11] D. Garlan, R. Allen and J. Ockerbloom. 1995. Architectural mismatch: Why reuse is so hard. *IEEE Software 12,* pp. 17-26.

[12] G. Jonsdottir, L. Davis, R. Gamble, J. Payton and D. Underwood. A notation for problematic architecture interactions. Presented at 8th European Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9).

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994, pp. 416.