A look at Programming Methods for solving problems of current Software Development

Kenneth Rohde Christiansen, Niek Oost

Department of Mathematics and Computing Science
Rijksuniversiteit Groningen
Blauwborgje 3
NL-9747 AC Groningen

{k.r.christiansen, h.oost}@student.rug.nl

Abstract
In this paper we will look at the problems of application development that are not directly solved by using standard programming methods like imperative, object-oriented or functional programming. We will introduce four proposals for alternative programming methods developed to solve some of these problems. This text evaluates the different proposals on practical applicability of these methods in the near future. The document assumes that the reader has basic knowledge of object-oriented programming.

Keywords: Programming Methods, Subject-Oriented Programming, Aspect-Oriented Programming, Separation of Concern, Computer Science.

Version of January 9th 2005; revised form of December 10th 2004.

1 Introduction

The main objective of software engineering is to improve software quality, reduce costs and also to facilitate easy maintenance of the software product. Over the last few years many new techniques have been introduced to support this. High-level languages such as C proved to be a major step forward from programming in assembly and today most people value the use of Object-Oriented languages with garbage collectors such as Java or C#. Unfortunately, Object-Oriented languages do not solve all of the software development problems.

The main objective of this paper is to look into the various problems of application development that are not solved by common programming languages in use today. The biggest problems of software development relates to code understanding, maintenance, extendibility and reuse. Object-Oriented Programming has been suggested to solve some of these problems by decomposing the code into small reusable objects, and has proven highly successful. Unfortunately, artifacts as for instance new features often decompose the product in a different way, so the different decomposition methods tend to scatter or tangle the code. Adding a new feature might require changes to multiple objects, this will make it harder to understand and maintain the code. The different ways a product can be decomposed are often referred to as the separations of concern. Each concern leads to a different kind of separation/decomposition.

1.1 Discussed techniques

In this text the following proposed techniques are discussed :

Subject-Oriented Programming [Harrison and Ossher, 1993] tries decomposing the code into subjects that deal with the same objects, separating the extrinsic properties from the actual objects and their intrinsic properties.

Aspect-Oriented Programming [Kiczales..., June 1997] looks at the non-functional concerns to be separated from the functional concerns.

Hyper modules and -slices are introduced in [Harrison..., 1999] as a general way to satisfy the separation of concern for functional dependencies.

Superimposition [Bosch, 1998] proposes a technique for component adaptation for reusable components.

2 Subject-Oriented Programming

Object-Oriented programming tries modeling the world as objects. Trees, birds, cars, etc. are all examples of objects. This has proven to be a successful approach and has revolutionized the software industry. Unfortunately, different subjects consider different properties of an object as important. Where a tree has intrinsic properties like height and age, a subject can also be interested in other properties. A bird, for instance is also interested in the possibilities to build a nest in the tree.

In the software developing community a growing suite of applications manipulate the same objects. During the development of these objects, programmers need to anticipate which properties that could be used in future applications, so that it is easier to reuse their code. This is an almost impossible task, since it is almost impossible to know what extrinsic values will be used in future applications. An extensive interface on the other hand also creates problems for programmers using these objects: different applications will only use part of the object interface, while they are forced to also consider the rest of the interface (with possibly state modifiers).

If object-oriented programming is to scale from the development of independent applications to the development of highly integrated application suites it has to relax the dependency of objects and instead concentrate more on how these objects are tied together. A technique that emphasizes on the subject view is known under the name Subject-Oriented Programming and is described in [Harrison and Ossher, 1993]

2.1 The idea

Subject-Oriented Programming builds on Object-Oriented programming, in that there exist objects containing intrinsic properties (often known as fields) and behavior (often known as methods). Each subject contains a library of the classes of objects known to the subject. The subject additionally contains extrinsic properties for each of these objects reflecting how the subject sees the given object.

By following this idea it is possible to develop subjects independent of other subjects, while they can still deal with the same objects. An application is then considered as a subject or a composition of subjects.

2.2 Subject composition

When a subject interacts with an object it might influence the objects' intrinsic properties, but this might, in effect, also influence the extrinsic properties of different other subjects. If, for instance, a bird subject builds a nest in a tree this might mean that the woodman is not allowed to cut down the tree, thus the `can-cut` property owned by the woodman has to be changed.

To make a subjects' action on an object affect the extrinsic properties of another subject, the subjects need to be composed. This can only work if a subject implements the same behavior/methods as the behavior of the subject that it wants to respond to. If the woodman wants to track the building of nests in a tree, which are controlled by the `build-nest` and `abandon-nest` behavior of the bird subject, the woodman subject will have to implement these as well.

2.3 Class matching

The subjects do not necessarily have to include the same class library. For instance, a pine might be derived from `object->nestable` in the bird subject or it can be derived from `object->tree->softwood` in the woodman subject. This means that the composition rule also needs to specify some kind of class matching. Additionally, some subjects might know objects that other subjects do not know about. For instance the bird might know about a tree (a locust for instance) that the woodman does not know about. The composition rule can then be used for concluding that a locust can be treated by the woodman as being a tree.

2.4 Subject activation

When a subject gets instantiated - or activated - to follow the terminology in the paper, all objects in the subject have to be created as well as the extra extrinsic properties reflecting the subjects view of the objects. In a subject composition there is more than one subject dealing with the same objects. This means that the other subjects need to instantiate their extrinsic properties for the objects as well. This can be done in different ways, for instance all extrinsic properties for all subjects will be instantiated when one of the subjects is activated. A better and more effective way may be to first instantiate these properties when needed.

2.5 Our Conclusion

Subject-Oriented Programming introduces some nice concepts for modeling the interaction of subjects in the real world. Whether this would work for modeling applications is a good question. We think it sounds very complicated to design - maybe huge - different class libraries for each subject. The idea is that these can be designed independently, but for class matching to work properly the developer probably needs to know about the class libraries in the other subjects. Another problem that we see, is the problem of saving/serializing objects. Lets consider a document to be an object and a word processor and a spreadsheet to be subjects implementing extrinsic properties for this document. If we now want to save the document the question is how we make sure that all extrinsic properties for all subjects are saved. We do not see an easy way to do this with the current model.

3 Aspect-Oriented Programming

When code has to be optimized (for example for performance), these optimizations often cross-cut the different components that the application consists of. During the optimization of a piece of code, one often reduces the number of procedure calls, creating code that is much harder to understand and maintain, than the original code, but faster. To get the best of two worlds : easy maintainable code and highly optimized software, aspect oriented programming is proposed. In this technique, 'aspects' are introduced as issues cross-cutting components, dealing with things like performance and memory usage.

The paper [Kiczales..., June 1997] uses a good example to show what Aspect-Oriented Programming is all about. When dealing with images, like for instance developing an OCR application, you often have to put the image through various filters. Each of these filters produces a new image which might live shortly before it is copied to a second filter, discarding the copy of the previous filter. Since this results in excessive memory references, which can result in cache misses, page faults, etc. it should be avoided. In this particular case developers often try to merge these different filters into one big procedure, reusing as much memory as possible. This merge is done by hand, and often results in buggy, tangled code that is hard to understand, hard to extend and hard to maintain. The authors of the paper implemented an ineffective subpart of an OCR application in as little as 768 lines of code. The effective, tangled version on the other hand consisted of 35213 lines of hard maintainable code.

3.1 How it works

In the example described earlier, the optimization is done by merging filters, which is in fact done by merging loops. Aspect-Oriented Programming can do this automatically, and does it by introducing a component language, an aspect language and an aspect weaver. It is also noted that the actual weaving can be done at runtime or at compile-time.

The idea is that you write your components in a language similar to what you are used to, but for the above example, the language will have to support a high-level looping construct so that the weaver can detect, analyze and fuse loops more easily. In the paper they introduce a `pixelwise` construct to do this.

Additionally, you will have to write a merging condition in the aspect language, which will then be used to compare nodes in the data flow of the application. If two of the filters fulfill the condition, like for instance that they are both pixel wise, then they can be merged. This exact merging is done by the weaver.

The weaver uses unfolding to generate the data flow graph, which then is processed by the aspect rules which will look for nodes to merge. As the last step the code generated walks though the data flow graph and generates the actual code.

With this method the authors implemented an easy maintainable version of the OCR subroutine, that was almost as effective as the tangled version and in as little as 1039 lines of code. With an improved code generator it should be possible to make the new version practically as effective as the tangled version.

3.2 Our Conclusion

There are many different kinds of aspects, that cannot cleanly be encapsulated in a generalized component. Loop fusion is just one of these. Others include, minimizing network traffic, synchronization constrains, error handling etc.

We find Aspect-Oriented programming to be an interesting new idea that can actually easily solve some of the problems that developers are dealing with today. Unfortunately, Aspect-Oriented programming is a very young idea and there has to be researched how it can help solving other aspects than loop-fusion, and what are good structures for use in aspect programs. There is also the question if people can easily learn to analyze and identify aspects in their programs, and if it can be made easy to debug these applications when the generated code will be very different from the actual implementation.

4 Hyper modules and -slices

Hyper modules and hyper slices is a recent idea for solving the separation of concern. It builds on Subject-Oriented Programming and the people behind Subject-Oriented Programming are also coauthors of this paper, [Harrison..., 1999].

The idea roots in Object-Oriented programming, and facilitated methods for easily adding new features that span multiple classes. A hyper slice resembles a subject in the Subject-Oriented method. The difference is that a hyper slice only implements extra 'extrinsic' properties and methods for the classes in the program. This means that we do not have the class matching problems as in the Subject-Oriented idea. This makes it possible to easily add new features without changing and polluting the existing classes with implementation details of this feature. This way the code for the feature addition will be kept together and code readability and maintainability will be maintained.

As with subjects, hyper slices can be combined by a composition rule. A composition of hyper slices is called a hyperplane, and it contains approximately the same problems as we noted in the section about Subject-Oriented Programming. There are fewer problems, as the slices build on top of an already defined class library. Because the slices do not implement different class libraries, class matching is much easier than with Subject Oriented programming.

4.1 Our Conclusion

Hyper modules sounds like a powerful new method to solve some of the separation of concern problems currently present in most development projects. It takes the best of Subject-Oriented Programming and makes it usable. It builds on top of Object-Oriented programming and is thus optional. Since it is in its early stages it is hard to tell if it will be over-used and thus complicate the code against the intention. There might also be some problems with class inheritance; what if I want/need to inherit from a class after it has been combined with a hyper slice? Also some research needs to be done in how to easily compose hyper slices. Hyper modules helps dealing with functional concerns, but not necessarily with non-functional concerns as those dealt with in Aspect-Oriented Programming.

5 Superimposition

As stated earlier, the main objective of software engineering is to improve software quality, reduce costs, facilitate easy maintenance and evolution of the software product. One way of doing this it to develop re-usable components. The components are often used in many projects and thus they are often very complete and well tested, which helps insuring low costs and improved software quality. Unfortunately, components are not the holy grail, as they often have to be adapted to the system requirements - which often requires understanding of how the components are implemented.

In the paper that we have examined, five criteria are used for classifying various ways of adapting components. These are:

- **Transparency**; the adaptation between the component and the user should be as transparent as possible, thus the components should not feel alien to the product.
- **Black-box**; the user should not need to know about the internal structure of the component, only its interface.
- **Composability**; the adaption technique should be composable with the component without changes to the component; the composed component should have the same composability as the original; it should be possible to compose the adaptions.
- **Configurability**; it should be possible to configure the adaptation technique to fit the users needs.
- **Re-usability**; the adapted components should be re-usable, which is often not the case since the configuration is often intertwined in the generic adaptation.

There are three often used methods for component adaption; copy-paste, inheritance and wrapping, all with there advantages and disadvantages. With copy-pasting the component into the project or inheriting from the component, only the *transparency* requirement is fulfilled. Wrapping the object, on the other hand, does not help *transparency*, but somewhat fulfills the *black-box*, *composibility* and *re-usability* requirements. The reasoning behind this can be found in [Bosch, 1998], though it should be quite obvious if you have worked with any of these techniques.

5.1 The idea

Superimposition is a way of describing an adaptation as some kind of mapping. This makes it quite easy to compose different adaptations and satisfy the requirements stated above. Each object is described as an interface, a set of methods, a set of states (instance variables) and a mapping from the interface to the methods. The adaptation is performed by modifying these by standard mathematical operators. Instead of going into details with these, we recommend the interested reader to read the actual paper.

As simple example to demonstrate the idea, is a restriction of an interface: A restriction can be described as a set of interface methods that we want to keep, plus a function that actually performs this operation.

Similar rules can be made for other types of adaptations. We mentioned that these adaptations can be categorized in three different types: *Changes to component interface* (like function renaming), *component composition* (like delegation of request), and *component monitoring* (reacting on certain conditions in the state of a component).

5.2 Our Conclusion

Superimposition sounds like a good way to improve component adaptation and it also makes it possible to reduce the overhead when composing adapted components. This is because the binding can be done directly (only one wrapping) instead of a wrapping of a wrapping. It still requires, though, that you fully understand the interface of the component before you do the adaptation. If the components does not export enough in its interface you will still have to deal with the internals of the component, and if the component exports too much, it will be hard understanding the interface. Whether the method works well in practice is hard to say, but it sound promising.

When reusing components in a project one often has to write a wrapper to make the components compatible with the components already in the project. Sometimes this is also necessary to make data types compatible (like different representations for strings). We do not see how superimposition can solve this often-occurring problem.

6 Further reading

If there is interest to read more about the suggested methods, we suggest reading the actual papers, as well as some of the papers referred from these. We suggest reading the papers in the same order as we have dealt with them. This way you will gain the background for the later papers and you will also get a good idea how the theories are progressing.

7 Conclusion

After reading the four proposals on improving software development techniques, as described in this text, we come to the following conclusion: While all texts provide interesting new ideas, the stage in which the development of the ideas are, differ a lot. The proposed ideas are not all usable in their current form. Subject-oriented programming, for instance, is a nice idea, but it seems virtually impossible to make it usable in the near future. Hyper modules, seen as a weak version of Subject-Oriented programming, on the other hand seems like a technique that could be introduced in programming languages in a foreseeable future. Superimposition and Aspect-Oriented programming seem like techniques that are created as solutions for practical problems that the authors experienced. Both techniques could be useful and could be implementing in some application-specific tools. After some experience is gained with these techniques, we think they are good candidates for being introduced into current programming languages.

8 Acknowledgements

We want to thank our teachers Rein Smedinga and Jan Terlouw for their course and for further reviews of this paper.

References

Bosch, J. (1998). Superimposition: A component adaptation technique.

Harrison..., W. (1999). N degrees of separation: Multi-dimensional separation of concerns.

Harrison, W. and Ossher, H. (1993). Subject-oriented programming (a critique of pure objects). *OOPSLA*, 411–428.

Kiczales..., G. (June 1997). Aspect-oriented programming. *Proc. European conference on Object-Oriented Programming (ECOOP), Finland, Springer Verlag*, **LNCS 1241**.