Towards Reusable Heterogeneous Data-Centric Disentangled Parts

Michael Reinsch and Takuo Watanabe
Department of Computer Science,
Graduate School of Information Science and Technology,
Tokyo Institute of Technology, Japan
mr@uue.org, takuo@acm.org

Abstract

This paper presents our ongoing research towards a safe system evolution.

Our approach is based on data-centric, object-oriented systems. Within those systems we utilise (i) multi-dimensional separation of concerns, (iii) explicit, language-independent type declarations in the form of an ontology and (ii) component technology.

With this combined approach it is possible to cope with a growing code base, and to safely reuse structure and code which supports a safe system evolution.

Keywords: object-orientation, data-centric architecture, reuse, ontology, separation of concerns, software evolution, components

1 Introduction

System integration still provides a great challenge. When looking at today's applications and electronic devices, the amount of integration is still very low. Seen from the perspective of ubiquitous computing [8], system integration in heterogeneous environments is probably one of the key aspects.

In our ongoing effort to research ways to build software that easily integrates with other software, we identified that data-centric architectures could be helpful. However there are some problems regarding code reuse, safe combination and safe evolution which arise easily within those architectures.

Safe combination and safe evolution of systems need even more attention when taking ubiquitous computing into account, as this implies a large, integrated, loosely coupled system, whose components are changed independently from each other.

1.1 Data-Centric Architecture

A data-centric architecture should help ease system integration by shifting the system architect's perspective: Data becomes the most important part of the system, with the functionality grouped around the data, providing the user with different tools to manipulate this data. Thus it becomes important to share this data between those tools.

We see this in contrast to the traditional, application-centric architecture, in which full-blown applications come with a large, but more or less fixed set of functionality and lock the data in their own and sometimes proprietary file formats.

The tools in a data-centric architecture on the other hand can be small and can concentrate on one specific function. Through loosely coupled co-operation between those tools, a large, flexible and integrated software system should be achievable.

To implement such a software system, an object-orientated approach seems to be adequate. This allows the creation of a model which is closely related to the real world and it also allows to utilise abstraction, information hiding and reuse. In addition, component technology can provide the techniques to realise safe deployment and safe reuse of components through contracts, and it can help to achieve a more loosely coupled system [5].

However, a naive implementation will most likely lead into problems, caused by system evolution. The rest of this section is devoted to discuss such problems.

1.2 Object-Orientation, Information Hiding and Reusability

For this discussion it is first necessary to examine the ways an object-oriented system can be designed. Our main objective is to create a high quality, simple and maintainable system whose object-oriented code makes use of information hiding and can be reused.

In order to follow the principle of information hiding, it is necessary to not expose the internal structure of an object to external entities. Unfortunately this is often done, e.g. by utilising reputedly safe getter and setter methods [3], and is likely to cause problems when changes to the internal structure are required. But on the other hand, without the possibility to access the internal structure, the class ends up as an example of the anti-pattern "The Blob" [1] because it has to provide all functionality that requires access to its internal structure. Thus a solution which can allow for code growth without an impact on information hiding is desirable.

A viable compromise for this dilemma seems to be provided by structural design patterns. They make use of several objects and the internal structure of an object is opened only to a small number of objects. However this increases the complexity of the solution and most likely also the complexity of the whole system since other design patterns such as abstract factories then probably need to be applied as well. Thus, in using structural design patterns we trade information hiding with an increase in complexity.

While this solves the problem for one static software system (assuming the number of classes with access to the internal structure of an object is low and well documented), changes to the internal structure of an object can cause problems when considering reuse in several independent components. Those changes need to be tracked and each component needs to be explicitly checked by a developer, as there is no formal description stating this dependency.

1.3 Dependencies, Information Hiding and Reusability

Regarding dependencies a similar discussion to the one above is required. The main question is once again, where code should be placed that contains a dependency to another class. Again there are two choices: Either in the class itself or in a helper class by e.g. applying a matching design pattern. Moving the code into a helper class normally comes with advantages but also increases complexity.

Additionally, another conceptional problem cannot be solved that way: if two different versions of a class are required within the same application and the execution environment does not support versioning, they are likely to clash.

1.4 Summary

From the discussion above it is obvious that object-orientation by itself does not provide a good solution for the described problems. It is necessary to explicitly add extra structure to cope with the growth of code. And this still leaves the questions concerning system integrity unanswered in the cases when code is reused or parts of a system are changed.

An approach which addresses the problems concerning complexity, reuse and information hiding as discussed above, and in addition allows to detect changes within the internal structure could thus simplify development and allow safer reuse, safer system composition and safer system evolution.

2 Concepts

In this section we present the concepts on which our approach is based. Generally speaking, we combine multi-dimensional separation of concerns with techniques known from component systems and add an explicit and language-independent type declaration in form of an ontology. In the following subsections we discuss this in more detail, and in section 3 we then present a possible implementation which also serves as an example for the points discussed here.
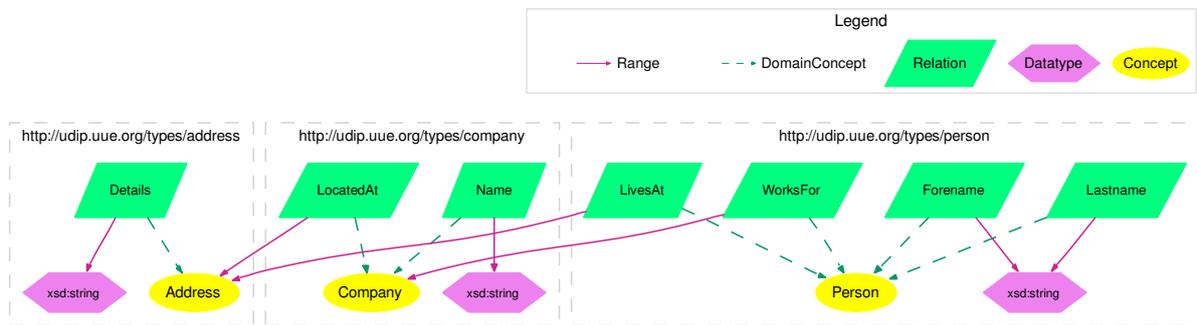
Figure 1: Simple ontology describing the concepts `Person`, `Address` and `Company`

2.1 Ontology

In order to encourage a data-centric architecture, we explicitly model the data structure of a software system using a language-independent ontology. A very simple ontology describing the concepts `Person`, `Address` and `Company` and their relations is provided in Figure 1 as an example. The system's data itself can then be stored within that ontology. This achieves language-independence for data as well, and allows reflection and reasoning over the stored data. From the meta-data contained in the ontology, code for different programming languages can be generated. All this then allows us to integrate code which is based on the same ontology, albeit possibly written in different programming languages.

In addition, it is also possible to store the declarations of the deployed parts (see next subsection) in the ontology. Therefor we added an additional complete meta-type system which uses OO-terminology and which fits our needs better than the one used by the ontology itself. Hence it is possible to reflect, for example, about the current deployment status and change system properties at run time. Using this mechanism, dynamic system adaptation could be implemented.

2.2 Parts

To cope with dependencies, information hiding, reuse and a growing code base, we introduce *parts*. Parts mainly serve to allow a multi-dimensional separation of concerns [6], but in another way they also serve as small components as they define the dependencies and the exported interface, and form the unit of composition [5].

To realise this, a part defines which other parts and which concepts from the ontology it requires. It also provides a list of methods which are to be "attached" to the ontology. This list of methods also forms the exported interface of that part.

For example, a part which provides the functionality to export the content of a `Person`-object in the VCard-format might be defined like this:

```
Part: VCardExport
  Expects: The ontology in Figure 1
  Exports: String toVCard() on Person
  Required Parts: None
```

The implementation (see section 3 for an example) of this method, which is also provided by the part, has access to the internal structure of the object it is attached to. To be more precise: It has access to that part of the internal structure that was defined in the expects-statement. Thus the expects-statement declares both, the requirements and the restrictions for the part. In addition, the implementation of a method can only execute the methods defined in its part or in one of the required parts, on those objects it can access.

2.3 Summary and Implications

With this approach all methods are now associated with a class and a part. The parts are used to group those methods together which belong to the same concern. Parts thus help to remove all unwanted methods from the view of other parts.

They also define a namespace within the class to avoid clashes of methods. In addition, the implementation of a method is no longer done within the class but within the part.

This separation of class and method using parts has several advantages. First, it prevents a class from becoming large and hard to maintain. The code for a part also only needs to be deployed when it is required. Second, the implementation for parts can be written in different programming languages, as the authority defining the structure and types (i.e. the ontology) is language neutral. Third, the code for a part can also be executed on a remote host, allowing e.g. the small or GUI related parts to reside on a thin client whereas the parts doing heavy data processing could reside on a server. Forth, by explicitly defining the data structure a part expects, changes to the internal structure can be detected at deployment time and incompatible parts can be rejected.

3 Early Prototype

To further test this approach we implemented a prototype in Java 5. We chose Java to show that a simple translation to an object-oriented language is possible. The prototype also states that: (i) each part should be compilable on its own without a special compiler, and (ii) the strong typing of Java should be kept and no casts should be required to access methods or attributes.

The ontology used by this prototype is stored in OWL [7]. This makes it easier to create and read the ontology because tools and libraries to process OWL are available.

The prototype mainly consists of a code generator. First, for every part interfaces like the following one are generated:

```java
public interface VCardExport {
    String toVCard();
}
```

Then for each concept in the ontology a class is generated:

```java
public abstract class Person extends Subject {
    public final VCardExport cVCardExport
        = new VCardExport() {...}
    protected final String getForename() {...}
...
```

The ontology's relations are translated into attributes (e.g. getForename) of those classes. Every class to which methods are attached, gets additional final attributes whose types are one of the parts' interfaces (e.g. cVCardExport). The generated code within the class dispatches all calls to a core object which holds all attributes as well as all registered part implementations.

An implementation for the part VCardExport could then look like this:

```java
public class PersonVCardImpl
    extends Person implements VCardExport {

    public String toVCard() {
        StringBuffer res = new StringBuffer();
        res.append("BEGIN:VCARD\nVERSION:3.0\n");
        res.append("N:").append(getForename());
...
```

To be able to call this method, some part must define that it uses the concept Person and the part VCardExport. Then some object within the part can obtain a reference to a Person-object and call the method like this:

```java
String vcard = myPerson.cVCardExport.toVCard();
```

With this we have shown that an easy translation to an object-oriented language is possible.

4 Related Work

In this section a discussion about related work is presented. We first introduce the related work briefly and then discuss the similarities and differences.

4.1 Subject-Oriented and Hyperspace Programming

The main concept behind hyperspace programming [4] (being by itself a more generalised form of subject-oriented programming [2]) is the multi-dimensional separation of concerns. Therefore it is quite similar to our approach, and in fact we borrowed several ideas from their approach. But there are some differences:

We bring in techniques from component systems which allow us to define clear dependencies. Through the dependencies between our parts, it is possible to build a layered system. Thus we do not require the parts to be complete applications on their own. We instead see our parts as being quite small, capturing what perhaps could also be called mini-concerns, and to be reused by other parts.

4

Composition rules are considered to be an important part of hyperspace programming. In our approach on the other hand, the types used by the parts are standardised through the ontology, eliminating the need to create mappings between classes and attributes. Though in a later stage, we might reintroduce some kind of composition rules in the form of mappings between different ontologies.

Through the meta-data present in our approach, it is also safe to compose the system at run-time, eliminating the need of a special compiler. Additionally, this meta-data can also be exploited in other ways as discussed earlier.

4.2 Aspect-Oriented Programming

In AOP the main focus lies on cross-cutting aspects which it tries to consolidate. Thus there are similarities to our approach which also allows us to group cross-cutting concerns together. But both approaches are different in the same way that hyperspace programming is also different from AOP: In AOP the aspects are weaved into a core implementation. With our approach, a core implementation does not exist, but only different concerns. Additionally, the granularity is different: AOP allows us to weave code into methods, whereas our approach allows adding methods to classes.

5 Concluding Remarks

We presented a combination of multi-dimensional separation of concerns with techniques known from component systems and an explicit and language-independent type declaration in form of an ontology. Through the separation of methods from their classes (which are defined by the ontology's concepts) and by regrouping those methods in parts we gain extra flexibility, can cope with the growth of the code base and reuse those parts. In addition, existing parts can be easily extended by adding new parts. Through explicitly stated dependencies between parts and the definition of the required type structure for a part, we achieve a safe way to combine several parts to a larger system. Finally, the usage of a language-independent ontology to define the types of the system helps in reuse of not only code but also the structure of a system.

6 Future Work

In this paper we presented a snapshot of our on-going research in this area, hence this work is not complete, much remains to be done:

After further extending our existing prototype, a larger set of integrated, yet loosely coupled tools needs to be implemented, also demonstrating software evolution. With this code base we then hope to be able to further examine safe composition and re-composition at run-time. Based on those results, automatic system adaptation could be studied.

References

[1] W. J. Brown et al. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 20 Feb. 1998.

[2] W. Harrison and H. Ossher. Subject-Oriented Programming: A Critique of Pure Objects. In *Proceedings of ACM OOPSLA '93*, pages 411–428, Sept. 1993.

[3] A. Holub. Why getter and setter methods are evil. *JavaWorld*, 5 Sept. 2003.

[4] H. Ossher and P. Tarr. *Software Architectures and Component Technology: Multi-dimensional Separation of Concerns and the Hyperspace Approach*, chapter 10. Kluwer Academic Publishers, 2002.

[5] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1998.

[6] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 107–119, 1999.

[7] Web-Ontology Working Group. OWL Web Ontology Language, Overview. Online available at http://www.w3.org/TR/owl-features/, 10 Feb. 2004.

[8] M. Weiser. The Computer for the 21st Century. *Scientific American*, Sept. 1991. Online available at http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html.