Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competitive market position.

Fraunhofer IESE is directed by Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern
Preface

The 2002 GCSE Young Researchers Workshop is the fourth in a series of Workshops supporting young researchers in the areas of component based software engineering, product line Engineering, generative programming and separation of concerns.

After the great success of the 3rd GCSE Young Researchers Workshop this workshop which is traditionally held in conjunction with the NetObjectDays again aimed at providing a platform for young researchers to present their work. The papers submitted and the presentations during the workshop were commented by experienced panelists (Prof. Mehmet Aksit, University of Twente, Netherlands; Prof. Peter Knauber, Mannheim University of Applied Science, Germany; Prof. Rainer Unland, University of Essen, Germany). The workshop serves as a forum for the participants to get in contact with other researchers in the field and to become familiar with other approaches and future research topics. The topics of the workshop were:

- Component-based software engineering: Reuse, distributed platforms, distributed systems, evolution, analysis and design patterns, development methods, formal methods
- Product line engineering: Architectures, scoping, domain analysis, product line implementation and testing, variability
- Generative programming: Reuse, meta-programming, partial evaluation, multi-stage and multi-level languages
- Runtime code generation, compilation, active libraries, synthesis from specifications, development methods, generation of non-code artifacts, formal methods, reflection
- Separation of Concerns: Aspect-oriented programming, intentional programming, and multi-dimensional separation of concerns
- Integration of generative and component-based approaches
- Domain engineering, analysis, and languages

This years workshop took place at October 10 in Erfurt during NetObjectDays (see http://www.cs.uni-essen.de/dawis/conferences/Node_YRW2002/). The seven papers included in these proceedings were presented there. We would like to thank our participants, the panelists and the audience for creating a friendly atmosphere, giving and receiving feedback making the workshop a success.

See you at next years young researchers workshop

The organisers

Matthias Clauss
Stefan Hanenberg
Isabel John
Katharina Mehner
Ragnhild van der Straeten
# Table of Contents

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrating Heterogeneous Data Sources into Federated Information</td>
<td>1</td>
</tr>
<tr>
<td>Systems</td>
<td></td>
</tr>
<tr>
<td><em>Harald Wehr</em></td>
<td></td>
</tr>
<tr>
<td>Dynamic Attribute Addition through the Use of Metaclasses</td>
<td>11</td>
</tr>
<tr>
<td><em>Kopanitsanos Antonis</em></td>
<td></td>
</tr>
<tr>
<td>Extracting Implicit Contracts from .NET Libraries</td>
<td>20</td>
</tr>
<tr>
<td><em>Karine Arnout</em></td>
<td></td>
</tr>
<tr>
<td>Graphical User Interfaces Composed of Plug-Ins</td>
<td>25</td>
</tr>
<tr>
<td><em>Johannes Mayer</em></td>
<td></td>
</tr>
<tr>
<td>Concepts for a Product Line Knowledge Base and Variability</td>
<td>30</td>
</tr>
<tr>
<td><em>Roland Laqua</em></td>
<td></td>
</tr>
<tr>
<td>Supporting Component-based Development by Enriching the Traditional API</td>
<td>40</td>
</tr>
<tr>
<td><em>Marco Castaldi, Nathan D. Ryan</em></td>
<td></td>
</tr>
<tr>
<td>Data Structures for Images on Java</td>
<td>46</td>
</tr>
<tr>
<td><em>Hans Braxmeier</em></td>
<td></td>
</tr>
</tbody>
</table>
Integrating Heterogeneous Data Sources into Federated Information Systems
– Work in Progress –

Harald Wehr
University of Applied Studies and Research
Friedrichstr. 57-59, D-38855 Wernigerode
hwehr@hs-harz.de

Abstract. Federated Information Systems are based on gathering information from different heterogeneous data sources. The Internet - a Meta FIS in one sense - demands for the ability to query many different information sources. Obviously, most of them are structured very differently. Considering databases, XML files, other structured text files or Web Services as information supplier the complexity of integrating the information with their various describing models is not easy to handle. Even companies that only want to setup a global information system (e.g. a Data Warehouse) are faced with integration problems. As a lot of departments were able to develop their own local information pools there can be a variety of different sometimes redundant data sources even inside one company.
In this paper an new approach of building such a global information system is described. Besides the theoretical aspects it is mainly shown that the system is really working and can easily be adapted to the special needs, a certain Federated Information System has to fulfill.

1 Introduction

A lot of research has been done in the fields of integrating different heterogeneous data sources during the past years. Besides the integration of (mostly) relational databases it is also desirable to add semi-structured text files for instance XML files or other structured ASCII files. Considering Web Services as XML streams this kind of information sources could also be integrated like simple XML files.

In this paper we present an architecture of a system that provides uniform access to the mentioned kind of data sources. The system is very easy to configure and maintain. Adding additional data sources does not need restarting the system. By writing simple XML files, describing the structure and mapping of a new source, integration is as easy as possible.

The following work has been developed within the MobiHarz project\(^1\). The project looks for new concepts to increase the attractiveness of public transport

\(^1\) The project MobiHarz is part of the global research program “Freizeitverkehr” sponsored by the German Federal Ministry of Education and Research.
in tourism. Therefore, a system has to be developed that can easily integrate different information sources, such as accommodation data of hotel chains, data of certain points of interest and information of available public transport companies.

The paper is structured as follows. The next section contains an overview of existing theoretical approaches of Federated Information Systems. Section 3 describes the overall architecture of the system, while section 4 explains important parts in more detail. After that different related projects are enumerated. The current work is finally concluded in the last section with an overview of the next steps.

2 Federated Information Systems

Federated Information Systems are expected to be a completely new generation of software systems [1]. Their main task is to operate as a global layer over existing data sources. It is important to consider that these sources have certain characteristics making the integration process very difficult: [1][2][3]

Heterogeneity Local data sources are mostly developed for a special purpose. This often results in different solutions for storing information of the same real-world objects. Information can be stored in databases with different models (relational, object oriented), ASCII files or be available as Web Services. It is obviously that these kinds of sources are accessed through different interfaces, protocols and languages (Syntactical Heterogeneity). Even the same data model can cause mapping conflicts due to different understandings of the real world (Logical Heterogeneity).

Autonomy Data sources, integrated in a FIS do not give up their autonomy. First of all they keep their Design autonomy. It’s up to them how the contained information is stored. Furthermore they are able to decide which other systems are allowed to communicate with them. The local data source should be able to enter or leave the federation at any time (Communication autonomy). Additionally each component is independent in deciding how the incoming requests are scheduled and executed. The federated system must not change execution priorities for its requests (Execution autonomy).

Distribution Sources that have to be integrated do not always reside on the same host. It’s likely to be that they are on different hardware platforms and operating systems and can only be accessed through certain network protocols.

Based on [1][2] FIS can be divided into three different types:

- **Loosely coupled information systems** do not have a federated schema. They only provide a query language to access the different local sources. By implementing this kind of system the user is responsible for addressing the right information source with the specific desired elements.
FIS that only contain database components are called federated database systems (FDBS). A lot of research has been done in these kinds of integration systems. For a detailed overview see [4].

*Mediator-based information systems* are not limited to certain kinds of components. The system is able to “mediate” between the user and structured as well as unstructured information sources.

According to the goals of our project our system architecture can be classified as a mediator-based system. It is not possible to limit the potential data sources to databases. Furthermore it is required to add or remove other components without changing the federated schema.

Building a FIS can be done by two different approaches [5]. The so-called *bottom-up strategy* is often used for developing a FDBS. Every component is completely integrated into the federated schema. This procedure often leads to a very complex global schema. Adding a new source always leads to a new and bigger global system. The *top-down strategy* is mainly used for mediator-based systems. It does not provide a complete and exact access to all information of the integrated sources. The system is built for a specific information need it has to fulfill. Schemas of the local components do not affect building issues of the federated schema. They will only be considered during the step of mapping elements of data sources to elements of the global schema. By following this approach there is no need for altering the federated schema. Additional sources can be integrated without any effects to the federated model. Of course it has to be considered that the global schema can’t be as complete as the one developed with the *bottom-up strategy*.

### 3 System Architecture

An overview of the general architecture of MobiHarz is depicted in figure 1.

As mentioned above the project provides a homogeneous interface to tourists hiding the underlying heterogeneous information sources. Several output formats like HTML, WML or PDF have to be offered as different Internet enabled client devices shall be able to obtain information from the federated system [6]. The content management system is therefore responsible for generating dynamic content in the desired output formats based on the business objects residing in the federated system. All business objects are kept in random access memory (RAM). A thread with lower priority builds a complete new object system at regular intervals that replaces the old one. This procedure entails several advantages:

- The response times of the system are very small as the underlying sources are not queried at each incoming request.
- If a source is temporarily unavailable its information can still be brought to the customer.
- Additional data sources can be integrated without restarting the system [9].
Besides these advantages it is clear, that very big sources cannot be easily integrated due to limited RAM. This problem could be solved with an additional database that consequently fits to the developed business objects and temporarily stores the integrated information as it is done in data warehouses.

In the following section the process of integrating different sources is described in more detail as this step has already been implemented in our FIS.

### 4 Integrating the Sources

Integration of heterogeneous data sources causes several conflicts that have to be resolved. They can be divided into semantic conflicts, descriptive conflicts, heterogeneous conflicts and structural conflicts [7][8]. Solving these conflicts the MobiHarz project uses a 'semi-automatic' approach for the process of integration [10]. Each source has to be described in a simple configuration file containing
field names, data types and optional primary and foreign keys. A second file is responsible for the mapping between one source and the project’s target business model. Figure 2 shows the way of integrating different kinds of sources.

Transforming the Sources

According to the five-level schema architecture of an FDBS ([2]) each source is transformed into the component schema by certain wrappers. One source and its information is hold in one or more Document Object Model (DOM) [11]. They represent the so-called canonical or common data model as the starting point for generating the corresponding Java objects. The different sources are transformed in the following ways:

Databases All relational databases that provide a “Java Database Connectivity” (JDBC) driver can be integrated into the system. Each relation in a database is added to the Document of the source. Every row is appended as further Element. Elements can reference other elements to reflect relationships between the former tables. As almost all database vendors develop the mentioned JDBC driver MobiHarz is able to handle a large number of different relational databases.

---

**Fig. 2.** Integration of different data sources
**Text Files** There is no common rule how many Documents are produced from a certain text file. To simplify matters each XML file could result in only one document. Other files could of course be represented by several documents. It is possible to write additional wrappers for files that cannot be transformed with the default ones provided by the MobiHarz’ system.

**Web Services** As Web Services deliver XML over HTTP they can be transformed as easy as normal XML files into one Document.

It is important to consider that the process of describing the structure of one source (relations, fields, data types) often can be automated. JDBC drivers are able to provide meta data about the connected database. Document Type Definitions (DTD) inform about the structure of a certain XML file. Even Web Services have their own Web Service Description Language (WSDL) that gives further information about the structure of the content.

**The Business Model**

As it is intended to use the federated system not only for touristic concerns the mapping into certain Java business objects must not be hard coded. By using the federated system the administrator can provide his or her desired business classes. Each class or one of its super classes must inherit from the abstract DefaultBusinessClass in order to provide all objects with the following required attributes:

- **Source:** the source from which a certain object has been instantiated,
- **LocalObjectId:** an Id that clearly identifies an object within its source (sometimes optional),
- **AttributesFillable:** information whether certain attributes could be filled by the source or not,
- **Instances:** class variable that holds all instances coming from one source.

Furthermore it is important to create the appropriate relationships between the business objects. Unary as well as binary OneToOne-, OneToMany-, ManyToOne- and ManyToMany-relations should be supported. In order to build a required relationship the business classes have to follow these restrictions:

- Objects on the “One”-side of a relationship must have an attribute whose type is of the class of the other participating object.
- Objects on the “Many”-side of a relationship must have an attribute whose type implements the interface java.util.List. This container holds the references to the participating objects in the relationship. Certain implemented mapping actions (see next section for details) make sure that the attributes are filled correctly.

**Integrating the sources**

As mentioned above every data source must be provided with a mapping file that describes how certain fields are used to build the attributes in the target business
objects. This mapping-file is again a simple XML-file. Neither the mapping of
the Business-Objects nor their attributes are hard-coded in Java. As we strongly
use the reflection API the mapping process can be described and easily altered
by writing simple text files.

At this point of integration it is not longer to be seen which kind of source
holds the information. As the mapping process takes places at the export schema
(see figure 2) the original structure of the integrated source is hidden. Certain
already implemented actions make sure that the listed attributes can be set
correctly from different sources with different structures. In addition the user
is able to write own actions that belong to certain sources. Thus it should be
possible to integrate even those sources whose structure completely differs from
the structure of the federated schema.

Following steps have to be done for integrating a new source:

1. First of all a new source must be introduced to the federated system. The
   file sources.xml contains all available sources with their names, responsible
   wrapper classes, access information, locations and so on.
2. According to the component schema, information of the source have to
   be transformed into documents. The name of the wrapper class and the
   corresponding implemented class must be available. Wrappers for JDBC
databases and XML files already exist in the federated system and can
be used by their names and Java’s dynamic class loading. Obviously, some
sources can’t be transformed by the supplied standard wrappers. In this case
the user must implement a new one for the desired source.
3. Additionally each source must be provided with two configuration files. The
   first one contains information about the fields and data types of the com-
   ponent schema. The second one informs about the mapping actions that
   have to be executed. In some cases it may be necessary to implement further
   actions for a source that can’t be mapped by the default ones.

Having done these steps for a new source during the next ‘refresh’ of the
system all sources (including the new one) will be integrated. New objects will
be instantiated and have to pass through the way of the identification process
as depicted in figure 1. Afterwards a set of objects is waiting for requests from
the content management system.

5 Related Projects

In the last years a lot of research has been done in the fields of federated in-
formation systems. Three different projects shall be described here in further
details.

– TSIMMIS (The Stanford-IBM Manager of Multiple Information
  Sources)

Following the descriptions of the project integration of heterogeneous data
sources is possible. As global federated schema the developed ‘object ex-
change model’ is used. This model is able to represent different objects that
can contain further objects. Inheritance is not possible within this model [12]. The successor of TSIMMIS is called MIX. It is completely based on XML as global schema. Queries against this schema are done by using the XMAS-language [13].

- **Information Manifold**
  This prototype has been developed by AT&T in 1995/96. The federated model is mainly a relational one provided with some extensions. Like TSIMMIS the federation is virtual. Every time a user accesses the system it identifies relevant sources and executes sub-queries to them. After that the user is responsible for cleaning overlapping information or performing object fusion mechanisms [14].

- **SIMS (Search in Multiple Sources)**
  The federated schema in this project uses the hierarchical terminological knowledge base (Loom). It consists of different nodes representing each class of objects and relationships between them. An incoming user query is translated by certain mapping descriptions in a corresponding local query. Again a mechanism of avoiding overlapping information is missing [15].

6 Conclusions

The outlined architecture satisfies almost all requirements of a federated information system. As printed in the title the described work is still in progress. The depicted architecture is the result of a former prototype shown in [6]. Besides the integration of different kinds of data sources it now offers a more flexible way of extending the system.

Our federated system currently provides following features:

- Several heterogeneous data sources can be easily integrated, updated or just removed from the global information system by simply changing text files.
- A large amount of available databases, structured text files and Web Services are supported due to already available wrappers. Of course it is possible to write own wrappers that import other currently not supported data sources.
- The mapping process is done by certain mapping actions. The system contains already implemented actions. Again, the mechanism of dynamic class-loading enables the user to write his own actions for special kind of data sources with an uncommon structure that does not fit into the global business model.
- Due to the JAVA Reflection-API the object-oriented federated model can be provided by the user. It is possible to use this system in nearly every environment where integration is needed.

In comparison with the related projects our current prototype uses the so-called ‘materialized integration’. On the federation level the content of the local data sources is completely mirrored in business objects. Following this way of integration update procedures have to be developed. Another disadvantage of the current approach is that all the information are kept in main memory. Of course
this can be changed by installing an object-oriented database at the federated level. According to our project needs this is not necessary at the moment.

This paper mainly refers to the problem of integrating different sources. After this step the conflicts of object identification have to be solved. As the sources may contain overlapping information several instantiated objects may refer to the same “real-world” object. Furthermore the business objects must provide a common interface to the mentioned content management system to support different output formats. These tasks still have to be done in future.

Acknowledgement:
I would like to thank Bernd Müller for a lot of useful comments on this paper and the work in the project MobiHarz.

References


Dynamic Attribute Addition through the use of Metaclasses

Kopanitsanos Antonis

Department of Informatics and Telecommunications,
University of Athens, Panepistimiopolis, Ilissia
GR15778, Athens, Greece
antonis@di.uoa.gr

Abstract. Class evolution is a key issue in object oriented systems evolution. In particular much research has focused in topics raised by class evolution either through coercion or versioning. In this paper dynamic attribute addition in terms of versioning is addressed. Dynamic attribute addition in this context concerns modifications of variables of objects and classes at runtime. A candidate mechanism based on the notion of metaclasses is proposed to serve for this purpose. A case study of the use of the proposed mechanism is presented. Finally an evaluation of the benefits and the drawbacks that emerge from its use is established.

1 Introduction

Object oriented models are believed to better and more naturally reflect the behaviour and organization of real world phenomena, incorporating more of the real world semantics. But the real world itself may change and new requirements may arise for an object-oriented system. So software developers should be able to perform a large number of changes to an already existing system so that the system may continue to suit the evolving user’s needs.

The relevant process commonly known as class evolution addresses issues, like behavioural and structural evolution. Behavioural evolution deals with the ability to change the behaviour of objects. Structural evolution is concerned with reclassification of existing objects and altering of objects attributes. The latter is the special problem we focus our interest on, in this paper. In particular we are concerned with the ability to change dynamically a class’s association with properties even if this class is already populated with instances. In this case special care must be taken with these instances, which they should always comply with the definition of their class even if this class has a newer version.

According to the Smalltalk meta-programming model (Goldberg and Robson, 1983) every class is itself an object and as such an instance of a metaclass. So a metaclass is a class of a class ie. the objects of this class can themselves act as classes. The notion of metaclasses is not directly supported by popular object oriented languages like, C++ and Java. Nevertheless it should be examined as a candidate mecha-
nism to provide a solution to the problem of attribute addition and removal at runtime.

In this paper we present a model based on the notion of metaclasses as a solution to the aforementioned problem.

The rest of the paper is organized as follows. In section 2 we present issues concerning class evolution and metaclasses. In section 3 we describe the example case we deal with in this paper. In section 4 using the example case we describe the problems that can be experienced if current languages are used. In section 5 we present the proposed approach and explain how it can support dynamic attribute addition and removal. Finally in section 5 advantages and disadvantages of the proposed approach are presented together with future work.

2 Class Evolution

As mentioned before object oriented models successfully describe real world phenomena. Objects combine data together with their interpretation. But the real world itself may change and new requirements for information, or functionality to manipulate information in the application may arise. So, software developers, working with an object oriented system should be able to modify classes or the class hierarchy so that they fully suit their needs.

According to [1] we can divide the changes one system may support into the following categories:

<table>
<thead>
<tr>
<th>1) Changes concerning instance variables</th>
<th>2) Changes concerning methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a variable</td>
<td>Add a method</td>
</tr>
<tr>
<td>Remove a variable</td>
<td>Remove a method</td>
</tr>
<tr>
<td>Rename a variable</td>
<td>Rename a method</td>
</tr>
<tr>
<td>Redefine the type of a variable</td>
<td>Redefine the signature</td>
</tr>
<tr>
<td>Change the inheritance origin</td>
<td>Change the code</td>
</tr>
<tr>
<td>Change the default value</td>
<td>Change the inheritance origin</td>
</tr>
<tr>
<td>Modify other kinds of variables</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3) Changes concerning classes</th>
<th>4) Changes concerning inheritance links</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a class</td>
<td>Add a superclass to a class</td>
</tr>
<tr>
<td>Remove a class</td>
<td>Remove a superclass</td>
</tr>
<tr>
<td>Rename a class</td>
<td>Change superclass precedence.</td>
</tr>
<tr>
<td>Modify other class properties</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5) Changes concerning the associations of a class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a new association</td>
</tr>
<tr>
<td>Remove an existing association</td>
</tr>
<tr>
<td>Change the multiplicity of an existing association</td>
</tr>
</tbody>
</table>
The possible changes shown in the above tables may have different impact for the underlying system. In particular the changes we focus our interest on in this paper may affect:

- Clients of the system, the application programs.
- The objects, which should always comply with the definition of their class, even if this class has a newer version.

With respect to this problem there are two different approaches, the coercion and the versioning approach.

The coercion approach regards all modifications as corrective and obsoletes former specifications accordingly. All instances of a modified class must be updated (coerced) according to the new definition, i.e., objects must always comply with the most recent specification of their class. But as all objects are converted, the coercion approach may imply information loss. Moreover, all clients of the system must also be modified and recompiled to be in accordance with the most recent version.

The versioning approach does not regard change as corrective, so former class or class hierarchy specifications are not obsolete upon change, but remain valid in coexistence with newer specifications. Objects must be properly enhanced, so that they may consistently behave under any class version. An important impact is that existing clients of a system may be unaffected by the change, continuing to regard the system in the same way as before. This may be useful for example when it is expensive or time consuming to modify and recompile application programs.

As we know objects are created as instances of classes. According to usual object-oriented approaches after their creation they cannot change association with properties because that is determined at the time of instantiation by the initial specification of their class. Nevertheless because of class evolution they have their structure altered by attributes added or removed dynamically and they need to be provided with extra storage space in order to be in accordance with the newer version of their class. At this point we should mention that special care must taken if the system provides persistence for the data.

A meta-class is a means to define the common properties of a set of classes. A meta-class is a "package" of properties. The assembly of properties in a meta-class package is sometimes called a meta-class structure. Creating an object with the properties defined in a meta-class is called instantiation of the meta-class. After that instantiation we obtain a new object, called a class, but now it is called also a meta-instance which has the properties packaged in the meta-class.

As mentioned before popular object-oriented languages like C++ and Java do not support directly this mechanism. However it can be implemented using the notion of static data they both provide. We come now to the description of the example case we deal with in this paper.

### 3 Example Case Description

Suppose we have a system that contains among other classes, the classes AdultPerson and IdentityCard the specification of which is presented below.
public class AdultPerson {
    int height;
    String surname;
    IdentityCard id;
    // methods
}

public class IdentityCard{
    int codeNumber;
    String dateOfBirth;
    // methods
}

Further suppose that the extent of those classes (instances) exists. But due to evolving user's needs a new variable is required to be added to each one of them. More specifically we need to add a variable of String type to the AdultPerson class having the name residence, and also a String type variable having the name authority to the IdentityCard class.

4 An Approach based on Inheritance

If current languages like C++ and Java were used, the incorporation of a new variable in a class could have been achieved using a well known object oriented technique called inheritance.

In our example the derived subclasses could have the following specifications.

class EnhancedAdultPerson extends AdultPerson{
    String residence;
    EnhancedIdentityCard eID;
    // methods
}

class EnhancedIdentityCard extends IdentityCard{
    String authority;
    // methods
}

After the insertion of the two subclasses, existing instances of classes AdultPerson and IdentityCard should be migrated to classes EnhancedAdultPerson and EnhancedIdentityCard accordingly. Moreover the reference, that every AdultPerson object holds, to its corresponding IdentityCard object should be replaced with a reference to an EnhancedIdentityCard object.

The use of inheritance either single or multiple leads to a number of well known problems including name collisions, homonymous attributes and common ancestor dilemma. Other problems that can be experienced particularly in our example are listed below:
a) Creation of complex hierarchies which are difficult to handle, if requirements for attribute addition are often.

b) Class explosion if for each attribute we want to add we have to insert to the system a new class.

c) Difficulties in attribute removal.

So we need a new approach in order to achieve attribute addition and removal in general, and in a dynamic fashion in particular.

5 Proposed Approach

We begin the description of the proposed approach by introducing a new concept concerning the presentation of attributes. Each one of these attributes is modeled as an instance of class Data the specification of which is presented below.

```java
public class Data{
    String variableType ;
    String variableName ;
}
```

As we can see for each one of the attributes its name and type are stored (using a String format) in the instance variables of the class Data. We could also have stored information concerning modifiers like protected or private, determining the access to the attributes being modeled but we decided to let it as part of our future work because its implementation requires extra support from the underlying software system. So we let all attributes be public.

According to the proposed approach the system’s class hierarchy should have the format presented in figure 1. As we can see at the top of the hierarchy is placed an abstract class called Meta class. All the other classes of the system inherit from that class.

![Proposed Class hierarchy](image)

The structure of the abstract class Meta class is as follows:

```java
abstract class MetaClass {
    static Dictionary variableMap;
    Dictionary variableValue;
}
```
Because the variableMap Dictionary is declared as static there is only one copy of it for all the concrete subclasses of MetaClass, so it can be used as a repository storing information about the attributes contained in these subclasses. According to our approach the initial information to be stored in the static variableMap Dictionary seems like the one presented in the following table:

**Table 1.** variableMap Dictionary.

<table>
<thead>
<tr>
<th>“AdultPerson”</th>
<th>data1</th>
<th>data2</th>
<th>data3</th>
</tr>
</thead>
<tbody>
<tr>
<td>variableType: int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>variableName: height</td>
<td>variableType: String</td>
<td>variableName: surname</td>
<td>variableType: IdentityCard</td>
</tr>
<tr>
<td>variableName: ID</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>“IdentityCard”</th>
<th>data1</th>
<th>data2</th>
</tr>
</thead>
<tbody>
<tr>
<td>variableType: int</td>
<td>variableType: String</td>
<td></td>
</tr>
<tr>
<td>variableName: codeNumber</td>
<td>variableName: dateOfIssue</td>
<td></td>
</tr>
</tbody>
</table>

As understood we can extend the above structure horizontally to include new data objects modelling attributes inserted at run-time, as well vertically to include new classes inserted in the system.

On the other hand instances of classes AdultPerson and IdentityCard store their instance data in the variableValue Dictionary, which their classes inherit from the abstract class MetaClass. The situation is depicted in the following tables:

**Table 2.** Instances of class AdultPerson.

| “height” | obj1 | “height” | obj4 | “height” | obj7 |
| “surname” | obj2 | “surname” | obj5 | “surname” | obj8 |
| “IdentityCard” | obj3 | “IdentityCard” | obj6 | “IdentityCard” | obj9 |

**Table 3.** Instances of class IdentityCard.

| “codeNumber” | obj10 | “codeNumber” | obj12 | “codeNumber” | obj14 |
| “dateOfIssue” | obj11 | “dateOfIssue” | obj13 | “dateOfIssue” | obj15 |

In the above picture we can see the mapping between the names of instance variables and the objects representing data for them for each one of the class instances. The variableValue dictionaries can be extended vertically in order to store values for the attributes that will be inserted in the classes. The type of each one of these objects is in accordance with the type of the variable it stands for. We must mention that obj3, obj6 and obj9 are original IdentityCard instances. For the rest of the objects that should have the type of Integer or String, this is achieved through the use of the Factory Design Pattern. The hierarchy used is depicted in figure 2 and consists of the following components.
• The Abstract Creator class serving no other purpose than defining the common interface of its subclasses.

• The Concrete Creator classes (IntegerCreator, StringCreator, BooleanCreator) with each one of them being responsible for the instantiation of objects having the appropriate type and containing data passed along by the user.

• The CreatorManager class which stores in a dictionary like the one presented in the following table the mapping between different types of variables and the corresponding ConcreteCreator objects.

<table>
<thead>
<tr>
<th>Type</th>
<th>Creator Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;int&quot;</td>
<td>IntegerCreator object</td>
</tr>
<tr>
<td>&quot;String&quot;</td>
<td>StringCreator object</td>
</tr>
<tr>
<td>&quot;Boolean&quot;</td>
<td>BooleanCreator object</td>
</tr>
</tbody>
</table>

• Finally the MetaclassManager class which through its insertValue() method is responsible for the creation of the appropriate data objects.

![Diagram](image_url)

**Fig. 2.** The Abstract Factory Pattern.

The tasks performed in the body of the insertValue() method are the following:

The CreatorManager’s find() method is first invoked with a String representing the variable type passed along as argument. The task of this method is to discover the ConcreteCreator instance corresponding to the received argument. That ConcreteCreator instance is then returned to the Metaclass Manager Object.

The create method of the ConcreteCreator instance is subsequently called. An argument representing the value the user wishes to store in the newly created object is passed along to it.

The create method creates an object of the appropriate type and returns it to the MetaclassManager object. After that, the storing of this object within the appropriate class instance takes place.
The insertion of a new attribute in a class using the proposed model could be achieved in the following way. 

First a new instance of the `Data` class should be created. Within its instance variables information about the newly inserted attribute (like its name and type as described before) should be stored. This data object should be stored in the static dictionary containing information about the class properties. The dictionary of our example case after the insertion of the two attributes should have the format presented in the following table.

**Table 5.** variableMap Dictionary after the insertion of variables.

```
<table>
<thead>
<tr>
<th>“AdultPerson”</th>
<th>data1</th>
<th>data2</th>
<th>data3</th>
<th>data4</th>
</tr>
</thead>
<tbody>
<tr>
<td>variableType:int</td>
<td>variableType: String</td>
<td>variableType: IdentityCard</td>
<td>variableType: String</td>
<td></td>
</tr>
<tr>
<td>variableName:height</td>
<td>variableName: Surname</td>
<td>variableName: ID</td>
<td>variableName: residence</td>
<td></td>
</tr>
<tr>
<td>“IdentityCard”</td>
<td>data1</td>
<td>data2</td>
<td>data3</td>
<td></td>
</tr>
<tr>
<td>variableType:int</td>
<td>variableType: String</td>
<td>variableType: String</td>
<td></td>
<td></td>
</tr>
<tr>
<td>variableName:codeNumber</td>
<td>variableName: dateOfIssue</td>
<td>variableName: authority</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

After that all the existing instances of the class should be updated using the before mentioned Abstract Factory mechanism. A new object having the appropriate type and containing the value passed along by the user should be created and stored in the instance dictionary. The format of the `AdultPerson` and `IdentityCard` existing instances after the insertion of the new attributes is depicted in the following tables.

**Table 6.** Instances of class `AdultPerson` after the insertion of variables.

```
| “height”      | obj1     | “height”      | obj4     | “height”      | obj7     |
| “surname”     | obj2     | “surname”     | obj5     | “surname”     | obj8     |
| “IdentityCard”| obj3     | “IdentityCard”| obj6     | “IdentityCard”| obj9     |
| “residence”   | obj16    | “residence”   | obj17    | “residence”   | obj18    |
```

**Table 7.** Instances of class `IdentityCard` after the insertion of variables.

```
| “codeNumber”  | obj10    | “codeNumber”  | obj12    | “codeNumber”  | obj14    |
| “dateOfIssue” | obj11    | “dateOfIssue” | obj13    | “dateOfIssue” | obj15    |
| “authority”   | obj19    | “authority”   | obj20    | “authority”   | obj21    |
```

Until this point we have shown the ability of our approach to support attribute addition. To achieve this in terms of versioning it should be used in combination with a mechanism monitoring the delta changes between the class versions (i.e. the attributes to be added or removed from the parent class version when a new one is inserted). This way the programmer would be able to use the version of his choice. Attribute removal in the context of versioning (where no physical removal of the attribute is accepted) could be handled using that monitoring mechanism too. The access to the
values of an attribute being removed from a certain version of a class should be denied when the programmer is using that version.

6 Discussion

The proposed technique could potentially support attribute addition and removal in terms of versioning. In this section we briefly present the advantages that it presents in conjunction to inheritance.

- Class explosion and complex hierarchies are avoided.
- No need for object migration.
- Attribute removal becomes an easier task

But there are also certain issues that emerge from its use and should be examined carefully. We let them as part of our future work:

- Substitutability and subtyping provided by inheritance are no longer supported.
- Attribute visibility needs to be considered.
- The behavioural part of a class that also undergoes modifications (because of methods addition and removal) must be treated seriously.
- The monitoring of behavioural changes could provide a means to control evolution in a unified way. This is the case because attribute addition could be simulated at the version control level by pairs of accessor methods for selecting (getter) and updating (setter) attributes. Consequently attribute and method modifications could be handled by the monitoring system in the same way.

7 References

2. Cooper, J.: Java Design Patterns - Addison Wesley, Reading, MA, 2000
Extracting implicit contracts from .NET libraries

Karine Arnout

Chair of Software Engineering – Bertrand Meyer
Swiss Federal Institute of Technology, Zürich
Karine.Arnout@inf.ethz.ch
http://se.inf.ethz.ch/people/arnout

Keywords: Design by Contract™, implicit contracts, .NET, metadata, Eiffel,
Contract Wizard, reuse, library design

Classification: 2 months’ work in 1st year of Ph.D. studies

Abstract. Do libraries written without explicit support for Design by Contract
in the language or the design methodology contain implicit contracts? To help
answer this question, we have performed an empirical study of some existing
non-contracted library classes from the .NET Collections library. This paper
reports on what we have found, and discusses whether the results could be used
to improve the design of the classes and make them easier to learn and use.

1 Introduction

Applying to reusable libraries the ideas of Design by Contract [1] mean equipping
each library class with precise specifications, or “contracts”, governing its interaction
with the library's clients. Contracts include the class invariant, stating general
consistency conditions to be maintained by every exported routine of the class, and,
for every routine, preconditions stating the clients' obligations, and postconditions
stating guarantees to the clients.

The experience of Eiffel libraries [2] shows that systematic application of these
principles leads to significant improvements in library design. A recent report by the
Software Engineering Institute [3] confirms that for components in general — not just
classes — the use of contracts appears to be a key condition of any effort to improve
“composability” and scale up the application of component-based technology.

Design by Contract as it has been applied so far, mostly in the Eiffel context, is not
an a posteriori addition to the design of a library; it is an integral part of the design
process. The resulting contract-rich library APIs are markedly different from more
traditional, contract-less designs. The difference is clear, for example, in a comparison
of two libraries that cover some of the same ground: the EiffelBase library, based on
Design by Contract, and the .NET Collections library.

Because the resulting styles appear so different, it is important to ask what happens
in a contract-less environment to the library properties that the Eiffel designer would
have expressed in preconditions, postconditions and class invariants. Is the contract-
rich style of Eiffel libraries an artifact of the presence of contract mechanisms in the
method, language and tools? Or are the contracts lurking anyway under the cover in
libraries for contract-less environments, whether suppressed or replaced by such ersatz techniques as explanations in user documentation or exceptions?

The search for implicit contracts is particularly interesting in the .NET context because the flexibility of the component model has enabled the development of a “Contract Wizard” [4] — a tool that enables a user to examine a compiled module (“assembly” in .NET), typically coming from a contract-less language such as C#, Visual Basic, C++, Cobol etc., and interactively add contracts to its classes and routines, producing a proxy assembly that is contracted as if it had been written in Eiffel, but calls the original. The Contract Wizard relies on the reflection capabilities provided in .NET by the metadata contained in every assembly. By nature, however, the Contract Wizard is only interesting if the conjecture discussed here holds: that in every contract-less library there are contracts trying to make themselves heard.

This paper is an empirical study exploring this conjecture. Using some classes of the .NET Collections library as our study target, we look for implicit contracts. Since the question is too subtle to allow automatic analysis by a software tool, we have examined the .NET classes manually, searching for the various forms that contracts can take when they cannot be expressed as contracts. The benefits we expect from this examination include:

- Assessing whether some “contracts” exist in any carefully-designed library, even if it has not been written in a language supporting contracts.
- Assessing the usefulness of contracting .NET libraries a posteriori.

Section 2 analyzes class ArrayList of the .NET Collections library. In light of the results, section 3 endeavors to build a contracted form of class ArrayList and compares it with its counterpart in EiffelBase, class ARRAY_LIST.

2 Analysis of Class ArrayList

Let us now examine class ArrayList, part of the core .NET library (mscorlib.dll) to find out whether .NET components are equipped with some form of contracts.

2.1 Implicit Class Invariants

The comments found in the .NET documentation for ArrayList reveal information about the class that falls into the category of class invariants in Design by Contract.

For example, one sentence of the documentation describing a possible constructor for ArrayList is: “The default initial capacity for an ArrayList is 16”, stating, almost as if in passing, that the capacity of the created object is positive. This property might at first appear to signal a postcondition. But we may notice that all three constructors of the class set the initial capacity of the list to a positive value. Since there is no other way to create an object of type ArrayList than to call one of these three constructors, such a global property of the class — which is satisfied after creation of the new instance and preserved by any further call to any other class routine — denotes a class invariant, which in Eiffel syntax would be expressed as:

\[
\begin{align*}
\text{invariant} & \quad \text{positive\_capacity:} \quad \text{capacity} \geq 0
\end{align*}
\]
Further exploration of the documentation helps deduce another class invariant for class ArrayList. The specification of the Count property mentions that “Count is always less than or equal to Capacity”. The self-assurance of this statement indicates that this property of the class always holds, suggesting that it is a class invariant; hence, in Eiffel syntax:

```eiffel
invariant
valid_count: count <= capacity
```

### 2.2 Implicit Routine Preconditions

Aside from implicit class invariants, the documentation also includes valuable information about implicit routine contracts.

For example, the specification of method Add of class ArrayList mentions that Add may throw an exception of type NotSupportedException, the condition being that the array list is read-only or has a fixed size. It means that the underlying code first ensures that the list on which the method is called is writable (not read-only) and extendible (does not have a fixed size) before adding value at the end of the list: this is a working requirement, namely a routine precondition in terms of Design by Contract. In Eiffel, one would write two preconditions of the form:

```eiffel
require
  writable: not is_read_only
  extendible: not is_fixed_size
```

This example — several similar ones can be found in the reference documentation — suggests a quite easy way to extract preconditions from .NET routines by using exception cases: simply read the exception condition (e.g. the array list is read-only) and take the opposite (required condition: !IsReadOnly) to obtain the underlying routine precondition: “writable: not is_read_only”, in Eiffel syntax.

### 2.3 Implicit Routine Postconditions

Let us now examine whether the documentation provided with the .NET Framework also reveals unwitting routine postconditions.

For instance, part of the specification of query IndexOf — whose signature is: `public virtual int IndexOf (object value);` — mentions that the return value is: “the zero-based index of the first occurrence of value within the entire ArrayList, if found; otherwise, -1”. The documentation clearly separates two cases: on the one hand, when the array list actually contains the value passed as argument to the routine, and on the other hand, when this value is not in the list. In the first case, the result is the index of the first occurrence of value found in the list: thus between zero (.NET lists are zero-based) and count (the number of elements in the list); in the second case, the result is -1. These properties we have just expressed informally result from the execution of query IndexOf, i.e. they fit in the category of routine postconditions described in Design by Contract. In Eiffel syntax, one would write:

```eiffel
ensure
  valid_result: contains (value) implies Result >= 0 and Result < count
  not_found: not contains (value) implies Result = -1
```

This example reveals that routine postconditions do exist in .NET libraries, although not explicitly expressed because of the lack of support from the underlying
environment. But contrary to preconditions which appear quite easily extractable in an automatic process, routine postconditions are likely to require a case by case examination involving a human-being, since they are mainly hidden behind remarks in the reference documentation, thus requiring the understanding of a person.

3 A Posteriori Addition of Contracts

The preceding analysis of .NET class ArrayList suggests that we construct a contracted variant, ARRAY_LIST, which has the same interface as the original ArrayList plus the elicited contracts. We will present a sketch of this class and compare it with EiffelBase’s ARRAYED_LIST.

3.1 A Contracted Form of Class ArrayList

Part of the interface of such a contracted class ARRAY_LIST would be similar to:

```plaintext
class interface ARRAY_LIST
create make
feature -- Initialization
make is
  -- Create and initialize.
  ensure
capacity_set: capacity = 16
writable: not is_read_only
extendible: not is_fixed_size
end
invariant
  positive_capacity: capacity >= 0
  valid_count: count <= capacity
end -- class ARRAY_LIST
```

3.2 The Benefits of Contracts

An a posteriori addition of contracts to .NET libraries would have many beneficial consequences. In particular, it would simplify the task of clients — comparing to the original ArrayList. As a matter of fact, the lack of contracts is very likely to lead to a very defensive programming style (see C# program code below), which is heavy and inefficient.

```csharp
ArrayList MyList = new ArrayList();
if( MyList.Capacity == 16 )
  if(!(MyList.IsReadOnly) && !(MyList.IsFixedSize))
    Index = MyList.Add( "Hello World!" );
```

On the contrary, a client using the contracted version of ArrayList, ARRAY_LIST, does not need to check that each routine’s execution has completed successfully: it simply relies on the contracts. Therefore the code required to perform the same task as before becomes lighter and thus more efficient:
```hpbc
create my_list.make
index := my_list.add ("Hello World !")
```

## 4 Conclusion

This discussion has examined some evidence from the .NET libraries relevant to our basic conjecture: do existing libraries designed without a clear notion of contract contain some “contracts” anyway?

This analysis provides initial support for the conjecture. The contracts are there, expressed in other forms. Preconditions find their way into exceptions; postconditions and class invariants into remarks scattered across the documentation, hence more difficult to extract automatically.

The analysis reported here provides a first step in a broader research plan, which we expect to expand in the following directions:

- Applying the same approach to other .NET and non-.NET libraries, such as C++ STL (a first informal look at [5] suggests that there are contracts lurking there too).
- Investigating more closely the patterns that help discover each type of contract — class invariants, routine preconditions and postconditions — to facilitate the work of programmers interested in adding contracts a posteriori to existing libraries, with a view to providing an interactive tool that would support this process.
- Turning the ISE Contract Wizard into a Web service to allow any programmers to contribute contracts to .NET components.

This area of research opens up the possibility of various generalizations of this work in a broad investigation of applications of Design by Contract.

## References

Graphical User Interfaces Composed of Plug-ins

Johannes Mayer

Department of Applied Information Processing and Department of Stochastics,
University of Ulm, D-89069 Ulm, Germany
jmayer@mathematik.uni-ulm.de
http://www.mathematik.uni-ulm.de/sai/jmayer/

Abstract. For programmers who are not involved in the development of a graphical user interface (GUI), it is usually not an easy task to adapt or extend the GUI. In this paper a possible solution being based on the plug-in concept is proposed which makes extensions of the GUI and independent development of GUI components much easier. Furthermore, GUIs designed that way can be easily extended with individual functionality for each user.

Keywords: GUI, plug-in, development method
Classification: Second year’s PhD work

1 Introduction

“Traditional” graphical user interface implementations have certain disadvantages for the developer:

1. The individual source files of a GUI usually have many lines of code, e.g. the source file with the main menu (and the corresponding commands). Therefore, the effort to add new functionality is greater for programmers who were not involved in the GUI development, since the appropriate place for the extension must be found.

2. The structure of the GUI (software) design is rather complex. This makes extensions or adaptions which involve changes in several parts of the GUI costly, since basic knowledge about the design must be acquired first.

3. An extension of the GUI cannot be done without changes in other modules, since every extension must be called — directly or indirectly — from a main GUI module. When the changed modules have already been tested, their tests are invalidated.

4. Third party extensions are currently only possible for some tasks through plug-ins (cf. e.g. [1, 4]).

Visual development tools, such as Sun’s Forte for Java and Borland’s JBuilder — in the case of Java —, do not really help to solve these problems. They only reduce the initial coding effort.
In the GeoStoch project a Java library for image analysis and image processing applying methods from spatial statistics and stochastic geometry is developed at the University of Ulm. The first version of the GUI developed for this library was a “traditional implementation” and thus, the above mentioned problems were experienced. The mathematicians who contributed new functionality to the library and wanted to add it to the GUI needed up to two days only for the (simple) GUI part. The reasons were that the GUI design was so complex, the modules had many lines of code, and the mathematicians were not involved in the development of the GUI. With the new plug-in-based GUI, it takes them by far less time than before to do this job.

Plug-in-based application development, as proposed in [3], can be tailored to the design of GUIs. This work shows how this contributes to the solution of the above mentioned problems.

After a short explanation of the plug-in concept in Section 2, various possible uses of plug-ins in the development of GUIs are presented in Section 3. In Section 4 related work is discussed. Finally, the contributions of this work are summarized and discussed in Section 5.

2 The Plug-in Concept

According to [3] (where the plug-in concept is explained in detail), a plug-in loader, a plug-in interface and concrete plug-ins are important parts of a plug-in-based application. Each concrete plug-in implements the plug-in interface which enables the communication between the application and the plug-ins. All available plug-ins are searched by the plug-in loader (at runtime) and made accessible.

Applying the plug-in concept to the development of applications, usually results in reduced complexity of the design as shown in Figure 1. The plug-

![Fig. 1. The basic structure of the design with (right) and without (left) employing plug-ins. A greater number of (plug-in) components (dark grey boxes) even increases the differences between the design complexities. Dependencies are depicted by solid lines. And the facade is illustrated through the dashed box.](image)

... in interface and the corresponding plug-in loader are the facade behind which
the concrete plug-ins are hidden. This usage of the Facade Pattern ([2]) helps simplifying the structure of the design. To develop further plug-ins, only the plug-in interface and the interfaces to the application’s library must be known.

3 GUI Plug-ins

The plug-in concept is now applied to the development of GUIs, to tackle the mentioned problems. A number of GUI plug-ins, i.e. plug-in types, from which a substantial part of graphical user interfaces — but clearly not everything — can be built, are presented in the following. For each GUI plug-in, a generic plug-in interface and generic plug-in loader, which acts in this case also as a GUI element, can be given to assist the programmer. So a programmer must only derive an own plug-in interface with custom methods and write the functionality, i.e. the concrete plug-ins. As illustrating example, the well-known email program Netscape Messenger, which is not explained here, is used throughout this section to make explications better imaginable.

3.1 View Plug-ins

Applications typically should be open to future document types. Possible document types of an email program are, for example, HTML text, plain text, and images. It cannot be foreseen how another type of document can be displayed. So, a (GUI) panel which displays documents of a certain type is a good candidate for a GUI plug-in. Two requirements must be fulfilled: There must be a superclass for all documents, also future ones. Furthermore, the view plug-ins must provide a so-called “voting method”. This method gets a document as parameter and returns whether this plug-in can display such a document. For each document which is to be displayed, the plug-ins are retrieved from the loader and all plug-ins are asked whether they can display such a document. If at least one plug-in can, one of them is chosen arbitrarily. Otherwise, a message is displayed, that such a document cannot be displayed.

3.2 Selection Plug-ins

Messenger offers different possibilities to retrieve mail, such as via the POP3 and IMAP protocol. The user has to select one of them and customize the selected protocol, e.g. an SSL connection may be used when the IMAP protocol is used, so this must be specified when IMAP is selected. For each protocol, a GUI plug-in (being a panel) can be designed which handles settings specific to this protocol. The name of the plug-ins, which can be accessed through a method call (to the plug-ins), are then displayed within a combo box, list box, or with radio buttons. When one gets selected, the panel implemented by this plug-in is displayed and the selected plug-in can be customized through user input (in the displayed panel). This may look like depicted in Figure 2a. Another possibility is that for the panel to be displayed a new tab within a tabbed panel is generated.
3.3 Preference Plug-ins

The first time, a user must enter information about his identity, such as name, email address, so that Messenger allows him to send mails. Therefore, a preference panel exists where such information can be specified. Furthermore, a view plug-in which displays HTML text may allow to customize colors and much more. This also requires a preference panel. Preference panels can easily be implemented as plug-ins. It is common, to use a tabbed panel for preferences. Each preference plug-in implements a panel which is displayed on an individual tab (cf. Fig. 2b). The name of this tab is the plug-in name which can be retrieved from the plug-in through a method call. Instead of a tabbed panel, a (possibly hierarchical) list box is also often used for preferences, as in the case of Messenger.

3.4 Command Plug-ins

Messenger also has icon buttons, which offer frequently needed functionality, such as writing a new mail, retrieve new mail, and so on. These actions and much more can also be accessed through the menu. Furthermore, when clicking with the right mouse button, a context menu is displayed, which varies depending on the position of the mouse pointer, e.g. over an email text or the list of messages. For all these purposes mentioned, plug-ins can be used to implement a command as reaction on the click. This command may be arbitrarily complex, and can also involve that new dialogs are displayed. The text of the menu entry as well as shortcuts and icons for buttons are provided by methods of the plug-ins. Different context menus correspond to separate plug-in types, i.e. plug-in interfaces. Through configuration files, the menus may be pre-configured, i.e. the menu items may be named, sorted and connected with plug-ins. Further plug-ins which are found at runtime are added to the preconfigured menus. Thus, customization of menus only involves editing the configuration files.
4 Related Work

Grail\(^1\) is an open source Internet browser implemented in the object-oriented programming language Python. On the website of this browser, the term “GUI plug-in” can be found. But GUI plug-ins are not explained there. This browser supports several plug-in types to extend the number of known file types, protocols, and HTML tags, and to add new preference panels. Preference plug-ins are here the first step towards a GUI composed of plug-ins.

Netscape plug-ins ([4]) are allowed to fill a part of the window or have a separate window to display certain MIME types. This is also a GUI plug-in, called view plug-in in the present paper.

5 Summary and Discussion

In the present paper a number of GUI plug-ins have been presented to ease the development of graphical user interfaces. Further possible GUI plug-in types are plug-ins for info dialogs, import and export filters, and maybe help.

The presented approach has proven useful for the development of the GUI for the GeoStoch library described earlier.

The plug-in concept helps to reduce the complexity of the GUI (software) design. Furthermore, modules which consist of many lines of code are not so common among plug-ins. The individual GUI plug-ins can be developed in parallel — even by third parties — only with the knowledge of the plug-in interface (and the library interface offered by the GUI application). Plug-ins do not require changes within other modules, since they are searched for at runtime.

However, it has to be pointed out, that the whole GUI cannot be modeled by plug-ins. Only components that are mutually independent and provide the same interface for the communication are candidates for plug-ins. For example, the main module of the GUI which uses the plug-ins, cannot be a plug-in. Another limitation of this approach is that it leads to a big number of GUI modules with not many lines of code. To keep the overlook, the modules have to be organized in relatively small packages.

References

2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)

\(^1\) http://grail.sourceforge.net/
Abstract. This paper presents the basic ideas of constructing a product line content & knowledge base on top of arbitrary configuration management system. Its need stems from the fact that evolving product lines arise a number of significant challenges, which can only be mastered with sound information base. The complexity of a product line engineering and its sheer number of additional artifacts and methods aggravates the intellectual control of a product line and its change over time. Management activities like product design, building of road maps or effort estimation become more difficult than for single product engineering and ask for more sophisticated information. We think that the problems can be solved with the explicit depiction of product line content & knowledge. Only then we can understand and control how a product is constituted from single software artifacts, what are their dependencies and relations, and how changes on shared assets affect the products in a product line. The product line content & knowledge base is a prerequisite for product line management services and processes, which makes it possible to manage the evolution of a product line and to control its consistency and completeness.

1 Introduction

Someone once said that, “a product line without a content & knowledge base is like a country without a map”. However interesting the experience of driving from A to B without a map might be, there can be no doubt that when the goal is to arrive at a destination as quickly as possible, a kind of map must be.

Similarly, if you are looking at a particular product in a product line the knowledge how the product is derived and where it is located in the content base is indispensable. Unfortunately, we are facing the situation that up to now there is neither a concept nor a system that provides this kind of map that guarantees the intellectual control of the products and their adhering assets in a product line.

Perhaps it is not very surprising that in the early years of product line engineering a product line content & knowledge base was not a metaphor. Today the situation is quite different; the sheer number of product line methods & principles (for example [2], [12], [5]) in the last years makes accurate and advanced knowledge base techniques of major importance. One answer to this problem in the realm of transportation is GPS (Global positioning system), in the realm of product line engineering it is a product line content & knowledge base.
Up until now there is no equivalent to the traditional single product configuration description like a make-file in the world of product line engineering. True enough; people mark their assets in configuration management systems as parts of a certain product in a product line, but the resulting configuration remains firmly in the paradigm of single product development. The world of product line engineering is quite different, here the distinction between individual products as isolated artifacts vanishes and the requirement is for a product line content & knowledge base to span multiple products, to cover a vast pool of relations and dependencies between them, and to control their change over time.

The problem has been recognized in configuration management for several years, but the methodology used to address it – variant branching, module building, or family oriented configuration descriptions - have only solved parts of the problem. The main purpose of variant branching is to support maintenance activities like corrective changes for eliminating errors or perfective changes for improving the quality of a product. Misunderstanding of this concept is misusing variant branching for defining products and is intermixing unconsciously maintenance structures with product structures. The product and maintenance variants are enmeshed in a clew that violates the principle of separation of concerns and hence is very hard to oversee. Modules in configuration management systems can help structuring a product by encapsulating components. Unfortunately, many CM systems do neither relate the modules nor version the modules (examples the concurrent version system (CVS), source integrity from MKS). Ergo the impact of module changes cannot be calculated in space and time what makes it impossible to control the evolution of a product line. Family descriptions like for example PROTEUS [10] that maps variabilities in PCL are static, of low abstraction, must be adapted manually to changes and do not reflect the dynamics of a product line. That is why a new technology is called for, and product line content & knowledge management provides an approach that marries and extends the best of several worlds. It uses existing configuration management systems for content management, extends them for a product line knowledge base, and introduces a more abstract, top down and language independent description of product lines. It is our firm conviction that product line content & knowledge base will become indispensable for tomorrows product line engineering like maps for travelers. And once they have become omnipresent, they will represent the standard GPS of product line engineering.

In this paper, we explore the ingredients of a content & knowledge base for product line engineering. Section 2 shows the core ideas for a content & knowledge base and sketches the essentials for a product line management system on top of it. Section 3 presents the currently available technology that can be used to transform the ideas in a real product line oriented content & knowledge base. In section 4 an abstract metamodel for the product line knowledge structure is presented that will be extended by variability mechanism introduced in section 5. Section 6 continues key structure of the technical realization and major visualization techniques. Section 7 summarizes the benefits of the product line content & knowledge base and section 8 closes with a short outlook about the current status and future activities.
2 The Idea

The basic idea is to manage product lines and product complexity based on explicit Product Line Content & Knowledge. The essential steps and considerations of our approach can be described as follows:

Fig. 1. Product Line Content & Knowledge Management System Layers

Product Line Repository (PLR). The product line repository consists of a formal content & knowledge model and a content & knowledge base that contain the instances of the model according to the formal model.

- Product Content & Knowledge Model. Content & knowledge are formalized in a model, which describes the structure, i.e. the constituting element types of a PL plus the knowledge about the internal structure of the elements and the relations/dependencies between them. A minimal model contains the basic PL-concepts: products, assets, features and decisions (see section 4). This model is customizable, i.e. it can be used to derive a company- or approach-specific PL-model like [12] or [2]. Although minimal it is sufficient to build up a PL-repository, which is independent from a given approach and can be used to derive a physical PL.

- Product Line Content & Knowledge Base. For a concrete product line the repository contains product instances according to the formal model. It consists of two parts, first the content base that is a physical storage for all product line artifacts, which are required to build a product, e.g. code, specifications, analysis & design documents. Second a product line knowledge base (PLKB) that contains the knowledge about the artifacts in the content base.

Product Line Management System (PLMS). The repository represents only a simple data-storage that provides primitive operations that deal with repository-items. The management of a PL requires more sophisticated services that support the consistent evolution of the PL, project management activities like cost- and effort estimations, or optimizations on the product line. We suggest to base these services on the primitive operations provided by the PLR and to implement them as a layer on top on the PLR (see Fig. 1). Due to its focus on PL-management issues we call it product line management system (PLMS).
Implementation. We suggest implementing the PLR and PLMS as a "Product Line based extension of CM", i.e. on top of existing CM systems. The PLMS provides basic services for the definition and creation of a product line, the creation and maintenance of products, assets and features, plus impact analysis, and repository and workspace management. PL processes like effort estimation will be built up on top of them.

We would like to mention that the basic principle - managing PLs on the basis of explicit PL content & knowledge - is already being applied successfully in manufacturing. For example using a sales system, a customer can configure a PC in its components & features according to his requirements like CPU (feature: clock rate = 400MhZ), hard-drive (feature: capacity = 40 GB) and memory (features: capacity = 256MB, access time = 8,5 ns). This is only possible because the sales system has access to explicit PL line content & knowledge, so that it can tell that certain features are incompatible or what a single feature will cost. Now this kind of information is exactly what we also need in PL management. So why should we do without this successful principle - the important question is: can we ever manage PLs without it?

However, we are absolutely aware that we can only transfer the basic ideas that are really applicable to software products. Actually, we will see in the further sections that the details of the approach are quite specific for software.

We have to mention again that this paper will concentrate on the product line content & knowledge base and does not explore the PL MS.

3 Ingredients of the Approach

We take reuse literally and therefore we identified all ingredients from state-of-the-art software-technologies & engineering that are required or helpful to make our approach work in detail. The main elements are:

Technical infrastructure. The technical infrastructure that is related to the PL content & knowledge repository are the CM systems that provide the basis for the repository and the integrated developer environments (IDE) that must be integrated in a product line management system
- **CM-systems** provide the basis for managing items in time. They offer the basic versioning services and hence form the bottom layer for the implementation of our approach where products and PLs are managed in time and space.
- **IDEs** are used to develop single products. In our approach they will keep this role without change. The only difference is that single products are instantiated with the help of the PLMS and the IDEs will access a product of a PL like a single product.

Engineering techniques. Most essential for PL are the variability concepts. Our relevant engineering techniques for their integration in a PL content & knowledge base are feature logic and weaving technologies.
- **Features & Feature Logic** are fundamental concepts in PL-Engineering [3] that we will use throughout the PLR & PLMS. Features are user valued functions, e.g. “validate the password of a user” [7] that characterize a product, and feature logic is used for assigning values to features and the definition how the different fea-
tures are combined in a single product [13]. This indicates that feature logic plays an important role in product instantiation.

- **Variabilities & Variation Points** are PL-concepts that describe the variability in the PL and how it is resolved in the product-instantiation process [9] [11].

- **Knowledge Representation** techniques are required for the formalization and modeling of the PL knowledge base. Actually there are already international standards available like the W3C standard “Topic Maps” that deal with this subject.

- **Weaving Techniques** are evolving techniques like frames [1] or aspect-orientation [6] that are already successfully used to weave variabilities into software artifacts. We think that they can be used to solve specific details in our approach, e.g. the implementation of single-product instantiation.

**Visualization infrastructure.** We think that management issues and daily PL-activities can only be executed efficiently when they are supported visually. Hence visualization will play an important role in the realization of our approach. Visualization concepts describe which PL-elements, -processes or -relations need to be visualized and how they can be presented in a meaningful way. Also visualization technology (e.g. web technology) will be required for the realization.

## 4 Product Line Knowledge Base

The PL knowledge base realizes the fundamental concepts for visualizing a PL, controlling its evolution and navigating through it. Today neither CM systems nor IDEs are providing the necessary capabilities for navigating in a PL or analyzing commonalities and variabilities between the different products. The minimal set of structural elements for representing the PL knowledge we found so far are products, assets, asset elements, features and decisions.

**Products.** Products are the most fundamental elements of a product line. They are the uppermost elements in a top down view on it. Products are configured by a set of assets that can be shared between different products (see Fig. 2).

**Assets.** Assets are self-contained software artifacts that can be versioned and are used for creation of products. They can exist in different structural variants different from the maintenance variants in the PL content base.

**Asset Elements.** Asset elements are the structural elements, which constitute an asset. For example, a function definition inside of a java source code file represents an asset element. Asset elements will be used for integrating variability on finer granularity level in order to increase the flexibility and expressiveness of PL knowledge base.

**Features.** Features are describing user as well as system functional requirements for asset and asset elements and are orthogonal to them because changing a feature does not affect assets, asset elements nor relations between them and vice versa. Features will be assigned by tags or similar techniques to a certain asset or asset element (see Fig. 2)
**Decisions.** Decisions in the context of a PL knowledge base are fixing the functional requirements that have to be fulfilled for a certain product. That means they resolve the features that will be set for the product.

![Diagram of Product Line and Repository]

**Fig. 2.** Products, Assets & Features

A product line contains an arbitrary number of products that will be constructed by at least one asset. The asset can be contained on the uppermost level of the repository or in another asset of type container. Assets as well as asset elements can be related by an asset relation that is either contained directly in the product line or again inside of an asset of type container.

![Diagram of Product Line Model]

**Fig. 3.** Product Line Model
5 Product Line Variability

Our PL knowledge base model (see Fig. 5) offers three different levels for integrating variability these are the product, asset, and asset element level. Variability on the product level just means that there can exist alternative products in the product line. In our knowledge model this case is out of scope, because it does not support reuse directly. On the asset level variability means that a certain asset can be shared between different products but not necessarily by all products. In the model (see Fig. 5) this is shown by the inheritance of an asset or asset relation from a variation point that indicates the variability. Exactly this is different from single product line engineering where no information about variable parts is available. Beside the situation that an asset can be optional there can also exist alternative forms of an asset (see Fig. 4). Variability on the asset element level means that the internal structure is variable. In the structural model an asset containing variant asset elements becomes a generic asset. Again variant asset elements are derived from variant point to indicate variability.

Variation Point. A variation point is a place in the PL knowledge base where variability will be integrated in the product line. Variable elements like an asset, asset relation, variant asset element are inherited from variation point in order to indicate that this element can be optional, it can exist an alternative, or values must be set for resolving the variability. In principle we are providing three different variability types in our PL knowledge model namely options, alternatives and range.

Variant Asset Element. A variant asset element is derived from asset element and variation point. Like the asset element the variant asset elements are the finest granular elements that can be managed by the PL knowledge base. They are provided for structuring the content of an asset in order to increase the flexibility and expressiveness of the PL knowledge base. In contrast to an asset the asset elements are not versionable and they reside always in a generic asset.

Generic Asset. Generic assets are derived from assets and differ from them by containing variant asset elements. For creating a product the generic assets must be instantiated what means that the variant asset elements must be resolved.

Fig. 4. Asset & Asset Element Variability

The basic elements for integrating variability in our PL knowledge base are assets, asset relation, generic assets, variation point and variant asset elements.
The depicted PL knowledge model (Fig. 5) shows the variability on the asset and asset element level. The variability handling for asset and asset element is different. Whereas variant asset elements depict explicitly the variability on the asset element level, an asset can always be variant elements because they are derived from variation point and hence there is no explicit model element for variant assets. The reason is that in a workspace assets are no longer variant elements.

Despite the pure existence of variant assets and asset elements we have to consider that variation can change over time. For example, variant asset can be transformed into generic assets or non-variant assets and vice versa (see Fig. 4).

**Variant Asset/Asset Transformation.** Variant assets will often contain common parts that will be changed in one asset variant. These changes must be reintegrated manually in other asset variants in order to keep the variant asset consistent. Therefore, it is advisable to separate this common part in a new and commonly used asset. This decreases the effect of common change in variant assets, and hence increases the reusability.

**Variant Asset/Generic Asset Transformation.** In some cases it is neither sensible nor possible to separate common parts in new assets. In that case a generic asset can help to indicate variant parts as variant asset elements. Now, like for the variant asset/asset transformation common part have to be changed only once and not for variant assets. Nevertheless, in contrast to variant asset/asset transformation there is major drawback, if a common part is changing, then the generic asset has to be re-instantiated for all products using it.

### 6 Benefits

The services that can be implemented based on our approach contribute to important issues in product-line evolution. We distinguish between management activities that do not affect the direct change of products and pure development activities for the modification and maintenance of products.

**Management.** The most relevant management activities that will be supported by our approach are project management activities, customer management and product design.
- **Project Management.** Project management parameters like estimations of effort, cost and time can be answered now quite simple. For example, we can easily identify important effort/cost drivers in our PL using an impact analysis (implemented in the PLMS) that tells us which features have a big change impact on the other assets in the PL. We think that “diving” into architectural details or - even more horrific - code details and doing estimations on that base is neither an efficient approach nor can it really answer the important questions.

- **Customer Management & Supply chain.** PL content & knowledge is now available for all participants in a customer oriented software production process. This has the potential to improve drastically the whole supply-chain between customer, marketing, project management, and development. For example, feature knowledge is now explicitly available for marketing and customers. When a customer requires a feature change, the related project effort is not a just a fantasy guess, but knowing the impact, it is estimated on a sound base. Hence the developers in the project have a realistic chance to realize the feature change within the planned effort. And - most important – the customer gets the product with the required change in the promised time.

- **Product design.** Our approach also helps to focus on designing a certain product like for example designing an automobile. Before a prototype or even the first release of a product will be manufactured, the product line management system virtually visualizes products and their constituting assets, features, decisions and relations between them. This allows to analyze, evaluate and improve product characteristics in advance, before the product itself will be instantiated and certain assets will be modified or developed completely new.

**Development.** The method supports a top down as well as a bottom up view on a product line. The bottom up view shows the different assets and their versions that are provided by the configuration management system, whereas the top down view depicts the different products, features and decisions inside of the PL content & knowledge base.

- **Product Evolution.** All kinds of development activities, e.g. asset changes are supported by specific services, which are implemented in the PLMS. Using the impact analysis a developer can calculate the impact of a planned asset change. This provides him with the exact information which other assets (including products) are affected by this change. Depending on the kind of change (e.g. an interface) the PLMS can also resolve automatically part (if not all) of the required update operations in the asset base. This kind of support leads to a well-defined, consistent and controllable evolution of the PL.

- **Product Creation.** Our approach also supports the creation of new products. The creation process consists of such steps as deciding which assets of the PL a product consists of and which features and respective feature-values the product has. As all the required information is an integral part of our PLR, the instantiation process, i.e. building the physical single product, can be executed automatically. This kind of intuitive and hence easy product creation and instantiation by configuring a new product from an overall PL-asset base using explicit PL-content & knowledge corresponds to the configuration of manufactured products that we described earlier in the idea section.
7 Outlook

The depicted approach covers all artifacts in the PL, integrates existing PL concepts, focuses on the relevant processes and issues during PL evolution, and provides a logical and consistent set of layers plus services that support these processes. We think that this is the first part but proper foundation for the realization of tools that support PL creation and evolution in practice. However, this is neither the end nor beginning of the end but the end of beginning to design a sound PL management system that starts with the construction of a sound PL content & knowledge base. There is still a lot of work to do when we look at the details, which have to be elaborated and cleared in each layer. Furthermore, we suspect that many questions will only arise at that point in time when such a tool is available and is used for the productive evolution of SPLs.

References

Supporting Component-based Development by Enriching the Traditional API

Marco Castaldi\textsuperscript{1} and Nathan D. Ryan\textsuperscript{2}

\textsuperscript{1} Dipartimento di Informatica, Università degli Studi dell’Aquila
Via Vetoio, 1, 67010, Coppito, L’Aquila, Italy
\texttt{castaldi@di.univaq.it}
\textsuperscript{2} Software Engineering Research Lab
Department of Computer Science, University of Colorado at Boulder
UCB 430, Boulder, CO, USA
\texttt{ryanmd@cs.colorado.edu}

Abstract. Component-based software, particularly distributed component-based software, makes heavy use of interfaces to define the interactions among components. Each of these interfaces is usually presented in the form of an API, as a specification of the capabilities of the relevant component. However, while the predominant API model captures the behavior of individual interface members, it neglects the interactions among those members. We propose an approach that incorporates such interactions into an interface specification, and provide an application of the approach.

1 Introduction

An Application Programming Interface—whether for an object, a protocol, or an architectural component—is normally expressed as, well, an interface, or in other words, a collection of functions (or methods) with perhaps some variables (or fields). These specifications are typically intended for automatic processing (as is the case with interfaces defined by systems such as the OMG IDL [7] and various programming languages such as Java. Often, each member of the API has an associated abstract, usually in the form of human-readable commentation, containing a description of the required behavior of each function and the intended semantics of each variable.

Such verbal descriptions are sufficient for human interpretation, and are certainly used to great effect. However, they are insufficient for automatic processing and a more formalized treatment of interfaces in general. Moreover, they are often the only source of description for interaction among interface members. For a component-based system, this “incompleteness” makes for more difficult development and validation.

2 A Problem with the Traditional API

Developers rely on interface member descriptions to understand just how to use an interface, since only in rare cases can the name of a function or variable convey
the appropriate semantics of its use, especially in relation to other members of the interface.

As a motivating example, consider a simple server interface that has two functions, \texttt{START} and \texttt{SUSPEND}. Intuitively, this interface definition indicates that \texttt{SUSPEND} should not be called before \texttt{START} (along with a couple of other obvious rules), or at least the implication exists. But without an accompanying description, we might not know the appropriate behavior. Does this sequence of calls cause an error, or throw an exception? Are either of these functions idempotent operations? Is an implementation of this interface in a “suspended” status before it is started for the first time? None of these questions are answered by the signatures of the two functions, unless some \textit{de facto} standard exists.

While the informal description may not be necessary for a function or variable as it stands alone, clearly there are some relationships (constraints) among entities of an interface that are \textit{only} captured by informal descriptions, even though they are likely simple enough to be represented formally.

2.1 Why Is This a Problem?

Simply stated, one can keep only so much in one’s head. Any developer would be able to understand a simple, isolated interface like the one in the example above. But what if, in a large component-based system, you have hundreds of interfaces, each with dozens of entities? Now the problem is significantly more complicated.

An additional problem arises when the notion of automation is introduced. With only informal descriptions, automatic enforcement of the relationships among the members of an interface is impossible. For more dynamically-oriented systems, such as a reconfigurable component-based system, the ability to automatically validate internal interface constraints that are possibly different or even unknown at the start of a run could be of great use.

3 A Solution for the Traditional API

The solution seems an obvious one: Formalize these informal descriptions, and include them as an integral part of the API. This has already been introduced in approaches such as Path Expressions [3] and Design by Contract [5], and in practical tools that allow one to easily specify validations within the code itself, such as APP for C [4] and the powerful assertion mechanism within Eiffel, among many others.

However, the relationships among the members of an interface should be specified within the interface on a global basis, separate from the realizations of those members. In other words, we would like an interface specification to include a section or sections (apart from the member declarations) for \textit{all} of the relationships among those members.
For the example in the previous section, a path expression such as

```
path START ; SUSPEND end
```

may somehow need to be expressed. For an interface without variables (or in other words, for an interface that makes no assumptions about the statefulness of an implementation), path expressions are often enough; however, if a variable such as MASTER is introduced to our example to capture a parent server's address, several additional constraints may also be required. Regardless, it is useful to capture these relationships as a whole, and bundle them with the interface in order to provide not only the appearance of an implementation but also the behavior required by the interface.

### 3.1 Why Is This a Solution?

Foremost, it allows the previously informal constraints to be formally expressed. This concept is nothing new, however; countless formalisms exist to describe such constraints.

But such an enriched interface has additional benefits. For example, if several functions have the same (possibly large) set of preconditions, these can be captured as a group and applied to each function as a group. As another example, relationships of implication among variables can be expressed, something difficult to do without a pair of "get" and "set" functions specific to each variable, which is something not always desirable.

However, the most important benefit is that a mechanism for validating the constraints of an interface can be automatically generated and applied to any realization of the interface, even apart from the implementation. This is especially convenient for distributed component-based systems, that conceptually deal exclusively with interfaces via a management protocol such as SNMP [1]. With such an interface (and proper coordination among managing components, of course), a requesting component could locally validate an action on another component, even without the receiving component needing to process the request. Enforcing constraints at the time of the operation but prior to execution (whether locally or remotely) not only isolates the components from some of the duties of error reporting but can possibly prevent components from entering an inconsistent state due to an invalid request. Furthermore, the same mechanism can be applied to all components that implement the interface.

### 4 A Practical Example

To demonstrate the problems with a traditional interface description and introduce a possible solution, we will extend the simple example above using LIRA (Light-weight Infrastructure for Reconfiguring Applications) [2], a system for remote management via dynamic and automatic reconfiguration of component-based applications in the context of large-scale distributed systems.
LIRA uses and extends the approach of network-management architectures and protocols, where an Agent directly controls the managed device and a Manager initiates the reconfigurations.

Here we further describe a LIRA Agent for a server in Siena (Scalable Internet Event-Notification Architecture) [6], a general publish/subscribe service that uses an innovative content-based routing mechanism.

4.1 LIRA

LIRA furnishes a general interface to act over components and applications. This interface is realized using a simple architecture and protocol, allowing one to "get" and "set" variables and to call functions.

The architecture uses three main actors: the Reconfiguration Agent, which contains the knowledge necessary to reconfigure the component (or system); the Component Description, which lists the functions and variables that the Agent exports in order to enable a reconfiguration of the component; and the Management Protocol, which allows Agents to communicate with each other.

To support a hierarchy of Agents, in which a higher-level Agent performs a reconfiguration at the application level using functions and variables exported by lower-level Agents, LIRA Agents can act in two different roles: an Actuator, which receives the reconfiguration order and performs the reconfiguration on the components; and a Manager, which gives orders to an Actuator regarding the reconfiguration to be performed on the components.

The Management Protocol is a simple TCP/IP connection-based protocol with six synchronous messages and one asynchronous message: GET/REPLY and SET/ACK (for "getting" and "setting" variables), CALL/RETURN (for function calls), and NOTIFY (to notify an Agent of some specific event.)

4.2 A Reconfiguration Agent for the Siena Server

The reconfiguration we want to perform on the Siena Server is to change the server's master, represented by the parameter MASTER, which binds the various Siena Servers together in order to create a Siena Network. If server $S_1$ is a master for servers $S_2$ and $S_3$, a message sent from a client attached to $S_2$ may reach another client attached to $S_2$, but only through $S_1$. If $S_1$ is down, $S_2$ and $S_3$ are isolated from each other.

To dynamically change a server's master, the LIRA Agent managing the server exports the variable MASTER. A portion of the Component Description (using a traditional interface) for the Siena Server follows:

```plaintext
Variables {
    MASTER {
        type = String;
        description = "The address as ' semp://host:port'."
        Cannot be changed without
```
losing subscriptions unless the server is suspended.
}
)
...
}
Functions {
START() {
    parameters {};
    return = "a message with the result of the start";
}
...
}

In the description field of MASTER, an important constraint is specified: that the master “cannot be changed without losing subscriptions unless the server is suspended”. (In fact the Siena Server has an internal mechanism that saves subscriptions when the server is suspended.)

Note that an automated manager can change the master and unknowingly lose information; this is a consequence of the informal interface specification, which does not allow an automated system to utilize crucial information furnished by the component developer. This is simply a demonstration of the problem with the traditional API, as discussed earlier.

A solution adopted for the LIRA Agent is to enrich the Component Description by specifying a list of Constraints and Actions. A Constraint is of the form

message => condition

where the operation associated with message is executed only if the boolean condition expression is satisfied. An Action is of the form

message (-&gt; result)+

where the boolean result expressions successively occur by sending the specified message.

For the Component Description of the Siena Server, we add the following:

Constraints {
    SET(MASTER, x)
        => STATUS == suspended;
}
Actions {
    CALL(SUSPEND, void)
        -&gt; STATUS == suspending
        -&gt; STATUS == suspended;
}
With this approach the constraints can be formally specified, allowing an automated manager to determine the conditions under which some messages may be sent, and what actions must be performed in order to satisfy those conditions.

5 Conclusion

Obviously, the example presented here is a simple one. Our intention was merely to give a concrete demonstration of the approach; how it is generalized and implemented has implications for wider applicability.

Regardless, by using this approach we can formalize constraints that normally are expressed verbally. When the relationships of an interface are formalized, not only can they be automatically validated, they can be validated across all implementations of the interface, independent of the implementations. Further, since the contract defined by the interface is more explicit, the behavior of components implementing the interface is better understood, thus assisting the development of component-based software.

References

Data Structures for Images on Java

Hans Braxmeier
Department of Applied Information Processing and Department of Stochastics
University of Ulm
Helmholtzstr. 18, D-89069 Ulm, Germany
hans.braxmeier@mathematik.uni-ulm.de
http://www.mathematik.uni-ulm.de/sai/braxmeier/

Abstract. Many data structures for images are limited to dimension two. Furthermore, common transformations for images are usually not supported (efficiently). In this paper, efficient Java data structures for images are discussed and their performance is compared.

Keywords: image processing, data structures, Java
Classification: first year of PhD studies

1 Introduction

For medical applications, e.g. in computer tomography, data are often given as three-dimensional images. Therefore, the image data structures supporting only two-dimensional images are not sufficient. For this reason data structures for images should be independent of the dimension or at least for dimension three. Furthermore, all commonly image transformations should be possible without great effort. Such image transformations are:

- **Subrange**: yields a rectangle subset of the image. This is important in processing parts of an image.

- **Reflection**: reflects the image at the origin. This is, e.g., necessary for the computation of the autocorrelation of an image.

- **Reduction of Dimensions**: yields an intersection of the image with a coordinate plane. This is used for analyzing sections of images, e.g., for medical purposes.

After an explanation and comparison of four possible image data structures in Section 2, the results of performance tests are compared in Section 3. In Section 4 related work is discussed and a summary of the results is given in Section 5.
2 Data Structures for Images

In this section, four data structures for images are explained. Their structure as well as the advantages and shortcomings of these data structures are discussed with respect to image representation.

In the following examples, assume the minimal values of all coordinates to be zero.

2.1 Multi-Dimensional Arrays in C

No matter how many dimensions are declared, a multi-dimensional array in C is one single array. The access to the element with indices x and y of a two-dimensional array is shown in Figure 1.

\[ \text{pixel} \ [x \times y \times \text{x-width}] \]

Fig 1. Two-dimensional arrays in C

Due to the addressing mechanism, the image transformations, presented in Section 1, cannot be implemented without running through each element of the array. But this addressing mechanism can easily be extended to higher dimensions.

2.2 Multi-Dimensional Arrays in Java

Multi-dimensional arrays in Java are arrays with references to arrays; the individual array components may themselves be references to arrays of different lengths as shown in Figure 2.

\[ \text{pixel} \ [x][y] \]

Fig 2. Two-dimensional arrays in Java

The main disadvantage of this addressing mechanism is the multiple indirection depending on the dimension of the image, which entails loss of performance. Also the data structure would be different in structure for different dimensions. Just as multi-dimensional arrays in C, this data structure does not (directly) support the mentioned transformations, except reflecting the image at the y-axis (reordering the pointers of the vertical array in Figure 2).
2.3 Index Arrays

For every dimension, this addressing mechanism uses an extra array with indices. The sizes of these arrays correspond to the number of elements of each dimension. A pixel can be addressed by summing up the indices of all index arrays as shown in Figure 3.

\[
\begin{array}{c|c}
\text{ind}[0] & \text{ind}[1] \\
\hline
x & + \\
\hline
\vdots & \vdots \\
\end{array}
\]

\[\begin{array}{c}
\text{pixel[\text{ind}[0][x] + \text{ind}[1][y]]}
\end{array}\]

\[\begin{array}{c}
\text{Fig 3. Two-dimensional indexed arrays}
\end{array}\]

This index data structure for images is independent of the dimension of the image. The more important aspect of this addressing mechanism is that the discussed transformations can be implemented without running through the elements. However, in contrast to the multi-dimensional array in C and Java, this data structure needs additional space for the index arrays.

2.4 Strides and Offset

As shown in Figure 4, an array with strides and offset is also based on a one-dimensional array.

\[\begin{array}{c}
\text{offset = 0}
\end{array}\]

\[\begin{array}{c}
\text{x-stride}
\end{array}\]

\[\begin{array}{c}
\text{y-stride}
\end{array}\]

\[\begin{array}{c}
\text{pixel [offset + x \times x-stride + y \times y-stride]}
\end{array}\]

\[\begin{array}{c}
\text{Fig 4. Two-dimensional arrays with Strides and Offset}
\end{array}\]

The index of the origin is given by the offset. Strides allow to process a whole image in a very fast and efficient way: if the index of a pixel is known, only one addition is necessary to calculate the index of the succeeding pixel, independent of the dimension. For the direct access one addition and one multiplication is necessary for each dimension. Index arrays need only as many additions as dimensions.

Similarly to the index arrays, this data structure is independent of the dimension. All the transformations can be implemented without running through the elements, and the overhead is minimal compared to multi-dimensional arrays in C or Java and less than the overhead of index arrays.
3 Performance Measurements

In this section, the performance of the presented data structures is evaluated. The attention is paid to index arrays and strided arrays. In both cases, different methods are implemented to retrieve and modify a pixel:

- Direct access to the used array
- Method access where the coordinates are given as integers
- Method access where the coordinates are given as an integer array

Every method uses an XOR operation between two images of the same size, which ensures that all pixels of both images are touched. The implementations are tested on images of different sizes: 2 MB, 20 MB, and 200 MB.

All experiments are conducted on a Pentium III@800MHz, 64KB L1, 512KB L2, 640MB, SuSE Linux 8.0, Kernel2.4.18, and on a Sun Enterprise 450, 4xUltra SPARC-II@400MHz, 1152 MB, Solaris 2.8. The programs are tested with the JavaHotSpot1.3.1 JVM and the gcc compiler on the SPARC and, additionally with the IBMJava1.3.0 JVM/JIT on the PC. Measurements are given in user times to compare the performance of the implementations as shown in Figures 5, 6 and 7.

![Performance Test Results on PC](image)

**Fig 5. Simple Arrays**

4 Related Work

Hoschek [1] implements the index data structure in Java. His main focus is fixed on rectangular dense and sparse multi-dimensional matrices. Accessing elements is done by set-and-get methods which cause loss of performance since method calls are expensive. For image processing, the sparse case is not relevant.
5 Summary and Conclusion

In this paper, different image operations and four data structures for images were analyzed. The performance of every implementation was tested.

Figure 5 shows that for both the gcc compiler and the IBM JVM/JIT, the time difference is nearly constant for every image size. The small difference is only a result of the compilation through the IBM JIT. Figure 5 shows also that it is advisable to use the IBM JVM/JIT on PCs instead of the Sun HotSpot JVM. As it can be seen in Figures 6 and 7 it is obviously that direct access using arrays with strides and offset is the fastest implementation.

Acknowledgment

I would like to thank Johannes Mayer for the helpful discussions and ideas.

References
