IARIA

PATTERNS 2013

The Fifth International Conferences on Pervasive Patterns and Applications

ISBN: 978-1-61208-276-9

May 27- June 1, 2013

Valencia, Spain

PATTERNS 2013 Editors

Alfred Zimmermann, Reutlingen University, Germany

PATTERNS 2013

Foreword

The Fifth International Conferences on Pervasive Patterns and Applications (PATTERNS 2013), held between May 27 and June 1, 2013 in Valencia, Spain, targeted the application of advanced patterns, at-large. In addition to support for patterns and pattern processing, special categories of patterns covering ubiquity, software, security, communications, discovery and decision were considered, as well as domain-oriented patterns. It is believed that patterns play an important role on cognition, automation, and service computation and orchestration areas. Antipatterns come as a normal output as needed lessons learned.

We take here the opportunity to warmly thank all the members of the PATTERNS 2013 Technical Program Committee, as well as the numerous reviewers. The creation of such a broad and high quality conference program would not have been possible without their involvement. We also kindly thank all the authors who dedicated much of their time and efforts to contribute to PATTERNS 2013. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

Also, this event could not have been a reality without the support of many individuals, organizations, and sponsors. We are grateful to the members of the PATTERNS 2013 organizing committee for their help in handling the logistics and for their work to make this professional meeting a success.

We hope that PATTERNS 2013 was a successful international forum for the exchange of ideas and results between academia and industry and for the promotion of progress in the field of pervasive patterns and applications.

We are convinced that the participants found the event useful and communications very open. We hope that Valencia, Spain provided a pleasant environment during the conference and everyone saved some time to explore this historic city.

PATTERNS 2013 Chairs:

PATTERNS General Chair

Jesus Tomas, Polytechnic University of Valencia, Spain

PATTERNS Advisory Chairs

Herwig Manaert, University of Antwerp, Belgium
Fritz Laux, Reutlingen University, Germany

PATTERNS 2013

Committee

PATTERNS General Chair

Jesus Tomas, Polytechnic University of Valencia, Spain

PATTERNS Advisory Chairs

Herwig Manaert, University of Antwerp, Belgium
Fritz Laux, Reutlingen University, Germany
Michal Zemlicka, Charles University - Prague, Czech Republic
Alfred Zimmermann, Reutlingen University, Germany

PATTERNS Research/Industry Chairs

Teemu Kanstren, VTT, Finland
Guenter Neumann, DFKI (Deutsches Forschungszentrum fuer Kuenstliche Intelligenz GmbH), Germany
Markus Goldstein, DFKI (German Research Center for Artificial Intelligence GmbH), Germany
Zhenzhen Ye, iBasis, Inc., Burlington, USA
Cornelia Graf, CURE - Center for Usability Research & Engineering, Austria
René Reiners, Fraunhofer FIT - Sankt Augustin, Germany

PATTERNS 2013 Technical Program Committee

Ina Suryani Ab Rahim, Pensyarah University, Malaysia
Junia Anacleto, Federal University of Sao Carlos, Brazil
Andreas S. Andreou, Cyprus University of Technology - Limassol, Cyprus
Senén Barro, University of Santiago de Compostela, Spain
Rémi Bastide, University Champollion / IHCS - IRIT, France
Bernhard Bauer, University of Augsburg, Germany
Noureddine Belkhatir , University of Grenoble, France
Hatem Ben Sta, Université de Tunis - El Manar, Tunisia
Silvia Biasotti, Consiglio Nazionale delle Ricerche, Italy
Félix Biscarri, University of Seville, Spain
Jonathan Blackledge, Loughborough University, UK
Manfred Broy, Technical University Munich, Germany
Michaela Bunke, University of Bremen, Germany João Pascoal Faria, University of Porto, Portugal
M. Emre Celebi, Louisiana State University in Shreveport, USA
Jian Chang, Bournemouth University, UK
William Cheng-Chung Chu(朱正忠), Tunghai University, Taiwan
Bernard Coulette, Université de Toulouse 2, France
Karl Cox, University of Brighton, UK
Jean-Charles Créput, Université de Technologie de Belfort-Montbéliard, France

Table of Contents

DAO Dispatcher Pattern: A Robust Design of the Data Access Layer

Pavel Micka
Faculty of Electrical Engineering
Czech Technical University in Prague
Technicka 2, Prague, Czech Republic
mickapa1@fel.cvut.cz

Zdenek Kouba
Faculty of Electrical Engineering
Czech Technical University in Prague
Technicka 2, Prague, Czech Republic
kouba@fel.cvut.cz

***Abstract*—Designing modern software has to respect the necessary requirement of easy maintainability of the software in the future. The structure of the developed software must be logical and easy to comprehend. This is why software designers tend to reusing well-established software design patterns. This paper deals with a novel design pattern aimed at accessing data typically stored in a database. Data access is a cornerstone of all modern enterprise computer systems. Hence, it is crucial to design it with many aspects in mind – testability, reusability, replaceability and many others. Not respecting these principles may cause defective architecture of the upper layer of the product, or even make it impossible to deliver the product in time and/or in required quality. This paper compares several widely used data access designs and presents a novel, robust, cheap to adopt and evolutionary approach convenient for strongly typed object oriented programming languages. The proposed approach makes it possible to exchange different data access implementations or enhance the existing ones even in runtime of the program.**

***Keywords*—data-access; software design; pattern; object-oriented; architecture; software evolution**

I. Introduction

Software design pattern can be understood as a well-established and reusable technique of designing certain software artifacts that are frequently present in various particular forms in a number of software projects. This paper introduces a novel software pattern aimed at accessing database objects. Its basic idea is motivated by the work of other authors that is briefly surveyed in section III.

Modern computer systems have to deal with increasing volume of data. According to the Moore's law [1], the number of transistors in integrated circuits doubles approximately every 18 months and as the computational capacity grows, grows also the volume of data processed. Hence, the systems and their storage engines (databases), became also increasingly complex in past decades.

To deal with the complexity of application (business) logic, object oriented programming was introduced. Nevertheless, the data itself is usually stored in conventional relational databases, which creates impedance mismatch between the data storage and the program itself. Object-relational technologies and frameworks, such as Java persistence API [2], were developed in order to minimize the differences and provide transparent persistence to the programmer.

Such frameworks help to separate the principal concern of business objects behavior (business logic) from the infrastructural concern of how business object's data is retrieved/stored from/to the database and make business objects free from this infrastructural aspect by delegating it to specialized data access objects (DAO). Thus, DAOs intermediate information exchange between business objects and the database. To facilitate the replacement of the particular mapping technology and to encapsulate database queries, data access objects layer pattern was devised. There are many possible implementations that differ mainly in their reusability, testability, architecture/design purity and by the means they provide to support software evolution.

II. Basic principles

In order to compare various implementations/designs, we use the following criteria, which describe their conformity with the object oriented paradigm and applicability in non-trivial and evolving software systems. Although these principles are well known within the software engineering community, we will describe them in next paragraphs in order to avoid possible misunderstandings stemming from different definitions.

Encapsulation – the data access module should be well encapsulated to hide implementations details (see the Protected Variations GRASP pattern). Minor changes in implementations should never affect interface of DAO module.

Do not repeat yourself (DRY principle) [3] – the code of the module itself as well as code needed for the usage of the module should not be duplicated (or even multiplicated). This constraint reduces the number of scripts needed to test the application and reduces the possibility of regression defects caused by modifying only one of the copies of the respective code.

You aint gonna need it (YAGNI principle) [4] – the user (programmer) should never be forced to create classes or structures, which he doesnt need at the moment. Also the module itself should fit the actual needs of the programmer, not needs of some feature, which may not be implemented yet. The YAGNI principle reduces code bloat and hence saves money, which would be otherwise spent to create, debug and test superficial features.

Single responsibility principle – every class/structure of the program should have only one responsibility. Hence, if

```
@Entity
@NamedQueries(
    {@NamedQuery(
        name =Book.Q_FIND_BY_TITLE,
        query = "SELECT b FROM Book b
            WHERE b.title = :title")})
public class Book {<CODE>}
```

Fig. 1. JPA Named Query code example

properly encapsulated, it can be easily replaced by another implementation. As the code is focused and has only limited set of dependencies, classes respecting this principle are easier to test.

Reusability – the generic DAO functionality should be reusable, project independent and possibly modularized. Reusability reduces costs of the module, because the generic core code is written and tested only once and developers shared by several projects have to be familiar with only one DAO implementation.

Testability – the testability criterion states that the DAO functionality should be controllable by external testing scripts, its behaviour should be observable and the number of scripts needed for its testing should be minimized.

III. CONVENTIONAL APPROACHES

A. *Generated queries (no DAO)*

The most straightforward implementation of data access is not to use the data access layer at all and hardcode the functionality into business objects/service layer. As an example may serve *Java Persistence API Named queries*[5].

The named queries are Strings written in JPA query language, which are passed to the framework as class annotations (metadata) as shown in Figure 1. The programmer invokes these queries by their name. The named queries are usually generated by integrated development environment and do not posses any means for structural parameterization (i.e. name of a columns passed as a query parameter).

Thanks to its support by development environments, named queries are convenient for rapid development of a product prototype.

Nevertheless, they are enormously inappropriate for usage in production. The main disadvantage stems from the above-mentioned fact that their structure cannot be parametrized. This means that for every entity and its every property a new named query has to be created, what results in massive code duplication and additional testing expenses. In addition, all queries are bound to entities, so they are not reusable at all in other non-related projects. Such a design violates encapsulation and single responsibility principle, because the data-access technology is invoked directly from business logic. This makes business logic dependent on the data access technology, although it should be technology agnostic, and when the data access implementation is changed, the business logic will have to be reprogrammed and retested as well.

B. *Simple data access object*

To encapsulate the technology used, data access objects may be introduced. In their simplest form [6][7] there is one DAO for every business object in the domain that provides all the functionality needed. This design can be seen as encapsulation of generated queries.

Although it solves the main architectural drawback of generated queries, there exists one DAO class per each business object class and it causes immense code duplication, which makes the objects hard to test and maintain. This is why this approach is not suitable for practical usage and the scientific community gone on in investigating more sophisticated methods.

C. *Generic data access object*

The above mentioned code duplication can be resolved using generic data access object (Figure 2) that contains methods common for all entities, such as *findById*, *remove*, *getAll*, in their generic form (i.e. property and names are passed as parameters when necessary).

GenericDAO class is highly cohesive and radically reduces code redundancy and thus improves testability as opposed to the generated queries design. When combined with templating features of the given programming language (e.g. templates, generics), then the class also provides type safe access to the underlying repository.

Generic DAO still possess some design drawbacks. First of all, there is a question, where to place specific DAO functionality. For example, let us have the query looking for all books that are currently in the borrowed state. One option might be to place all specific queries into GenericDAO class, which will result in creating poorly cohesive class with responsibilities over several entities. The second solution might be to create a new specific DAO (e.g. BookDAO) for all persistent business objects, when needed. Although this second option is better, it still does not satisfy another requirement: the data access should support software evolution. Let us suppose that software, which has been developed for a long time, uses GenericDAO in conjunction with specific DAOs. Then a new requirement appears, which implies a specific functionality of getById method for the Book entity. Again there are two options, how to realize the new behaviour. The first one requires sub-classing of the GenericDAO and overriding the getById method so that it behaves differently for Book entity (testing the type of the entity by *instanceof* operator). The

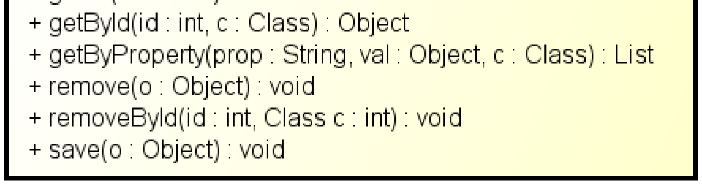

GenericDAO
+ getAll(c : Class) : List
+ getById(id : int, c : Class) : Object
+ getByProperty(prop : String, val : Object, c : Class) : List
+ remove(o : Object) : void
+ removeById(id : int, Class c : int) : void
+ save(o : Object) : void

Fig. 2. UML Class diagram of GenericDAO class

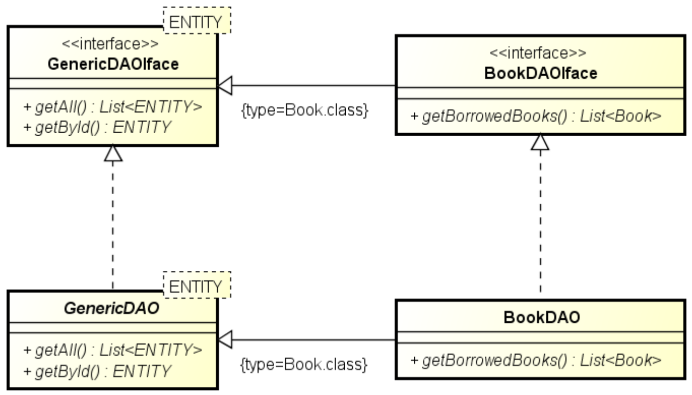


Fig. 3. UML Class diagram of Generic superclass DAO

second option is to put this modified method into specific DAO. While the first approach will later or soon result in spaghetti code — code with enormously tangled structure, usually massively using branching and loop statements — when more modifying functionality will be added, the second approach requires rewriting all calls of the respective method of GenericDAO to specific DAOs one. Thus, none of these two options is satisfactory.

D. Generic superclass

To make possible the code evolution, GenericDAO (see Figure 3) can be modelled as a common (abstract) superclass of all DAO objects. Rosko [8] presents a very similar approach, but he isusing factory to instantiate particular DAOs. Because there will be a mandatory implementation of a specific DAO for every entity, the situation described above will never happen. Software evolution is well supported, because the programmer can consistently override the generic implementation in the respective specific subclass, easily add new specific data access methods and last but not least, the implementation can be easily protected by interfaces and reused in other projects.

Although the generic superclass DAO solves most of the design flaws of the previously discussed implementations, it creates a new one. According to our experience with development of enterprise systems, for the most of entities the generic method implementation is sufficient and also many entities do not require any additional specific methods. And as the specific DAO classes are mandatory, the design results in many classes with empty specification, which is prepared only for possible future changes. This is premature generality, which strongly violates the YAGNI principle.

IV. DAO Dispatcher pattern

To overcome violation of YAGNI, we introduce a new DAO Dispatcher pattern, which combines benefits of both simple Generic DAO and Generic superclass DAO.

A. The overall structure

The pattern (see Figure 5) uses internally Generic DAO class mentioned in the previous chapter to handle all generic requests at one place. If necessary, additional data access methods can be defined in specific DAO classes derived w.r.t. inheritance from the more generic one. If present, the specific data access object mandatorily implements all generic methods, forwarding the call to the respective method of the generic DAO class by default. The signatures of the corresponding methods of specific and generic DAOs are identical except the following point. As the generic DAO class processes data objects of various types, its methods have to have the class parameter that determines the exact type of the processed data. This class parameter is not necessary in case of specific DAO classes. In this case, the type of processed data is implicitly determined by the type of the specific DAO class itself. As a common facade for all generic calls, a new class GenericDAODispatcher was introduced.

B. Registry/DAO Dispatcher class

The registry object is the core of this pattern. It implements the GenericDAO interface and all generic calls should be always made though the registry object. When no specific data access object is registered, it simply delegates the call to the GenericDAO, otherwise the DAO specific to the given class is called.

This mediator makes it possible to introduce the specific functionality without any changes to the code (only the project configuration) just in time, or even to hotswap DAO implementations at runtime.

C. Abstract specific DAO

The abstract specific DAO is a common ancessor of all specific DAO implementations. As it was stated in the previous chapter, usually, the generic functionality is sufficient for most use cases. This is why the default functionality of the specific DAO just routes the query to the generic DAO implementation.

When need for a new DAO functionality occurs, the programmer subclasses the abstract specific DAO and creates only the new method — writes only what he needs. The modification of the generic behaviour is analogous and requires only overriding of the respective method.

D. User interaction

From the user's point of view, there are four major types of interaction with the framework. The interactions are shown as UML transactional diagram in Figure 4.

The first interaction type depicts a call of *getAll* method on a specific DAO type – BookDAO. As it was already stated, the programmer typically does not need to override the existing generic functionality, but wants to extend it. Hence, when the dispatcher is called and the call is delegated to the BookDAO, it only propagates the call further to the generic DAO implementation.

On the contrary, when the getAll functionality is overriden in the specific DAO, than only the delegation from dispatcher is made and the call is executed by the specific method itself (second interaction type).

When the programmer does not specify any specific DAO for the Book entity, than the dispatcher calls directly the GenericDAO in order to provide the default common functionality. This interaction type, third in the image, is predominant for

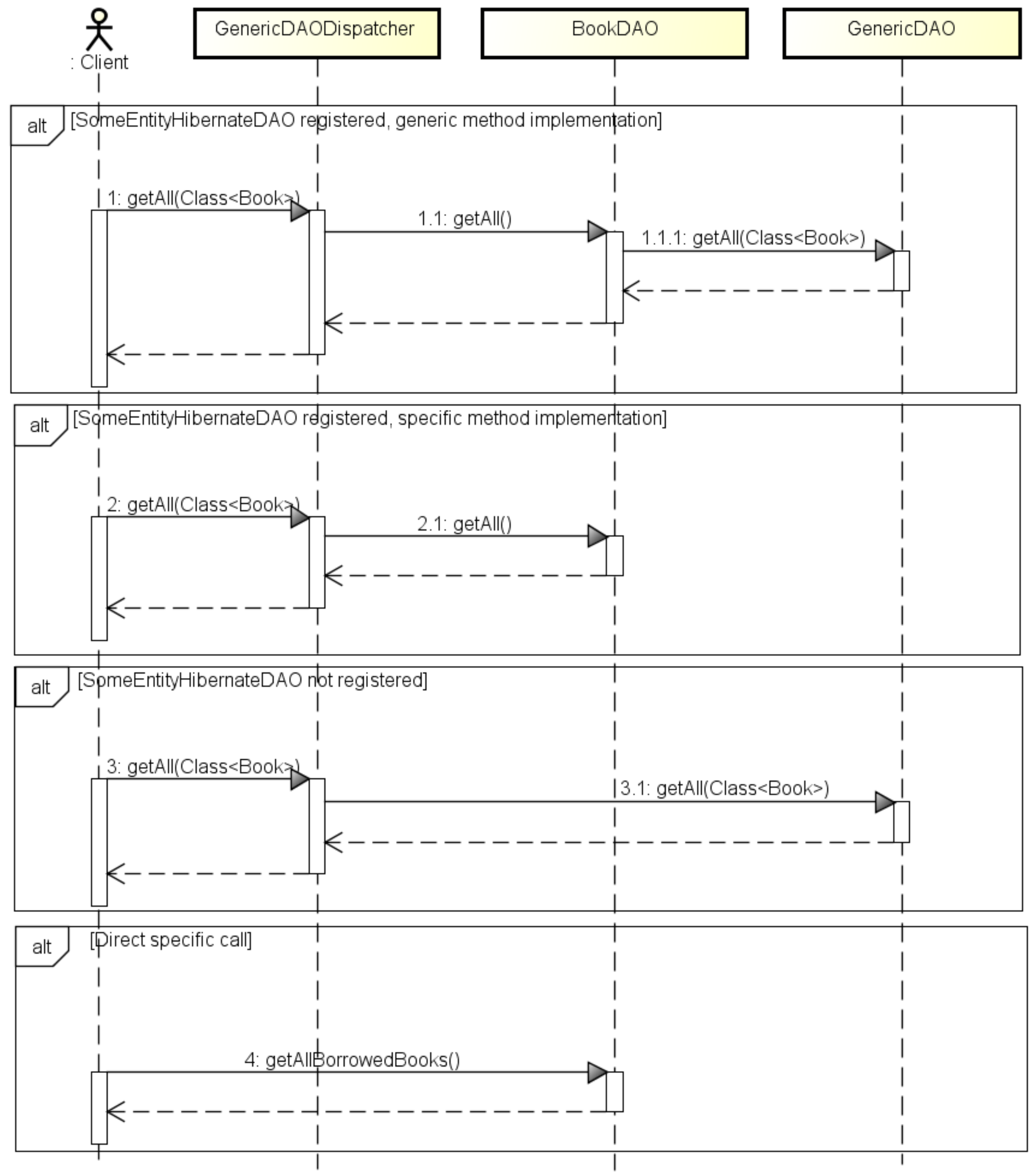


Fig. 4. UML diagram of DAO Dispatcher pattern with Hibernate (JPA) data access implementation

new or not fully fledged applications, where data handling does not have any exceptions from the general flow.

The fourth interaction shows direct invocation of a method specific to the given entity. This method cannot be called through the dispatcher, because the generic interface does not contain its contract and there is naturally no generic implementation in the GenericDAO class. For this reasons the specific functionality calls are always made directly.

E. Advantages of the pattern

The above described structure of the pattern in conjunction with the designed interaction flow provide significant benefits for the end programmer (programmer which creates a system with DAO Dispatcher pattern already implemented as a submodule).

Namely the programmer does not need to write and test the generic DAO functionality, which is already embedded in the submodule.

Also he does not need to prematurely determine, whether the given entity will need any special handling when being stored or retrieved from the database. The framework allows the programmer to make this decision just in time – when it is really needed.

Last but not least, the pattern structure is highly dynamic and flexible. The overriding functionality can be easily plugged-in using configuration of the application, because this change does not require any modifications of the source code of the project itself. The behaviour of the application can be modified/extended even at runtime.

V. Related patterns

Our novel DAO Dispatcher design pattern uses and extends several commonly known patterns and principles already described by other works. To allow the reader to understand our approach in detail, this section lists these patterns/principals and depicts their usage, similarities and how they relate to DAO Dispatcher classes.

Singleton [9] – all presented classes in the DAO Dispatcher pattern are singletons by their nature. It means that there exists at most one instance of each of them. The reason is that they are either stateless or their state has the application global scope. An example is the Dispatcher class fulfilling the role of a registry in terms of the Registry pattern described bellow.

Ports and adapters (Hexagonal architecture) [10] – In a nutshell, the hexagonal architecture dictates a design of a component in such a way that it communicates with external entities through an API consisting of technology specific ports that are easily adaptable. This makes the core of the component independent on the specific technologies used by the given project and thus the component core is easily portable to other environments.

In particular, DAO Dispatcher pattern as a whole can be described as a single module with clearly defined boundary (interface/ports), which can be accessed through technology specific adapters, when needed. The Dispatcher pattern API also provides means for setting the implementation of the data source (eg. *JPA EntityManagerFactory*), which can be easily exchanged by a mock implementation for testing purposes.

For example: while the core of the module is stable and provides means for direct (binary) calls, in some cases it might be useful to create a serializing adapter, which will transform the input/output objects into JSON, XML or to any other transport format and back. Because the adapting functionality is located externally from the core, it is still possible to test it directly using ordinary unit tests.

Registry – The dispatcher class is an exact realization of the registry pattern as described by Martin Fowler in [11]. The fundamental principle of this pattern is an associative container enabling service providers to register their services in this container using an (typically unique) identifier. Later on, the clients may look up and use the registered services using these identifiers.

Such an architecture is very flexible. From the perspective of the proposed DAO Dispatcher pattern, it is important that the registry allows for on-the-fly inferencing of the appropriate Specific DAO implementation.

For example: if the DAO object for the Novel entity is requested but not available then the more generic Book DAO object shall be used rather than falling back to the purely generic DAO.

Inversion of Control [12] – In conventional programming, the programmer defines the control flow from the beginning to the end himself. However, if he applies a generic framework to the specific problem domain, he usually designs and implements a plugin to that framework. In such a case, he cannot influence the control flow that is determined by the framework itself. Programmer only fills in additional or overriding functionality using pre-prepared join points. In other words, the code of the programmer has the role of a library, while the control flow (in our case of the query evaluation call) is controlled by the framework (DAO Dispatcher pattern).

VI. Application

In typical software systems, the maintenance and enhancement expenses outweigh the costs of development [13], hence it is crucial to use sufficiently robust components during its construction. The pattern is in particular convenient for applications in enterprise systems, which usually evolve continuously and require means for specialization of generic use cases (and respective data access procedures).

Since, as was already described, the DAO layer forms a well encapsulated module, it can be easily interchanged with another implementation. This might be very useful property, when developing a generic system, which will be used by many different customers, each of whom can use completely different data source.

VII. Future work

Although the pattern is intended to be used in strongly typed languages, some dynamic properties might be also employed in future. Mainly, the DAO Dispatcher (registry) class code is in its static form highly duplicate, because each call of the registry only has to delegate the functionality to the appropriate implementation. However, this duplication is well hidden from the user of the module, it would be convenient to use reflection abilities of the host language in order to simplify the registry implementation and reduce the number of lines of code needed to extend the core module functionality.

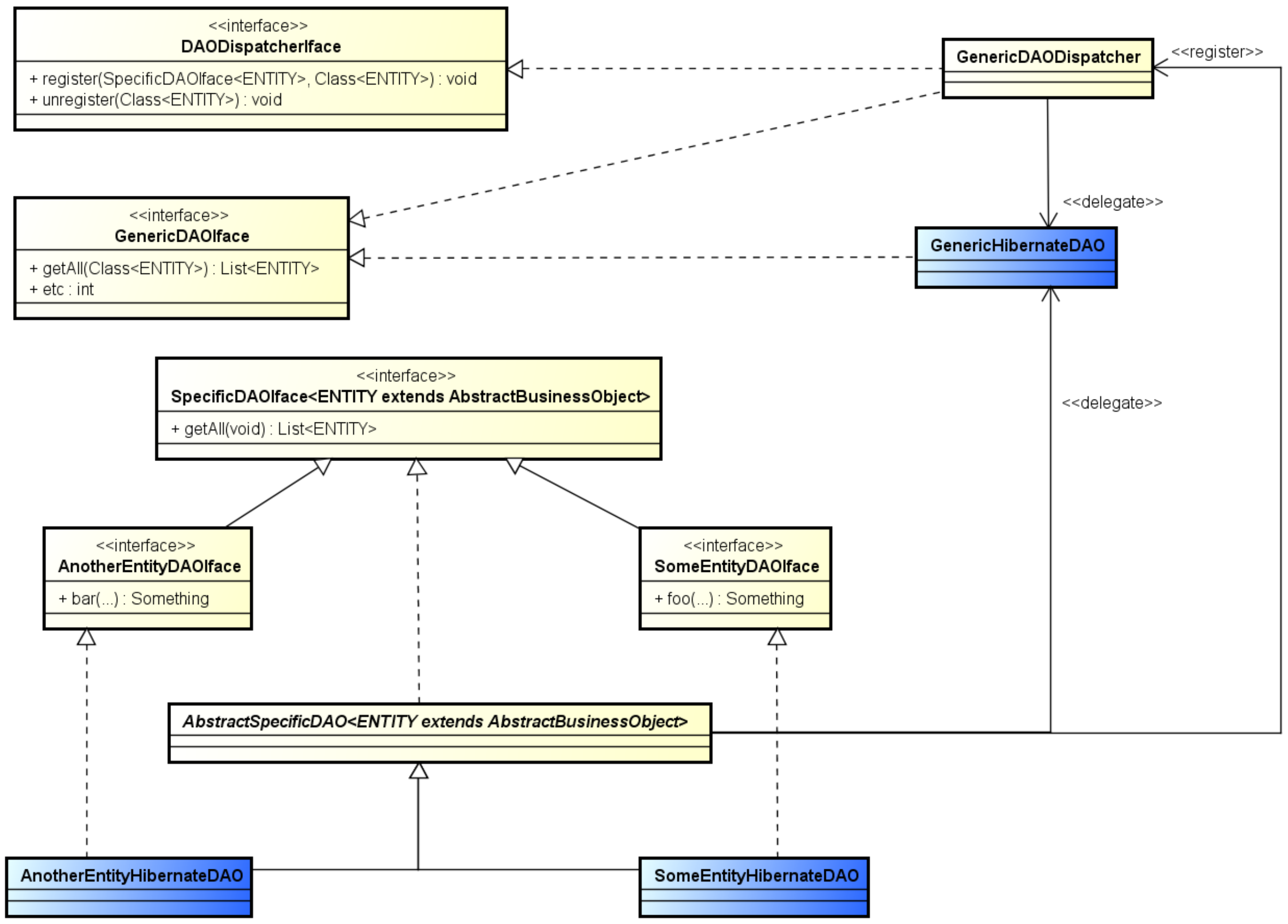
The extensibility of the core of the module can be also improved by application of the visitor pattern [9]. Each visitor, accepted by the registry, will provide new generic functionality of the core module and, when needed, also overriding functionalities for specific DAO implementations.

VIII. Conclusion

This paper proposed a novel approach to robust data access design, which overcomes imperfections of common implementations. Mainly it is well testable, reusable, honoures the single responsibility and YAGNI principles and last but not least it supports software evolution.

Because the main logic of the module is well hidden behind a facade, the programmer working with it can be familiar only with the general principle, generic DAO interface and the Abstract specific DAO class. This makes the implementation easy to use and cheap to adopt.

However the reference implementation written in Java, using Spring framework [14] for dependency injection, it provides sufficient means for porting the code to other programming languages and clearly proves that the pattern can be easily implemented in strongly typed language, some language specific improvements might be also employed. The reference implementation can be found at https://kbss.felk.cvut.cz/web/portal/dao-dispatcher.



<<interface>>
DAODispatcherIface
+ register(SpecificDAOIface<ENTITY>, Class<ENTITY>) : void
+ unregister(Class<ENTITY>) : void
GenericDAODispatcher
<<register>>
<<interface>>
GenericDAOIface
+ getAll(Class<ENTITY>) : List<ENTITY>
+ etc : int
<<delegate>>
GenericHibernateDAO
<<interface>>
SpecificDAOIface<ENTITY extends AbstractBusinessObject>
+ getAll(void) : List<ENTITY>
<<delegate>>
<<interface>>
AnotherEntityDAOIface
+ bar(...) : Something
<<interface>>
SomeEntityDAOIface
+ foo(...) : Something
AbstractSpecificDAO<ENTITY extends AbstractBusinessObject>
AnotherEntityHibernateDAO
SomeEntityHibernateDAO

Fig. 5. UML Class diagram of DAO Dispatcher pattern with Hibernate (JPA) data access implementation

ACKNOWLEDGMENT

This work has been supported by the grant by the grant of the Czech Technical University in Prague No. SGS13/204/OHK3/3T/13 — Effective solving of engineering problems using semantic technologies.

REFERENCES

[1] G. Moore, Cramming More Components Onto Integrated Circuits. McGraw-Hill, 1965.

[2] Oracle. (2013) Java persistence api. Retrieved: 12/03/2013. [Online]. Available: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[3] A. Hunt and D. Thomas, The Pragmatic Programmer: From Journeyman to Master. Pearson Education, 1999.

[4] C. Zannier, H. Erdogmus, and L. Lindstrom, Extreme Programming and Agile Methods - XP/Agile Universe 2004, ser. 4th Conference on Extreme Programming and Agile Methods, Calgary, Canada, August 15-18, 2004. Proceedings. Springer, 2004.

[5] Oracle. (2013) Oracle fusion middleware kodo developers guide for jpa/jdo, chapter 10. jpa query. Retrieved: 12/03/2013. [Online]. Available: http://docs.oracle.com/html/E24396_01/ejb3_overview_query.html#ejb3_overview_query_named

[6] M. Berger. (2005) Data access object pattern. Retrieved: 11/03/2013. [Online]. Available: http://max.berger.name/research/silenus/print/dao.pdf

[7] D. Matic, D. Burotac, and H. Kegalj, "Data access architecture in object oriented applications using design patterns," Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference, 2004. MELECON 2004., vol. 2, pp. 595–598, 2004. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1347000

[8] Z. Rosko and M. Konecki, "Dynamic data access object design pattern," Information and intelligent systems CECIIS 2008 : 19th International conference, 2008. [Online]. Available: http://www.ceciis.foi.hr/app/index.php/ceciis/2008/paper/view/41

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, ser. Addison-Wesley Professional Computing Series. Pearson Education, 2004.

[10] A. Cockburn. (2005) The pattern: Ports and adapters (”object structural”). Retrieved: 11/03/2013. [Online]. Available: http://alistair.cockburn.us/Hexagonal+architecture

[11] M. Fowler, Patterns of Enterprise Application Architecture, ser. The Addison-Wesley Signature Series. Addison-Wesley, 2003.

[12] ——. (2005) Inversionofcontrol. Retrieved: 11/03/2013. [Online]. Available: http://martinfowler.com/bliki/InversionOfControl.html

[13] R. L. Glass, Ed., Frequently Forgotten Fundamental Facts about Software Engineering, ser. IEEE Software, IEEE, May/June 2001.

[14] SpringSource. (2013) Spring framework. Retrieved: 12/03/2013. [Online]. Available: http://www.springsource.org/spring-framework

Android Passive MVC: a Novel Architecture Model for Android Application Development

Karina Sokolova*†, Marc Lemercier*
*University of Technology of Troyes, France
{karina.sokolova, marc.lemercier}@utt.fr

Ludovic Garcia†
†EUTECH SSII, France
{k.sokolova, l.garcia}@eutech-ssii.com

***Abstract*—Nowadays the demand for mobile application development is very high. To be competitive, a mobile application should be cost-effective and be of good quality. The architecture choice is important to ensure the quality of the application over time and to reduce development time. Two main leaders are very represented on the mobile market: Apple (iOS) and Google (Android). The iOS development is based on the Model-View-Controller design pattern and is well structured. The Android system does not require any model: the architecture choice and the application quality highly depends on the developer experience. Heterogeneous solutions slow down the developer, while the one known design pattern could not only boost development time, but improve the maintainability, extensibility and performance of the application. In this work, we investigate some widely used architectural design patterns and propose a unified architecture model adapted to Android development. We provide the implementation example and test the efficiency of the proposed architecture by implementing it on a real application.**

Keywords—Smart mobile devices (smartphones, tablets); design patterns; Model-View-Controller; Android architecture model; Android passive MVC.

I. Introduction

The mobile market has grown rapidly in recent years. Many enterprises feel the need to be present on mobile markets and propose their services with mobile applications. Compared to computer programs, mobile applications often have limited functionalities, shorter shelf life and lower price. New applications should be developed fast to be cost-effective and updated often to keep users interested. The quality of the application should not be neglected, as mobile users are very pernickety and competition is stiff. Architecture choice remains important for mobile applications to ensure quality: mobile applications as well as other systems could be complex and evolve over time.

The demand for smartphone application development is very high especially for the two market leaders: Apple (iOS) and Google (Android). Multi-platform solutions, such as Phone-Gap, Rhodes Rhomobile and Titanium Appcelerator reduce development time, as one application is developed for several platforms [1], but have limited possibilities – often requiring native plug-ins. Multi-platform solutions also add complexity to the native code (e.g. web layer) that decreases the performance of the application. The support of non-native solutions could be abandoned. Native solutions enable use of all the platform's options with better performance and lighter code, therefore developers often choose native software development kits (SDK).

The iOS SDK imposes the Model-View-Controller (MVC) design pattern for the iOS application development [2]. Android requires no particular architecture [3] – developers choose a suitable architecture for their applications that is especially difficult for less experienced developers. Complex applications that do not follow any architecture can end as a big ball of mud code: incomprehensible and unmaintainable [4]. Suitable architecture can improve three non-functional requirements of software structural quality: extensibility, maintainability and performance. A defined architecture could additionally reduce the complexity of the code, simplify the documentation and facilitate collaboration work [5].

Android development books and tutorials are mostly focused on Android SDK technical details and user interface design. Only a few works have been dedicated to the Android application architecture, while the Android community identify an architecture as an important part of successful system design and development. Developers open many discussions about suitable Android architecture on forums, blogs and groups.

In this work, we provide an overview of some widely used architectural patterns and propose an MVC-based architecture particularly adapted to the Android system. Android Passive MVC simplifies the development work giving the guidelines and solutions for common Android tasks enabling the creation of less complex, high-performance, extendable and maintainable applications.

The remainder of the paper is organized as follows: the second section presents several architectural patterns used in software development. Section 3 presents briefly the Android SDK and existing difficulties in adapting one known architecture to Android. In Section 4, we propose an adaptation of the MVC design pattern to the Android environment and provide an implementation example. Section 5 evaluates the Android Passive MVC model and Section 6 concludes this work and presents some perspectives.

II. Fundamental Design Patterns

We present four popular MVx-based design patterns in historical sequence. These patterns are widely used in desktop and web applications development. If mobile development assimilates similar design, developers moving from other systems could take advantage of their knowledge. Different components and existing variants of models are included in the description.

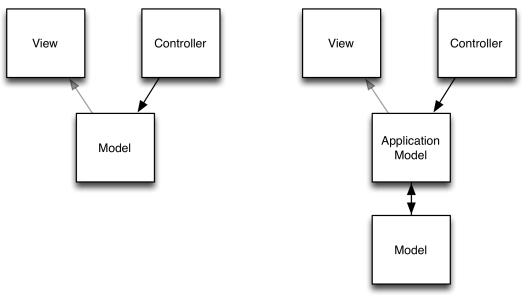


Fig. 1. Classic MVC and Application Model MVC

A. *Model-View-Controller (MVC)*

Presented in 1978 [6], Model-View-Controller is the oldest design pattern and has been successfully applied for many systems since it's creation [7], [8]. The goal of this model is to separate business logic from presentation logic. The business logic modifications should not affect the presentation logic and vice versa [6]. MVC consists of three main components: *Model*, *View* and *Controller*. The *Model* represents a data to be displayed on the screen. More generally, *Model* is a Domain model that contains the business logic, data to be manipulated and data access objects. The *View* is a visual component on the screen, such as a button. The *Controller* handles events from user actions and communicates with the *Model*. The *View* and the *Controller* depend on the *Model*, but the *Model* is completely independent. The design pattern states that all *Views* should have a single *Controller*, but one *Controller* can be shared by several *Views*.

MVC model have three varieties: Classic MVC, Passive Model MVC and Application Model MVC (AM-MVC). The scheme of two MVC model varieties is shown in Figure 1. The Classic MVC is shown on the left and the AM-MVC is shown on the right.

In all variants, *Controller* handles events and communicates directly with a *Model* that is indicated by a black arrow. On the Classic MVC the *Model* processes data and notifies the *View*. The *View* handles messages from the *Model* and updates the screen using the data received from the *Model*. This behaviour is implemented using the Observer pattern (grey arrow in Figure 1). Conversely, the communication between the *Model* and the *View* in Passive Model MVC is done exclusively via the *Controller*. The *Model* notifies *Controller* which then notifies *View* and finally the *View* makes changes on the screen [9]. The AM-MVC is an improved Classic MVC with an additional component. The *Application Model* component was added for the presentation logic (e.g. change the screen colour if the value is greater than 4) that was often added to View or Controller previously and makes a bridge between the *Model* and the *View-Controller* couples.

B. *Model-View-Presenter (MVP)*

The Model-View-Presenter was introduced in 1996 as an MVC adaptation for the modern needs of event-driven systems [10]. The model consists of three components: *Model*, *View* and *Presenter*. In this model, the *View* represents a full screen and it handles events from the user actions. The *Presenter* is responsible of the presentation logic. The *Model* is a Domain model.

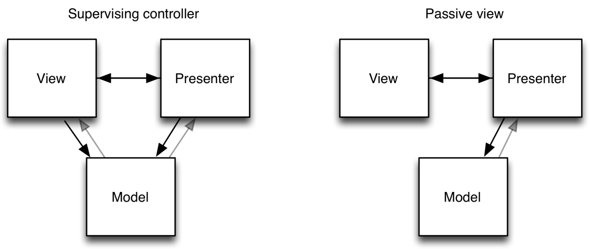


Fig. 2. Supervising controller and Passive view

There are two types of MVP: Supervising controller and Passive view. Both models are shown in Figure 2. The Supervising controller uses the Observer pattern for the communication between *Model* and *View*. The *View* can interact directly with the *Model* to save the data if there is no change to be made on the screen. Otherwise, the communication between the *View* and the *Model* is made via the *Presenter*. Interaction between *View* and *Model* of the Passive View MVP is done exclusively via *Presenter* [10].

C. *Hierarchical-Model-View-Controller (HMVC)*

The Hierarchical-Model-View-Controller was first introduced in 2000 as an Classic MVC adaptation for Java programming [11]. This model takes into account the hierarchical nature of Java graphical interface components: the main window frame contains panes that contain components. The authors propose to create layered architecture for the screen with Classic MVC triads for each layer communicating with each other by controllers. The HMVC model is shown in Figure 3.

Thereby the child controller intercepts methods from its view. If a view of the upper hierarchy (parent view) needs to be changed, the child component informs the parent controller, which makes the changes. The communication between layers is made exclusively via controllers.

D. *Model-View-ViewModel (MVVM)*

Model-View-ViewModel is another model to separate the presentation and business logic. The ViewModel is a linking component between View and Model. This design pattern is mainly used in Microsoft systems [12]. The realization of this model is done with binding between components [13]. The binding is not supported in Android by default but could be implemented using the very recent Android-binding framework.

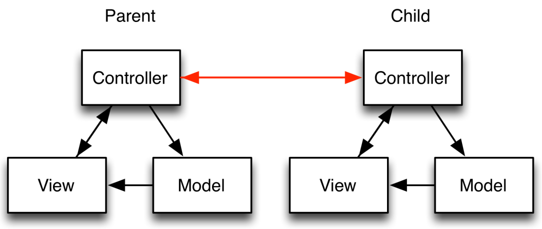


Fig. 3. Hierarchical-Model-View-Controller

As stated in [14], a good basic model should not use any additional framework and should be easily implemented with original components, therefore this model is not dealt with in the paper.

III. Android application development experience

Android is a Linux-based open source operation system designed for mobile devices. Android was first presented by Google in 2007 and in spite of huge competition from Apple has been the leading smartphone platform since 2010. Google continues to work on the system systematically integrating new features and correcting bugs. Many manufacturers of smartphones and tablets adopted this open-source solution; the National Security Agency and NASA also choose Android for their projects.

Android applications are mainly written in Java using the Android SDK [15]. The code is compiled to be executed on the Dalvic virtual machine on a smartphone. Additionally, developers can use the Native Development Kit (NDK) to add a C or C++ written code referred to as native. NDK allows more advanced features and better performance, however, the complexity of the code increases with the quantity of native code [16] – Google suggested minimizing the use of this kit.

Four principal components of Android SDK are used in Android Application development: Activity, Service, Content provider and Broadcast receiver. Activity is a main component of Android applications created while the application that is also the entry point to the application is open. Many Activities can exist in the application but only one is active at a time. The service works on the background of an application permitting an execution of long tasks (e.g. file download). When the application is closed, unlike Activity, the work of the Service is not interrupted. The Content provider component gives access to the local data stored in SQLite databases. The Broadcast receiver is a messaging system that enables communication inside the application and between multiple Android applications installed on the phone.

Activity causes major difficulties in implementing the known architecture: is it a *View*, a *Controlle*r, a *Presenter* or none of them? Some developers say Android actually imposes the MVC model where the layout.xml (file, defining the layout of the screen) is a *View*, Activity component is a *Controller* and the rest is a *Model*. This proposition is not really the MVC: layout.xml only defines what the screen looks like, but button actions, text information and other presentation logic are usually placed in Activity. Therefore, Activity handles events as *Controller* and manages the visualization as *View*, replacing the *View-Controller*. It leads to the creation of a heavy and complex Activity class [17]. Huge classes that have many responsibilities (event handler, presentation logic, etc.) violate the Single Responsibility Principle of Object Oriented Programming and could be hard to understand, test and extend [18].

Other developers place Activity as a *View* of MVC creating *Controller* apart. This solution works for simple applications where one Activity represents one visual block, while Activity usually manages several *Views*: main screen, menu, dialogue box, etc. In complex visual applications Activity becomes heavy; *View* components are strongly linked to each other and are not reusable. *Controller* will be either complex or divided into parts by a number of embedded Activity *Views* that go against the MVC statement of one *Controller*, one *View*. Replacing MVP *View* with Activity can cause similar problems.

Some developers observed that Android have predefined *View*s as ViewFlipper. It brings another solution where the Activity became a *Controller* and *Views* are created apart. Solution seems more adaptable to Android as event interception in Activity can be defined in layout.xml but actually creates problems similar to previous implementation: many *Views* make the only *Controller* (Activity) complex. *Views* are reusable but the corresponding *Controller* should always be added to the new Activity using the *View*. To delete or modify the *View* developer should modify the full Activity. Final application is complex and hard to maintain.

Even if MVC and MVP architectures seem suitable for Android developments they are not intuitive to implement. A new architecture should be easily implemented with Android-specific components, such as Activity. The implementation of the model should improve the application and code quality. More precisely, the model should reduce the complexity of an application, clarify the code and improve extensibility. The coupling between components should be weak to avoid the modification of other components if one is modified. Modules should be reusable [14], [18]. A mobile phone has a limited memory and a garbage collector could have unexpected behaviour therefore the creation of unnecessary objects should be avoided. Finally, objects remaining in the memory should be lightweight [16].

IV. Novel Design Pattern for Android Platform

The first part of the section explains the novel architecture for Android application development we named Android Passive MVC. The second part of the section presents a simple example implementing the Android Passive MVC. The third part of the section recommends an architecture of the business logic of the application – the Model. Android applications have similar needs: internal database management and access, web service access and reusable components use. Clear main architecture of business logic could also simplify the development process.

A. Android Passive MVC Presentation

We have decided to base our architecture on the MVC model, as MVC is well-known and widely used in desktop and web systems as well as in iOS mobile development. Developers coming from other systems would be able to easily appropriate the Android development architecture.

Activity is an inevitable component of the Android application. Previous experience of the Android community shows Activity does not fit well on the MVC model, while it seems to be well adapted to developers' needs. We decided to create MVC model around Activity making the Activity the fourth component. We can also think of Activity as a main screen (parent) controller in HMVC model.

Observer-observable pattern is relevant for multi-screen systems but only one screen is active at a time in Android application. This pattern supposes keeping in memory Views and Models that appear heavy for the mobile environment, therefore we chose the Passive Model MVC as a base for our architecture.

In our model, Activity becomes an intermediate component between the Views and the Controllers, thereby Controllers take the event handling responsibility and the Views take the presentation logic making the Activity lightweight. The scheme of the Android Passive MVC model is shown in Figure 4. The grey dashed arrows show the interaction via Android native methods. Black arrows indicate direct calls and grey arrows represent listener events.

The Activity is like a screen controller. The starting Activity creates a link between a View and a corresponding Controller to make them communicate directly. The communication between Controllers is made via Activity.

The Views are the interface components, such as a form, a menu or a list of elements. View components contain methods that allow the setting or obtaining of data from the user interface on Controller demand, the setting of event listeners on visual components and the modification of visual components (set errors, change colours, etc.). Views are independent and do not communicate.

The Controller handles events from the user action (e.g. button click), calls necessary methods from the Model and then notifies the View to be updated on Model response. The Controllers are independent from one another and do not communicate directly.

This solution makes Activity lightweight by moving all event handlers to Controllers and interface management to Views. Views and Controllers created on demand avoid unnecessary objects, saving memory. Developers can easily modify or remove application components by only modifying or deleting the corresponding view-controller couple. Application can be extended with view-controller couples. The Model is independent from the View, the Controller and the Activity. The user interface could be replaced without any impact on Model thereby the maintainability of the application is high.

We perform the communication between Activity, Controller and Model via message listeners implemented via interfaces as proposed by [19]. Figure 5 shows the Android Passive MVC implementation diagram. Listeners increase the performance of the application and create a weak coupling between components that improve maintainability.

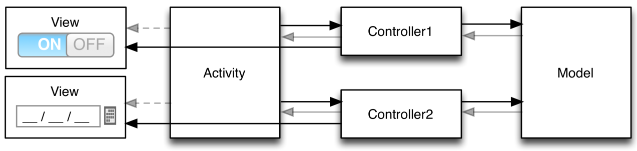


Fig. 4. Android Passive MVC

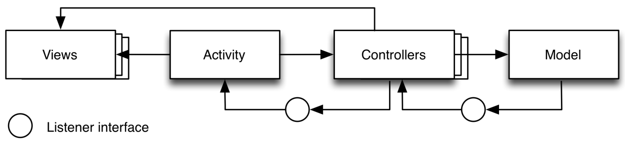


Fig. 5. Android Passive MVC implementation

B. Android Passive MVC Implementation

This section presents an implementation example of communication between Android Passive MVC components. This implementation is suitable for the new manually created Activities. Some predefined Activities, especially from third-party libraries, will possibly not fit the implementation. We created a login screen with a classic login form to enter the login and password; if the login is successful the user goes to the welcome page, otherwise the error message appears.

The example contains two Activities: Login Activity managing the login page and Welcome Activity for the welcome page. The login form is managed by Login View and Login Controller. Login Activity implements the LoginControllerListener interface to be able to receive calls from the Login Controller. The schema is shown in Figure 6.

Login View contains methods for obtaining login and password (getters), methods to set button listener and methods to set errors. Login Controller handles event from the login form implementing the onClickListener; while the button is pressed Controller launches simple verifications and calls the model. If login is successful, the answer goes back to the Login Activity, which opens a welcome screen. To simplify the example we do not include the model, but the communication between the Controller and the Model can be implemented similarly. A full code example can be found on [20].

C. Android Domain Model

The Model of Android Passive MVC is a Domain Model containing business methods, web service call methods, database access objects, reusable methods and data model objects.

A Domain Model architecture should include components usually used in Android applications, such as Database manager, Web services manager and Business logic. Those components should be independent, as the architecture should be adaptable. Reusable components should be also separated. The basic model architecture is shown in Figure 7.

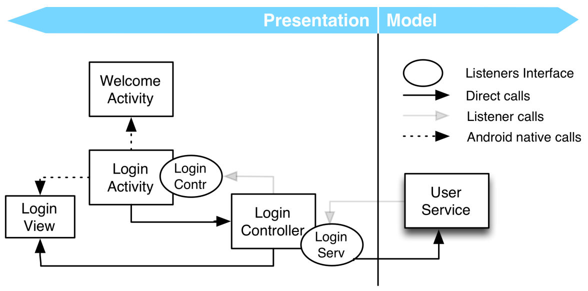


Fig. 6. Login implementation example

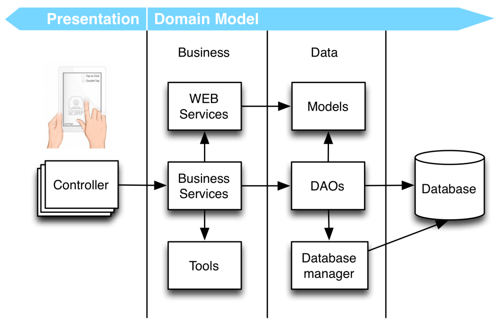


Fig. 7. Domain Model Architecture

The architecture of Domain Model proposed in this document is inspired by 3-tier architecture that separates the presentation, the business and the data access layers [21].

The business layer of our model regroups objects and methods that use web services, business services and reusable tools. Business services contain business logic. If an application works via Internet as well as locally, all necessary verifications are done in Business services, which calls corresponding methods. The communication between a presentation and a domain model layer are made via Business services.

The data layer contains Models, Data Access Objects (DAO) and Database Manager. DAO and Model are the implementation of the Data Access Object pattern. Model contains data being persisted in the database or retrieved by web services calls. Model is a simple Plain Old Java Object (POJO) that contains only variables and their getter and setter methods. Data is manipulated and transferred through the application using those lightweight objects that are often called Data Transfer Object (DTO).

Persistence methods are organized in DAOs. DAO contains methods that enable the data in a database to be saved, deleted, updated and retrieved. Even if Android proposes an abstraction on the data access level with Content Provider, DAO simplifies the code of the application. The DAO design pattern creates a weak coupling between components and use a lightweight Model object instead of an Android cursor object in the application. DAO can also be used for the data stored in XML or text files. Good practice is to make DAO accessible via interfaces. It allows DAO modification (for example the change of SQLite to XML storage) without any change in Business services, which increases maintainability.

Database manager is in charge of the database creation. Database manager exists only if SQlite database is used by the application. It stores the name of the database, and of its tables and methods to be able to create, drop, open and close the database.

This architecture regroups logically similar methods together, increases cohesion. High cohesion facilitates the maintainability of the software. The final code of the application could be organized in packages by architectural components: Activities, Views, Controllers, Business Services, Tools, Web Services, Model, DAOs and Database. It gives the clear structure of an application and limits the package number. Additional packages could be created for interfaces, parsers (e.g. XML, JSON) and constants.

V. Architecture Evaluation

We evaluated the architecture in two steps. First, we ensured that the architecture fit the lists of code quality criteria proposed by [14], [16]. Second, we ask an Android developer to rewrite one of his latest applications using Android Passive MVC, compare results and give feedback regarding the model.

A. *Code quality*

The evaluation of our architecture is based on the following code quality evaluation criteria: techniques used, maintainability, extensibility, reusability and performance.

The use of standard platform techniques is important for the model: the support of third-party functionalities could be interrupted making implementation of the model impossible. The Android Passive MVC could be implemented using Android SDK without any additional libraries.

A high-quality application has high maintainability and extensibility: codes have weak coupling between components, easy code suppression possibility and high testability. The Passive MVC architecture ensures high maintainability. Clear separation between presentation and business logic simplifies testability of components. Weak coupling between all layers is carried out via listeners. One component (ex. interface, DAO, web service) could be replaced or modified without changes in others. The extension or modification of the user interface itself is done by simply adding, deleting or modifying the view-controller couples.

The reusability of components make the code clearer and boost development time. The view-controller components of the Android MVC model could be reused through the application and could be easily embedded in other Android applications made with Android Passive MVC.

Good performance is especially important in mobile environments: resource utilization should be limited as mobile devices have little memory. Short response time is essential for modern users. The Android MVC architecture makes a very lightweight Activity component. Controllers, View and Model objects are also small and kept in memory only if used, which minimizes resource utilization. The use of listeners also slightly increases response speed.

B. *Architecture implementation*

We asked an Android developer with three years' experience to test the Android Passive MVC. He chose to redevelop one of his latest applications which had become complex, hard to maintain, extend and test. The application is called 'TaskProjectManager' and it enables tasks to be assigned to different employees and to view the full calendar of tasks on the screen by day, week and month. The application also generates reports by given parameters.

Measurements of both versions of the application are made with javancss, a source measurement suite for Java, and the results are shown in Table I. Android Passive MVC reduces all code parameters.

The Android Passive MVC helps with organizing classes in packages. The original version of the application had many packages created partly using the MVP model, partly the application logic, and partly the Android components named. The limited number of packages of the Android Passive MVC version gives the application a clear structure.

The full code became smaller: both the number of classes and the number of functions were reduced. The Android Passive MVC enables high reusability of components.

The code complexity is evaluated using Cyclomatic Complexity Number (CCN). 'Cyclomatic complexity measures the number of linearly independent paths through a program module.' [5]. Normal method complexity without any risks is 1-10 CCN, with 11-20 CCN the complexity is moderate, with 21-50 CCN the complexity is very high and and with CCNs greater than 50 the program is untestable. Table I shows that the average complexity of the application of the application has decreased slightly. The maximum CCN dropped significantly: an original version has methods with CCNs of 40, 50 and even 100 and 110, while the new version has the only JSON parser with a CCN of 30 and several methods with a CCN of 10 to 15.

The developer's feedback was that the Android Passive MVC model is easy to understand and to follow. The final application was visibly more reactive: the response time became almost nil, while the users of the original version complained about a very long response time for each screen. The Android Passive MVC version is open to extensions and easily modifiable. Application components are not only reusable in the application, but could also be reused in future Android development.

VI. Conclusion and Future Work

We have analysed some well-known architectural design patterns and proposed an Android architecture solution based on an MVC design pattern and the Domain Model organization. The architecture defined can simplify the work of novice and experienced developer alike and enable creation of less complex and well-structured applications. The existing Android application was reimplemented using the Android Passive MVC, resulting in better maintainability, extensibility and performance. The complexity of the new implementation was lower.

We consider a wider evaluation by the Android community. We are currently working on a user-friendly model description and several well-commented implementation examples. We are also drawing up on a questionnaire for the developers who have tested the model. We plan to spread the documentation, examples and a survey over the important websites and blogs to reach a larger audience.

TABLE I
TaskProjectManager statistics

	Original	**Android MVC**	**% Gain**
# Packages	25	17	32
# Classes	393	275	30
# Functions	2186	1683	23
Avg CCN	2,30	1,87	19
Max CCN	110	30	73

This work can be continued by testing the observer-observable design pattern integrated in the Android Passive MVC. The adaptation of the MVP model can be envisaged. The same testing software could be redeveloped to compare the results. Finally, the same test using the Android-binding MVVC framework could be implemented to choose the most effective solution for different types of applications.

References

[1] S. Allen, V. Graupera, and L. Lundrigan, *Pro Smartphone Cross-Platform Development: IPhone, Blackberry, Windows Mobile and Android Development and Distribution*, 1st ed. Berkely: Apress, Sep. 2010.

[2] D. Mark and J. LaMarche, *More IPhone 3 Development*, ser. Tackling Iphone Sdk 3. Berkely: Apress, Jan. 2010.

[3] J. Steele, N. To, S. Conder, and L. Darcey, *The Android Developer's Collection*. Addison-Wesley Professional, Dec. 2011.

[4] B. Foote and J. Yoder, *Big Ball of Mud*. Addison-Wesley, 1997.

[5] T. Ihme and P. Abrahamsson, "The Use of Architectural Patterns in the Agile Software Development of Mobile Applications," *ICAM 2005*, pp. 155–162, Aug. 2005.

[6] G. Krasner and S. Pope, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *Journal of object oriented programming*, vol. 1, pp. 26–49, 1988.

[7] P. Sauter, G. Vögler, G. Specht, and T. Flor, "A Model-View-Controller extension for pervasive multi-client user interfaces," *Personal and Ubiquitous Computing*, vol. 9, no. 2, pp. 100–107, Mar. 2005.

[8] M. Veit and S. Herrmann, "Model-view-controller and object teams: a perfect match of paradigms," in *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. ACM Press, Mar. 2003, pp. 140–149.

[9] S. Burbeck. (1997, Mar.) Applications Programming in Smalltalk-80TM: How to use Model-View-Controller MVC. [Online]. Available: http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html [retrieved: March 2013]

[10] M. Potel, "MVP: Model-View-Presenter the taligent programming model for C++ and Java," Taligent Inc., Tech. Rep., 1996.

[11] J. Cai, R. Kapila, and G. Pal, "HMVC: The layered pattern for developing strong client tiers," *Java World*, pp. 07–2000, 2000.

[12] J. Smith. (2009, Feb.) Wpf apps with the model-view-viewmodel design pattern. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx [retrieved: March 2013]

[13] R. Garofalo, *Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern*. Microsoft Press, Mar. 2011.

[14] S. McConnell, *Tout sur le code : Pour concevoir du logiciel de qualité*, 2nd ed. Dunod, Feb. 2005.

[15] R. Meier, *Professional Android 4 Application Development (Wrox Professional Guides)*, 3rd ed. Birmingham: Wrox Press Ltd., May 2012.

[16] I. Salmre, *Writing Mobile Code: Essential Software Engineering for Building Mobile Applications*. Addison-Wesley Professional, Feb. 2005.

[17] F. Garin, *Android - Concevoir et développer des applications mobiles et tactiles*, 2nd ed. Dunod, Mar. 2011.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, Nov. 1994.

[19] W.-Y. Kim and S.-G. Park, "The 4-tier design pattern for the development of an android application," in *Proceedings of the Third international conference on Future Generation Information Technology*, ser. FGIT'11. Springer-Verlag, Dec. 2011, pp. 196–203.

[20] K. Sokolova. Android passive mvc implementation example. [Online]. Available: https://github.com/KarinaSokolova/android-mvc-example [retrieved: March 2013]

[21] P. D. Sheriff, *Fundamentals of N-Tier Architecture*. PDSA Inc., May 2006.

Project Planning Add-In based on Knowledge Reuse with Product Patterns

Fuensanta Medina-Dominguez, Maria-Isabel Sanchez-Segura, Arturo Mora-Soto, Antonio de Amescua Seco
Computer Science Department
Carlos III University
Leganes, Madrid, Spain
{fmedina, misanche, jmora, amescua}@inf.uc3m.es

***Abstract*—This work presents an approach that incorporates knowledge reuse to the planning process. Project managers can reuse knowledge using product patterns to learn project management techniques. In addition, they can use the add-in support tool proposed in this work to link information to the Gantt chart; therefore, people assigned to each activity in the Gantt chart can reuse existing product patterns that help develop the assigned activity. The authors have corroborated that the proposed solution improves the satisfaction of the people involved in the development of a software project that has been planned using the proposed solution.**

Keywords-Knowledge Management; Project Management; Software Aplication Component; Product Patterns.

I. INTRODUCTION

Project planning has been recognized by the European Commission as essential for a project's success and, as such, is often considered the most important phase in project management [1]. An immense benefit to planning is that, in case a problem arises during the project development, it functions as an alarm mechanism. Also, project planning is a widely explained process in standards like PMBOK [2] and is supported by a wide variety of software tools (analyzed later, in Section II). Nevertheless, there is an aspect of project planning performance that has not been addressed properly, namely, how to incorporate software reuse while project planning is being developed?

Software reuse is the area that studies how to use a thousand times the same piece of software always in a different way. Software reuse is being applied for products developed in different phases of the software development lifecycle by the use of analysis patterns [3], design patterns [4], requirements patterns [5], etc., but in project planning phase, software reuse has not already been incorporated.

It would be very useful to plan the activities to be performed in a software project and, at the same time, plan the potential pieces of software that could be reused on each activity, or even the potential knowledge that could be reused to develop an assigned activity. So, the authors believe that there are at least two scenarios where reuse can be very interesting while planning. One scenario that can occur is when a project manager faces the challenge to develop a project planning; he or she could reuse the knowledge about project planning from experts in the field. Another scenario could be when the project manager is planning the project activities and would like to provide more information regarding the activities the human resources are assigned to. This information can include examples of this activity developed in other projects, lessons learned while developing this activity previously, or references where the person assigned to the activity can learn more about how to develop the activity assigned. The project manager has to always bear in mind the context, the problem, and the forces of the project under development.

Existing project planning tools do not cover these two potential scenarios. This is why the authors propose the use of reuse artifacts, called *Product Patterns* [6] and the use of a software module that has been developed as a Microsoft Project Add-in, which allows the management of product patterns while developing project planning in the previously described scenarios.

The reminder of this paper is structured as follows: Section II describes an analysis of the most remarkable software tools for project management. Section III presents the solution, an add-in support tool based on knowledge reuse with product patterns; this section describes the product pattern, product patterns in project planning and the add-in support tool architecture. Section IV presents the description of the experiment and analysis of the results. Finally, in Section V, the authors present their conclusions and future works.

II. ANALYSIS OF THE MOST REMARKABLE SOFTWARE TOOLS FOR PROJECT MANAGEMENT

When a project manager wants to plan a project, there is no doubt that the most common technique is the Gantt Chart [7], which is typically drawn using a software application. Among the wide variety of project management software tools, according to International Data Corporation (IDC), one of the most notable global providers of market intelligence and analysis [8] is Microsoft Project (MS Project), as can be seen below in Figure 1. MS Project is the most used project manager software tool worldwide [9] and by this fact alone, MS Project could be selected as the best tool to use since it seems that it is the most popular and trustworthy application. However, before choosing a software to implement the authors' knowledge of reuse solution for project planning, an analysis of the most important tools available in the market will be presented in this section, emphasizing whether or not these tools support knowledge reuse to back up project tasks execution.

Nowadays, the most remarkable project management tools available are cloud-based applications or services [10]. In addition, there are software desktop applications that could offer a sort of web synchronization service or feature [9], [11] that include not only project management features, but also, project portfolio features as well as collaboration tools. Since the authors' proposal is focused only to back up a project manager in project planning, the authors only analyzed those tools whose main purpose is project planning. They also analyzed those that are mentioned as relevant by IDC in [9] and by Gartner in its MarketScope for Project and Portfolio Management Software Applications [11] and its Magic Quadrant for Cloud-Based Project and Portfolio Management Services [10]. The tools selected for this analysis were the following:

- *Microsoft Project Professional 2010* [12]. This is the most popular project management software, it is developed and sold by Microsoft [13] and is designed to assist a project manager in developing a plan, assigning resources to tasks, tracking progress, managing the budget, and analyzing workloads.
- *Augeo6* [14]. This is a software solution that organizes and automates all activities related to the life cycle of projects, from the initial evaluation of the project proposal until the completion of the project.
- *Genius Project* [15]. This is a web-based tool that delivers a highly flexible and configurable portfolio and project management software allowing for tailored feature sets for a wide array of project teams and project types.
- Planisware 5 [16]. This is a web-based application that supports the end-to-end governance of company portfolios; it offers a complete project management capability with features such as project and resource scheduling, portfolio reporting, simulation, comprehensive project reporting/cost control, and collaboration tools.
- *Planview Enterprise* [17]. Among its capabilities, this tool delivers visibility into and control of project portfolios, allowing to efficiently prioritize work and make better decisions around request management, planning, and resource capacity.
- *Project.net* [18]. This is a web-based tool aimed to maximize the performance of any organization tracking a single project or a portfolio of projects.
- *Sciforma 5* [19]. This is project and portfolio management software aimed to help project managers to administer all aspects of project, resource, risk, and change management.
- *AtTask* [20]. This is a web-based tool that features task, management, issue tracking, document management, time tracking and portfolio management.

Table I shows the criteria defined to assess the knowledge reuse capabilities. Each criteria is defined by: the criteria, the description and the phase of the knowledge lifecycle supported. To analyze if a tool fulfills a criterion or not, every tool was used to plan a simple software project, looking if the capabilities depicted in the criteria were present or not. The presence of a criterion was rated with the following scale: (0) meant that the criterion was not present; (1) meant that the criterion was partially present; (2) meant that the criterion was completely present; the objective is that the ideal tool could reach a rating of 10, meaning that it has all the criteria completely present. The final results of the analysis are shown in Table II, as can be seen MS Project, Project.net, and AtTask obtained the best ratings. However it is important to highlight that none of the tools analyzed offered any formal knowledge representation mechanism, such as the *Product Pattern* defined by the authors; all of them only offered basic knowledge representation mechanisms such as notes, document attachments, blogs, or wiki. This is an important contribution; nevertheless it is not formal enough to accomplish the goals proposed by the authors, especially to foster an accurate knowledge reuse in project planning.

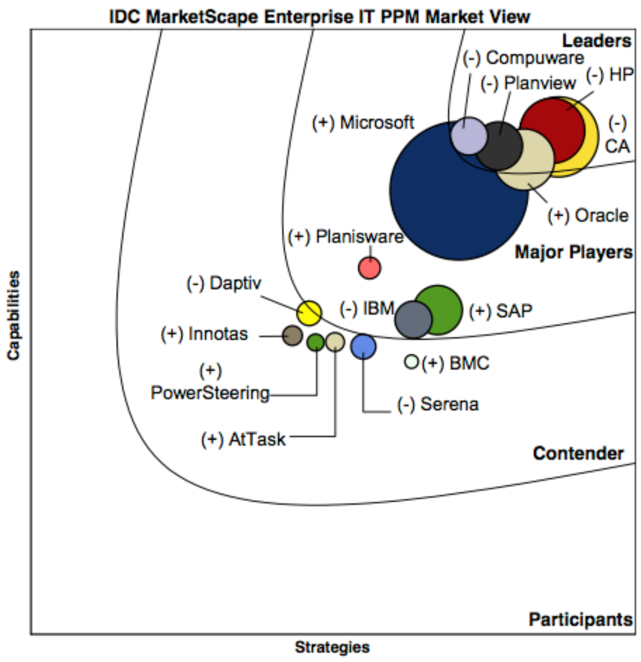


Figure 1 Marker share of project management tools according to IDC

This fact encourages authors to try to improve one of the existing project management software tools, and implement a mechanism to support *Product Patterns* to help project managers improve their project planning activities.

After this analysis, and considering the results offered by Gartner [10], [11] and IDC [9], the authors decided to choose MS Project as the tool to be extended for incorporating a new functionality to link *Product Patterns* and project plan tasks. This decision was made due to Ms Project's wide adoption in the market, a key factor to spreading the use of the solution presented by the authors in this paper, as well as to the large amount of existing documentation to develop new functionalities for this program using the programming languages provided by Microsoft.

TABLE I. CRITERIA DEFINED TO ASSES THE KNOWLEDGE REUSE CAPABILITIES OF PROJECT MANAGEMENT SOFTWARE TOOLS

Criteria	Description	Phase of the Knowledge Lifecycle supported
C1: Basic knowledge assets representation mechanism.	This criterion is intended to identify if the tool offers an integrated mechanism to represent *basic knowledge assets* related to project plan tasks. This kind of asset is any piece of knowledge (an idea, a comment, best practices, or thoughts) that is explicitly represented in natural language, which in turn can be stored in some way that could be shared or used by any person (e.g. document attaching, document sharing, notes, embedded documentation, etc.)	Create
C2: Formal knowledge assets representation mechanism.	This criterion is intended to identify if the tool offers and integrates mechanism to represent *formal knowledge assets* related with project plan tasks. This kind of asset is a piece of knowledge that is represented using a formal or standard notation, such as a metamodel, a pattern language, or a graphical notation (like UML or BPMN).	Create, Codify
C3: Knowledge-tasks linking protocol.	This criterion is intended to identify if the tool offers rules to link the tasks of a project plan with existing knowledge assets that could be helpful to perform them. Knowledge assets could be basic or formal as described above in criteria C1 and C2.	Embed, Diffuse
C4: Knowledge improvement mechanism.	If the tool offers some of the characteristics depicted in criteria C1, C2 and C3, this criterion is intended to identify if the tool offers a mechanism to improve existing knowledge assets that were linked to project plan tasks, for example, by adding new information that could complement the existing one.	Create, Codify, Embed, Diffuse
C5: Tool extension capabilities.	This is not a criterion related to knowledge reuse, but due to the authors' desire to extend the capabilities of the software tool, this criterion is intended to identify if the tool's features can be extended using a programing language.	This criterion does not apply.

TABLE II. RESULTS OF THE PROJECT MANAGEMENT SOFTWARE TOOLS ANALYSIS

Criteria	MS Project Professional 2010	Auge6	Genius Project	Planisware 5	Planview Enterprise	Project.net	Sciforma 5	AtTask
C1	2	2	2	1	2	2	2	2
C2	0	0	0	0	0	0	0	0
C3	1	1	1	1	1	1	1	1
C4	1	1	1	1	1	1	1	2
C5	2	0	0	0	1	2	0	1
Rating	**6**	**4**	**4**	**3**	**5**	**6**	**4**	**6**

III. SOLUTION

This section describes the solution developed to reuse knowledge with product patterns in project planning, therefore describing the product pattern concept, product patterns in project planning and the add-in support tool architecture.

A. *Product Pattern*

Product Patterns are reusable artifacts that store the experts' knowledge and best practices to develop a product [6]. Although product patterns can be used in different fields, in this paper they have been applied in the software engineering field, where the authors are experts.

For the authors, a software product is any product obtained along the activities of the software project life cycle (for example, requirement specification, data base, planning, etc). Product patterns have been formalized in a wiki, which is available at [21]

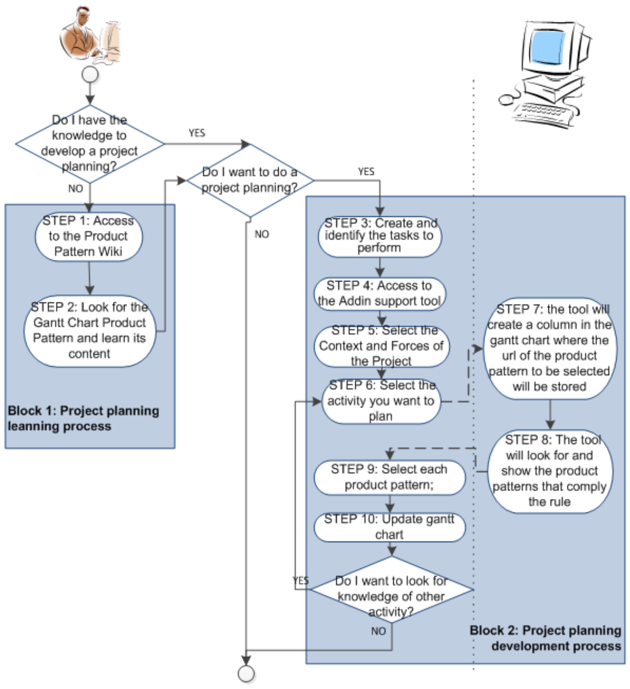
The Gantt Chart Product Pattern [22] is an example of product patterns to perform project planning.

B. *Product Patterns in Project Planning*

When a project manager has to perform a project planning, he or she must think about the next question: Do I have the needed knowledge to develop a project planning based on the software engineering best practices?

Figure 2 illustrates the way authors propose to use product patterns in projects planning. There are two possibilities, that the project manager does not know how to perform a project planning (which is illustrated in Block 1 in Figure 2), or the project manager knows how to develop a project planning and wants to perform it (which is illustrated in Block 2, Figure 2).

Project planning learning process (Block 1 description): If the project manager does not know how to develop a project planning, he or she can learn the software engineering best practices and the experience of other project managers. The project manager has to follow the next two steps: STEP 1: Access to the product pattern wiki, available at [21]. STEP 2: Look for the Gantt Chart Product Pattern and learn its content. In the product pattern wiki, the project manager should look for the "Project Planning Product pattern". With this product pattern, the project manager will learn step by step how to perform a project planning. Lessons learned, information resources, knowledge and skills to perform project planning are also available.

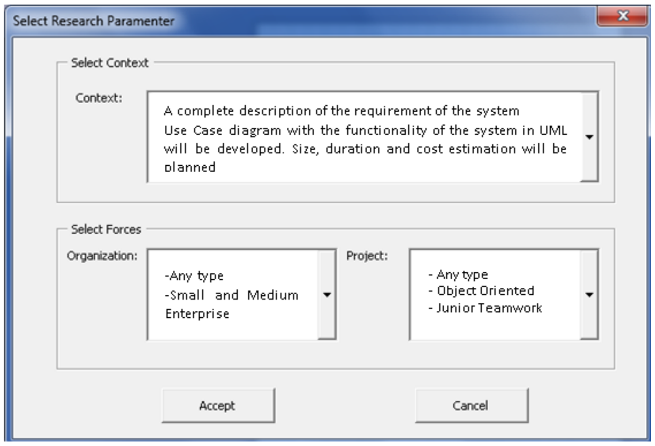


Do I have the knowledge to develop a project planning?
YES
NO
Do I want to do a project planning?
YES
NO
STEP 1: Access to the Product Pattern Wiki
STEP 2: Look for the Gantt Chart Product Pattern and learn its content
Block 1: Project planning learning process
STEP 3: Create and identify the tasks to perform
STEP 4: Access to the Addin support tool
STEP 5: Select the Context and Forces of the Project
STEP 6: Select the activity you want to plan
STEP 7: the tool will create a column in the gantt chart where the url of the product pattern to be selected will be stored
STEP 8: The tool will look for and show the product patterns that comply the rule
STEP 9: Select each product pattern;
STEP 10: Update gantt chart
Do I want to look for knowledge of other activity?
YES
NO
Block 2: Project planning development process

Figure 2 Product Patterns in Project Planning

In this way, product patterns will be useful to the project manager to learn the needed knowledge to perform project planning, using the best practices of software engineering and the experience of software managers who have used and given feedback about the product patterns with the knowledge of using the Gantt Chart product pattern in different projects.

Project planning development process (Block 2 description): If the project manager knows how to perform the project planning and he or she wants to develop a Gantt Chart, the project manager must follow the next steps: STEP 3: Create and identify the tasks to be performed during the software project. STEP 4: Access the add-in support tool developed by the authors. STEP 5: Select the context and the forces of the project; the project manager will have to select the context where the project will be developed and the generic and specific forces that affect the project planning under development, such as the kind of organization, team experience, etc. Figure 3 shows the screenshot where project manager has to select the context and forces. STEP 6: Select the activity you want to plan. As can be seen see in Figure 4, once the context and the force are selected, the project manager will have to select the activity to be planned. STEP 7: The add-in suport tool will create a column in the Gantt Chart where the selected product patterns url will be stored (it can be seen in Figure 5).

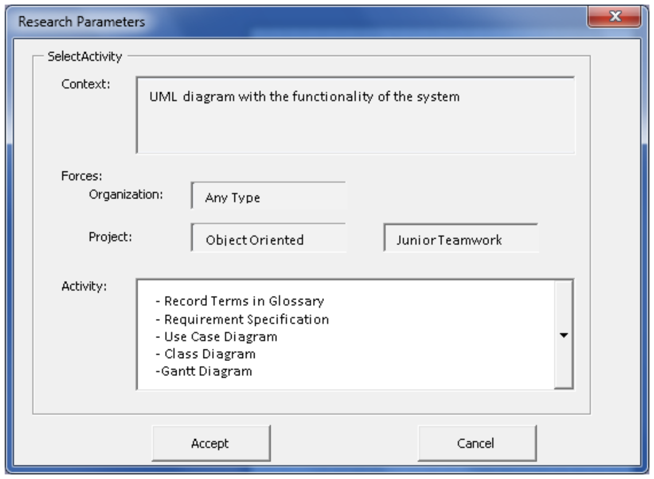


Context: A complete description of the requirement of the system Use Case diagram with the functionality of the system in UML will be developed. Size, duration and cost estimation will be planned

Organization:
-Any type
-Small and Medium Enterprise

Project:
- Any type
- Object Oriented
- Junior Teamwork

Figure 3. Context and forces selection

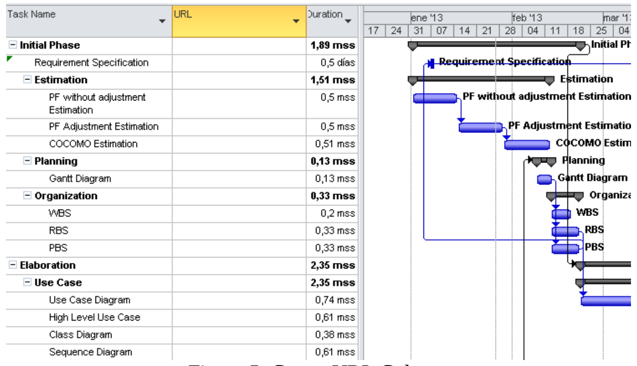


Context: UML diagram with the functionality of the system

Forces:
Organization: Any Type
Project: Object Oriented | Junior Teamwork

Activity:
- Record Terms in Glossary
- Requirement Specification
- Use Case Diagram
- Class Diagram
-Gantt Diagram

Figure 4. Activities selection



Task Name	URL	Duration
Initial Phase		1,89 mss
Requirement Specification		0,5 días
Estimation		1,51 mss
PF without adjustment Estimation		0,5 mss
PF Adjustment Estimation		0,5 mss
COCOMO Estimation		0,51 mss
Planning		0,13 mss
Gantt Diagram		0,13 mss
Organization		0,33 mss
WBS		0,2 mss
RBS		0,33 mss
PBS		0,33 mss
Elaboration		2,35 mss
Use Case		2,35 mss
Use Case Diagram		0,74 mss
High Level Use Case		0,61 mss
Class Diagram		0,38 mss
Sequence Diagram		0,61 mss

Figure 5. Create URL Column

STEP 8: The add-in support tool will look for the product patterns which comply with the context, forces and the activity (problem) that the project manager wants to plan; so the tool will execute the next rule:

If you find yourself in this **context**
  (and) with this **problem**
  (and) entailing these **forces**
then
  map a product pattern in your project
  (and) look for more product patterns

The product patterns that comply the rule will be shown in the tool, this can be seen in Figure 6. STEP 9: The project manager can select each product pattern and the add-in support tool will show the description and the url of the product pattern where the project manager will have access to the best practices and the experiences of other software engineers related to the activity being planned (time, resources, lessons learned, etc).

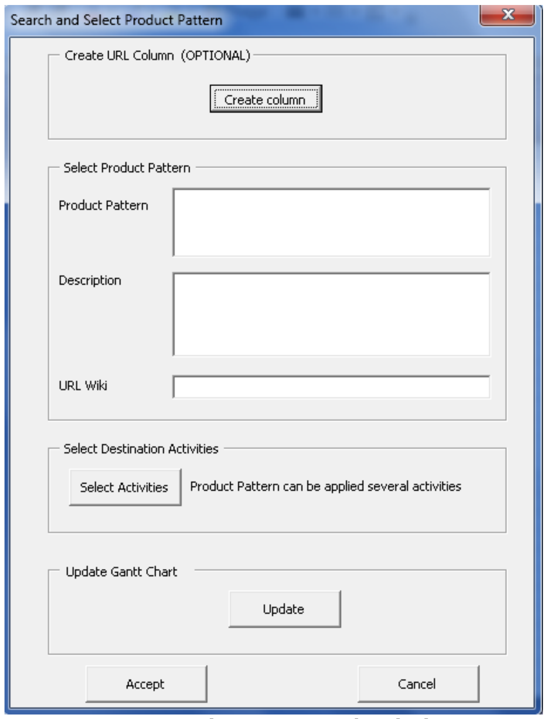
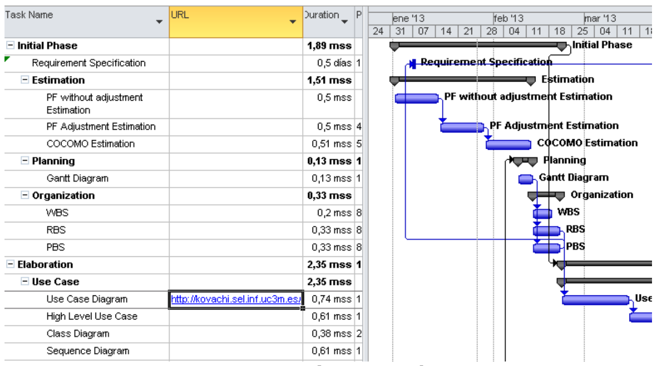

Search and Select Product Pattern
Create URL Column (OPTIONAL)
Create column
Select Product Pattern
Product Pattern
Description
URL Wiki
Select Destination Activities
Select Activities
Product Pattern can be applied several activities
Update Gantt Chart
Update
Accept
Cancel

Figure 6. Product pattern search and select



Task Name	URL	Duration	P
Initial Phase		1,89 mss	
Requirement Specification		0,5 días	1
Estimation		1,51 mss	
PF without adjustment Estimation		0,5 mss	
PF Adjustment Estimation		0,5 mss	4
COCOMO Estimation		0,51 mss	5
Planning		0,13 mss	1
Gantt Diagram		0,13 mss	1
Organization		0,33 mss	
WBS		0,2 mss	8
RBS		0,33 mss	8
PBS		0,33 mss	8
Elaboration		2,35 mss	1
Use Case		2,35 mss	
Use Case Diagram	http://kovachi.sel.inf.uc3m.es/	0,74 mss	1
High Level Use Case		0,61 mss	1
Class Diagram		0,38 mss	2
Sequence Diagram		0,61 mss	1

Figure 7. Update Gantt Chart

At this point, the project manager will have to select the product pattern that best fits with the activity which is being planned. STEP 10: Update Gantt Chart: the add-in suport tool will update the Gantt Chart with the url of the selected product pattern, this can be seen in Figure 7. STEP 11: Save the changes. The add-in suport tool will save the updated project planning.

C. *Add-in Support Tool Architecture*

The architecture of the add-in consists of three modules clearly identifiable:

- The client (or component add-in) is embedded within the Microsoft Project program. The add-in is installed on client computers using a simple self-install, slightly configurable, and outside the building application.
- Web service: it works thanks to an application server; both are located in a server computer. The Web Service WSDL descriptor allows that the services can be public and accessible for the customer.
- Database manager: it is located in a server computer. The database query manager handles the queries of the project manager to obtain the knowledge needed to perform the project planning.

IV. EXPERIMENTATION

This section describes the experiment and the analysis of results.

A. *Description of the experiment*

This solution provides an add-in support tool for project planning using product patterns. The authors validated the time spent developing the project planning from the satisfaction of project managers and teamwork involved in the development of each planned project. The experiment was conducted in two phases:

Phase I: Implementation of project planning without using add-in support tool, and development of the projects planned.

Phase II: Implementation of project planning using add-in support tool developed by the authors and development of the projects planned.

The authors believe that although the development time using the add-in support tool will increase, the level of satisfaction achieved will increase as well because it provides the knowledge of the best practices and the experience and knowledge of experts in software engineering.

To validate this goal, six software projects were developed at Carlos III University in Spain. All the project managers who participated in this validation had between 10 – 15 years' experience, and a Bachelor's degree in computer science. Each of the 6 projects that took part in the validation included:

- Two different project managers that participated in the validation, one project manager to develop the planning without using the add-in support tool, and another one using the add-in.

- Two different teams, one team that used the planning made by the project manager without using the add-in support tool, and the other one that used the planning made by the project manager using the add-in support tool.

A survey was performed to value the level of satisfaction of people involved in the experiment.

B. *Results Analysis*

The data analysis results are shown in Figure 8. The bubble figure shows a comparative: for each project (x-axis) there are two bubbles with the development time (y-axis) and a level of satisfaction (bubble area), within each bubble there are numbers that represent "time; satisfaction".

As can be seen in Figure 8, the development time is greater in phase 2 where the add-in support tool is used. This increase is the result of the project manager learning the knowledge provided by the product patterns to select the ones that best fit with each project activity. Although for each project the time spent in project planning is greater when using the add-in support tool, the bubble area is larger as well because the project planning is done with the knowledge about how each activity affects the project planning. Also, the project manager provides for each activity, using the add-in support tool, a URL to the products patterns wiki, where the person in charge of each activity can access the knowledge on how to perform the assigned activity.

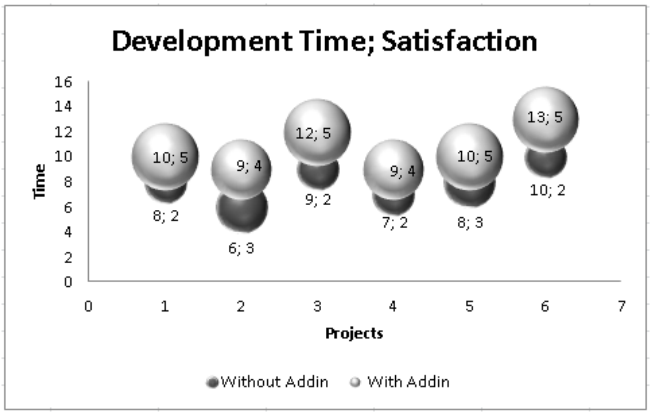


Figure 8. Results Analysis (Phases – Development Time – Level of Satisfaction)

V. Conclusions and Future Works

The most important mission of this work was to focus on the scarce reuse being done in general in project management and specifically in project planning. The authors proposed an easy way to incorporate into project planning all the necessary information (i.e., activities to be done, product to be obtained, people assigned, time schedule, budget, etc.), but also the know-how the software engineers have on developing software products, which can be reused a thousand times and never in the same way to develop the project activities. This has been done by using product patterns, proposed by the authors as artifacts to gather the know-how on how to develop software products and easily accessed by the wiki [21], and an add-in support tool, that can be easily developed by any project development platform (in this case developed to be added to Microsoft Project). The use of the proposed solution has demonstrated that, although the time spent in the project planning increased, the satisfaction of the teamwork while developing their assigned activities also increased. The authors want to demonstrate as future work that the productivity of the teamwork increases as well.

Using this approach is an interesting way to ensure the company which is developing software projects, that the planning has been done by reusing the know-how of the people working in the company and in this way it is easy to assess how the know-how is giving a return of investment to the company.

Acknowledgment

This work has been partially funded by the Spanish Ministry of Science and Technology through the project TIN2011-27244.

References

[1] European Commission. Managing projects. Available at: http://ec.europa.eu/eahc/management/Fact_sheet_2010_01.html. [Accessed: 30-Jan-2013].

[2] PMBOK A Guide to the Project Management Body of Knowledge. Fifth Edition. PMI. 2013.

[3] S. Ketabchi, N.K. Sani, and L. Kecheng, "A Norm-Based Approach towards Requirements Patterns". Computer Software and Applications Conference (COMPSAC), 1 IEEE 35th Annual Topics: Computing & Processing (Hardware/Software), 2011, pp: 590 – 595, Digital Object Identifier: 10.1109/COMPSAC.2011.82

[4] C. Larman. Applying Uml and Patterns: An Introduction to Object-Oriented Analysis and Design, and the Unified Process. 2002. Prentice Hall

[5] L. Hagge, K. Lappe, and T. Schmidt, "REPARE: The Requirements Engineering Patterns Repository". 13th IEEE International Conference on Requirements Engineering (RE'05), 2005, pp. 489-490.

[6] M. Sanchez Segura, F. Medina-Dominguez, A. Amescua and A. Mora-Soto. "Improving the Efficiency of Use of Software Engineering Practices Using Product Patterns". Information Sciences (Impact Factor: 3.291), Vol. 180, Issue 14, July 15, 2010, Pages 2721-2742.

[7] H. Kerzner, Project management: a systems approach to planning, scheduling, and controlling, 10th ed. Hoboken, NJ, USA: Wiley, 2009, p. 1120.

[8] USA IDC Corporate, "International Data Corporation." [Online]. Available: http://www.idc.com/. [Accessed: 24-Jan-2013].

[9] M. Ballou and J. C. Pucciarelli, "IDC MarketScape Excerpt: IT Project and Portfolio Management 2010 Vendor Analysis. Four Views to Enable Effective Evaluation," Framingham, MA, 2010.

[10] D. B. Stang and R. A. Handler, "Magic Quadrant for Cloud-Based Project and Portfolio Management Services", Stamford, CT, USA, 2012.

[11] D. B. Stang and R. A. Handler, "MarketScope for Project and Portfolio Management Software Applications", Stamford, CT, USA, 2012.

[12] Microsoft Corporation, “Microsoft Project Professional 2010. Product Information.,” 2010. [Online]. Available: http://www.microsoft.com/project/en-us/project-professional-2010.aspx. [Accessed: 26-Jan-2013].

[13] Microsoft Corporation, “About Microsoft,” 2013. [Online]. Available: http://www.microsoft.com/about/en/us/default.aspx. [Accessed: 25-Jan-2013].

[14] Augeo Software SAS, “Augeo6 Product Description,” 2012. [Online]. Available: http://www.augeo.com/page/augeo6. [Accessed: 24-Jan-2013].

[15] Genius Inside, “Genius Project. Product Description.,” 2012. [Online]. Available: http://www.geniusinside.com/software/software. [Accessed: 24-Jan-2013].

[16] Planisware, “Planisware 5 information.,” 2011. [Online]. Available: http://www.planisware.com/main.php?docid=12420. [Accessed: 24-Jan-2013].

[17] Planview Inc., “Planview Enterprise description.,” 2013. [Online]. Available: http://www.planview.com/products/enterprise/. [Accessed: 24-Jan-2013].

[18] Project.net, “Project.net overview.,” 2013. [Online]. Available: http://www.project.net/overview. [Accessed: 24-Jan-2013].

[19] Sciforma Corporation, “Sciforma 5.0 overview.,” 2012. [Online]. Available: http://www.sciforma.com/page?id=927. [Accessed: 24-Jan-2013].

[20] AtTask Inc., “attask product overview.,” 2012. [Online]. Available: http://www.attask.com/product. [Accessed: 24-Jan-2013].

[21] Product Pattern Wiki. Available at: http://kovachi.sel.inf.uc3m.es

[22] Gantt chart Product Pattern. Available at: http://kovachi.sel.inf.uc3m.es/800-spanish/801_libreria_de_patrones_de_producto/Diagrama_de_Gantt

Comparing Two Architectural Patterns for Dynamically Adapting Functionality in Online Software Products

J. Kabbedijk, T. Salfischberger, S. Jansen
Department of Information and Computing Sciences
Utrecht University, The Netherlands
J.Kabbedijk@uu.nl, Tomas@salfischberger.nl, Slinger.Jansen@uu.nl

***Abstract*—Business software is increasingly moving towards the cloud. Because of this, variability of software in order to fit requirements of specific customers becomes more complex. This can no longer be done by directly modifying the application for each client, because of the fact that a single application serves multiple customers in the Software-as-a-Service paradigm. A new set of software patterns and approaches are required to design software that supports runtime variability. This paper presents two patterns that solve the problem of dynamically adapting functionality of an online software product; the Component Interceptor Pattern and the Event Distribution Pattern. The patterns originate from case studies of current software systems and are reviewed by domain experts. An evaluation of the patterns is performed in terms of security, performance, scalability, maintainability and implementation effort, leading to the conclusion that the Component Interceptor Pattern is best suited for small projects, making the Event Distribution Pattern best for large projects.**

***Keywords*—architectural patterns. quality attributes. software architecture. variability.**

I. Introduction

Software as a Service (SaaS) is a rapidly growing deployment model with a clear set of advantages to software vendors and their customers. SaaS allows vendors to deploy changes to applications more rapidly, which increases product innovations while reducing support-costs as only a single version is to be supported concurrently [1]. In the SaaS deployment model a single application serves a large number of customers. These customers are called tenants, which can be a single user or an organisation with hundreds of users. Because all tenants use the same application, the cost of development and setup of the application can be amortized over all contracts.

The multi-tenant deployment model requires the application to be aware of different tenants and their users, for example in separating the data visible to different groups of users. We define multi-tenancy as: “the property of a system where multiple varying customers and their end-users share the system’s services, applications, databases, or hardware resources, with the aim of lowering costs”. Database designs for multi-tenant aware software require specialized architecture principles to accommodate multiple tenants [2]. One of the challenges in multi-tenant application architectures is the implementation of tenant-specific requirements [3]. Variability of software to fit requirements of specific customers can no longer be done by directly modifying the application for each client, because a single application serves multiple customers. A new set of software patterns and approaches are required to design software that supports runtime variability. The patterns vary in impact on the technical properties of the software like performance and maintainability, impact on the cost-drivers of the SaaS business model, and the requirements they can fulfil.

The concepts of variability and quality attributes are explained in Section II, after which the expert evaluation used is explained in Section III. The architectural problem related to variability, faced by software architects, is explained in Section IV. The Component Interceptor Pattern and the Event Distribution Pattern, two patterns both solving the problem of dynamically adapting functionality of online business software, are presented in Section V. The patters are compared in terms of security, performance, scalability, maintainability and implementation effort, of which the results in be found in a summarizing table in Section VII.

Please note; in the text, we set pattern names in small caps according to the convention by Alexander et al. [4].

II. Related Work

Variability - The field of software variability has been the subject of research from both the modeling perspective as well as the technical perspective. Software variability modeling is common in software product lines as described by Jaring and Bosch [5]. The application of variability modeling as used in product line variability [6] to software as a service environments has been described by Mietzner, Metzger, Leymann, and Pohl [7]. Variability modeling as dicussed in the aforementioned works contributes to the understanding of where the application architecture needs to be able to accomodate change or extension. Patterns play an important role in modeling and solving variability in software products [8].

Svahnberg, van Gurp, and Bosch [9] propose feature diagrams as a modeling technique to describe the different variants of feature in a software product. Svahnberg et al [9] use their feature diagrams as the basis for a method to identify variability in a product, constrain this variability, pick a method of implementation for the variability and further manage this variability point in the application lifecycle. The main difference from the objectives of our research is that Svahnberg et al. [9] describe implementation techniques for variability per installation instance of the software, whereas we focus on *runtime* variability in a multi-tenant context.

Quality Attributes - Benlian and Hess [10] identify *security* as one of the most important risk-factors perceived, followed by performance risks. To assess security risks, SaaS vendors need to include security as a quality attribute in their design of the architecture. This leads to security as the first desired quality attribute for business SaaS. *Performance* as an important factor to SaaS users is closely related to the most important factor as found by Benlian and Hess [10]; cost. When performance is insufficient, clients are lost, when the system uses too many resources to gain an acceptable level of performance, cost is increased. A SaaS vendor must thus assess the possible performance impact of changes to the software. To control cost in business SaaS, the SaaS vendor needs to utilize its opportunities for scalability to decrease the cost of hardware or hosting fees (e.g. using scalable software to make optimal use of cloud-hosting).

Another cost driver in SaaS is the *cost of development* and *maintenance* of the software product. Maintenance cost is generally decreased by having to maintain only a single version instead of multiple previous releases. On the other hand this maintainability cost-saving must not be lost while implementing runtime variability. Thus scalability and maintainability are also desired quality attributes for business SaaS. Another way the implementation of runtime variability will influence product cost is through implementation-cost. Development is a cost-driver for SaaS, thus if one or more specialized developers are required to implement a certain pattern this will influence the final product cost.

The identified quality attributes are the following: **Security** - The ability to isolate tenants from each other and the possible impact of security breaches in custom components on other parts of the system.
Performance - The utilization of computing, storage and network resources by the application at a certain level of usage by clients.
Scalability - The relative increase in capacity achieved by the addition of computing, storage and network resources to the system as well as the flexibility with which these resources could be added to the system.
Maintainability - The ease with which the system can be extended and potential problems can be solved.
Implementation Effort - The effort required to implement and deploy a specific system.

III. Research Approach

In order to gather the patterns in this research, a design science approach [11] was used in which the initial solutions are observed in case studies in which one of the authors took part as a consultant. The solutions are implemented in current commercial software products. Solutions that are observed in multiple at least three products are presented as patterns and are evaluated by two domain experts as feedback mechanism. The evaluation of the cases by experts enhances the validity of the cases, as described by Runeson and Höst [12].

During each evaluation session, a pattern is discussed with an expert, in a semi-structured way. Standard questions related to the quality attributes are asked, after which issues are freely discussed per quality attribute. The first expert selected is a senior software architect in an international software consulting firm specialized in large scale development of Enterprise Java applications. His role is to investigate technologies and methodologies to help design better architectures resulting in faster development and more extensible software. A recent project includes a multi-tenant administrative application storing security sensitive data for multiple organizations.

The second expert is a technology director and lead architect for an application used in distributed statistics processing of marketing data, previously working in software performance consulting for web-scale systems. His experience lies in the field of high-performance distributed computing. The application his company works on focuses of low-latency coordinated processing of large volumes of data to calculate metrics used for marketing. Performance and scalability are important areas of expertise for their product.

IV. Architectural Problem Definition

Software product vendors not only need to offer a *data model* that fits an organisation's requirements, *software functionality* also has to meet an organisation's processes [13]. When tailor-made software is developed, it is possible to set the requirements to exactly match the processes of a specific organisation. For standard online software products this is not possible and differences between requirements of organisation have to be addressed at runtime.

A requirement for the ERP system of a manufacturing company could be to send a notification to the department responsible for transportation if tomorrow's batch will be larger than a certain size. If this requirement is not met by the software product selected, the company could either decide to select another software product or develop a tailor-made application that does meet their requirements.

To allow for the addition of extra functionality in the application a solution is needed that allow to configure this functionality. This functional situation is modeled in Figure 1, the envisioned functional situation. The *StandardComponent* is a normal component of the software with default functionality, this component has a set of *ExtensionPoints*. An *ExtensionPoint* is a location within the normal workflow where there is a possibility to add or change functionality. This functionality is specified in an *ExtensionComponent*, which contains the actual functionality that is to be executed at the specified *ExtensionPoint*.

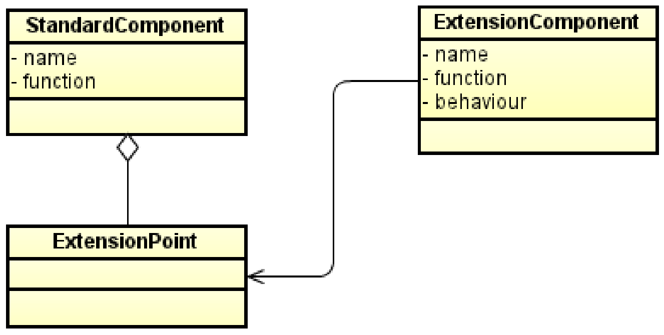


Fig. 1: Functional Model for adapting functionality

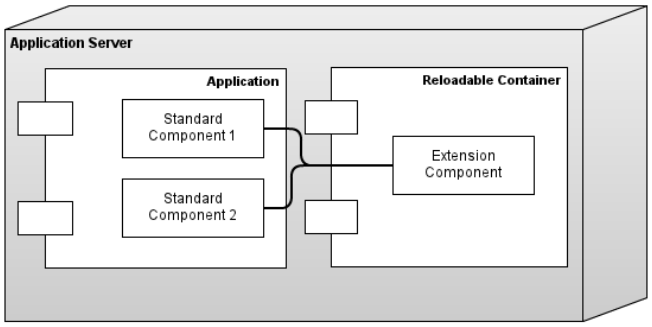


Fig. 2: Component Interceptor Pattern: System Model

V. Dynamical Functionality Adaptation Patterns

This section presents two different patterns, both offering a solution to dynamically adding functionality to a software product.

Component Interceptor Pattern - The Component Interceptor Pattern as depicted in Figure 2 consists of only a single application server. Interceptors are tightly integrated with the application, because they run in-line with normal application code. Before the *StandardComponent* is called the interceptors are allowed to inspect and possibly modify the set of arguments and data passed to the standard component. To do this the interceptor has to be able to access all arguments, modify them or pass them along in the original form. Running interceptors outside of the application requires marshalling of the arguments and data to a format suitable for transport, then unmarshalling by the interceptor component and again marshalling the possibly modified arguments to be passed on to the standard component that was being intercepted. This is impractical and involves a performance penalty [14].

Running the extension components inside the application-server while supporting runtime variability requires support for adding and changing interceptors at runtime. The system model depicts this requirement in the form of a reloadable container. In some implementations this could be as simple as changing a source file, because the programming platform used will interpret source code on the fly. Other platforms require special provisions for reloading code, such as OSGi for the Java platform or Managed Extensibility Framework for the .NET platform.

Figure 3 depicts the interaction with interceptors involved. Interaction with standard components that can be extended goes through the interceptor registry. This registry is needed to keep track of all interceptors that are interested in each interaction. Without the registry the calling code would have to be aware of all possible interceptors. As depicted, multiple interceptors can be active per component. It is up to the interceptor registry to determine the order in which interceptors will be called. An example strategy would be to call the first registered interceptor first or to register an explicit order when registering the interceptors.

Each interceptor has the ability to change the data that is passed to the standard component, modify the result returned by the standard component, execute actions before or after passing on the call or even skip the invocation of the next step all together and immediately return. Immediately returning would for example be used when the interceptor implements certain extra validation steps and refuses the request based on the outcome of the validation. As a result of these possibilities the interceptors must be invoked in-line with the standard component, the application cannot continue until all interceptors have finished executing.

In the event distribution pattern the application generates events at extension points, which are distributed by a broker. At each extension point the standard component is programmed to send an event indicating the point and appropriate contextual data (e.g. which record is being edited) to a broker. For example in a CRM system the standard component for editing client-records sends a *ClientUpdated* event with the ID of the client that was edited. Extension components listen for these events and take appropriate actions based on the events received. In the example of a *ClientUpdated* event an extension component could be developed that sends a notification to an external system to update the client details there.

Event Distribution Pattern - The system model in Figure 4 depicts the distributed nature of the Event Distribution Pattern. Standard components run in the application server, sending events to a central broker, which can be run outside of the application. Extension components are isolated and can be on a separate physical server or run as separate processes on the same server depending on capacity and scale of the application. Components are loosely coupled, sharing only the predefined set of events.

The standard components are unaware of which extension components listen for their events, execution of extension components is decoupled from the standard components. Executing the extension components separately allows for independent scalability of these components. Depending on system load and the volume of events each component listens for, it is possible to allocate the appropriate amount of resources to each component. Because there is no interaction between listeners, it is possible to execute all listeners in parallel if appropriate for the execution environment.

Standard components publish events to the broker as depicted in the sequence diagram in Figure 5. The activation of the standard component not necessarily overlaps with its listeners. After publishing the event, a standard component is free to continue execution. Depending on the fault tolerance and nature of the events it is up to the standard component

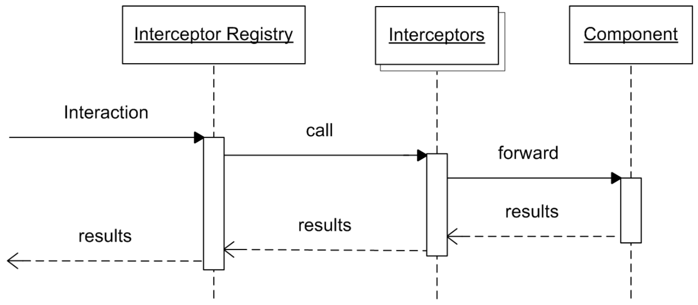


Fig. 3: Component Interceptor Pattern: Sequence Diagram

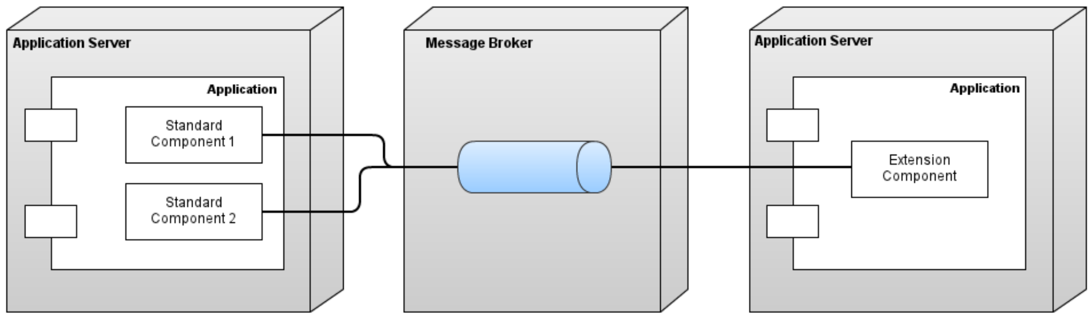


Fig. 4: Event Distribution Pattern: System Model

to make a trade-off between guaranteed delivery at a higher latency by waiting on the broker system to acknowledge reception of the event or continue without waiting for such an acknowledgement. If, for example, an event is only meant to prime a cache for extra performance the loss of such a message would not impact critical functionality of the system while waiting for the message might mitigate any performance gains. If on the other hand an event is used for updating an external system for which no other synchronization method is available the system needs guaranteed delivery to function correctly. At design time this decision can be made on an event by event basis depending on the capabilities of the messaging system used.

Because of the one-way nature of events and decoupled execution of extension components it is not possible for an *ExtensionComponent* to stop standard functionality from happening. In the observed system this was solved by allowing *ExtensionComponents* to execute a compensating action in their listener. The compensating action is sent from the listener component back to the system independently of the original action that caused the event. An example of such a compensating action is an extension component that monitors changes to certain records and reverts the change in case special conditions are met. This approach has the added benefit that any changes made by extension components are clearly visible in audit logs, which simplifies tracing possibly unexpected system behaviour back to an *ExtensionComponent*.

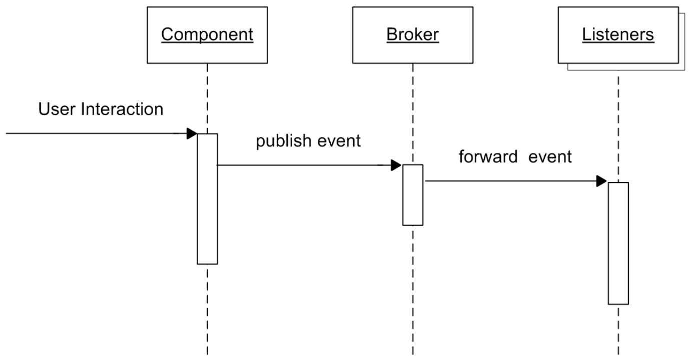


Fig. 5: Event Distribution Pattern: Sequence Diagram

VI. Pattern Comparison

This section presents an analysis of both patterns on the five presented quality attributes.

A. Security

When adapting functionality of an application, there is always the possibility of introducing new security vulnerabilities. This is an inherent risk of extending an application. The variability patterns do however influence how much larger the attack surface becomes and how well a breach in one of the components is isolated from other components. In the COMPONENT INTERCEPTOR PATTERN the code handling the new functionality becomes part of the application and will have the ability to execute arbitrary code within the context of the main application as depicted in Figure 2. It will also have full access to any parameters passed to intercepted functions as well as any returned values. A security breach in the extension components (interceptors) is not isolated to only those components unless extra security measures are implemented to separate the components from the main application. This isolation would however have an impact on performance because of the nature of the integration.

The EVENT DISTRIBUTION PATTERN isolates the extension components from the application by executing them in a separate context based on incoming events as depicted in Figure 3. This execution in a separate context allows for more isolation between extension components and the main application components. The components also have far more limited access to standard functionality, because any change the component wants to make has to go through explicitly exported APIs or messages. Combined with event-sourcing, any change to data as a result of custom functionality is fully traceable including the original values [15].

B. Performance

The COMPONENT INTERCEPTOR PATTERN executes interceptors within the context of the application. This results in little overhead when executing the extension components, because data does not need to be marshalled, unmarshalled and transferred between applications. For security reasons it could however be necessary to separate the interceptors from the main application as described in the previous section. This removes one of the performance advantages of the component interceptor pattern because data must be transferred between the different contexts.

Applications implementing the EVENT DISTRIBUTION PATTERN require the setup of a message broker that handles all events coming from the application and going into the extension components. This requires extra processing and network resources and in the case of durable message delivery mechanisms also storage resources reading and writing the messages. To transfer the events from the application via a message broker to the extension components the events must be marshalled into a format suitable for transferring over a network and unmarshalled upon reception by the extension component, these steps add non-trivial cost to the operations.

C. Scalability

Applications using the COMPONENT INTERCEPTOR PATTERN will execute interceptors within the context of the application. This has performance advantages described in the previous section, however the interceptors cannot be scaled independently of the application. When a high number of interceptors exists requiring significant resources the application as a whole needs more application servers to execute. The interceptors must be available to all application servers in that case.

The EVENT DISTRIBUTION PATTERN on the other hand decouples the execution of the event handlers from the application by running them on a logically separate application server. Because events are handled outside the execution flow of the standard components they can also be distributed to multiple systems. Adding extra application servers subscribing to the same events in the message broker the processing capacity of events could increase linearly. For the EVENT DISTRIBUTION PATTERN this requires a message broker system that is able to handle the increasing numbers of messages. Those systems are available off the shelf from open source projects like Fuse Message Broker, JBoss Messaging, RabbitMQ and commercial offerings like Microsoft BizTalk, Oracle Message Broker, Cloverleaf and others.

D. Maintainability

When adapting the functionality of an application, maintainability is also affected by the necessity to make sure future extensions and modifications are compatible with any custom functionality implemented for tenants. This is a trade-off between the flexibility and depth with which *ExtensionComponents* can affect the application and the impact that changes to the application will have on the *ExtensionComponents*. As an example of the aforementioned trade-off a simple system with only a single *ExtensionPoint* will have a much lower impact on maintainability than a complex system with a very high number of *ExtensionPoints*. This however affects both patterns equally.

The way the patterns decouple *ExtensionComponents* from *StandardComponents* is however a differentiating factor. In the COMPONENT INTERCEPTOR PATTERN the *ExtensionComponent* is more tightly integrated with the *StandardComponent* because calls to a *StandardComponent* at an *ExtensionPoint* go through the interceptor providing all parameters and return values of the call. When changing calls by adding or removing parameters this will directly affect the input of each *ExtensionComponent* registered from that *ExtensionPoint*. When applying the event distribution pattern the integration is more decoupled because calls to StandardComponents are not directly affected by the *ExtensionComponents*. Instead the *ExtensionComponent* receives a standardized event-message and uses a provided API to send any changes or other actions back to the application. This allows for changes to the *StandardComponent* without changing the event-messages going to the *ExtensionComponent*. At the same time the API used by *ExtensionComponents* to influence the application can be kept stable for small changes or versioned to support future compatibility using methods like the one described by Weinreich, Ziebermayr, and Draheim [16].

E. Implementation effort

When implementing a pattern for adding functionality to an application we distinguish two factors determining the implementation effort. The first factor is the direct effort required to implement the pattern in the system, e.g. adding *ExtensionPoints* to the *StandardComponents* of the application. The second factor is the effort necessary to implement *ExtensionComponents*. Later changes to the components might also require development effort, this is however excluded from implementation effort because it is covered under maintainability. Both patterns require the definition and implementation of *ExtensionPoints*, the way these points are implemented differs per pattern. When implementing the COMPONENT INTERCEPTOR PATTERN it is necessary to setup an Interceptor Registry and modify calls to *StandardComponents* to go through the Interceptor Registry.

In the EVENT DISTRIBUTION PATTERN, a message broker system must be setup to handle the event-messages flowing from *StandardComponents* to *ExtensionComponents*. The application still has to be modified at the ExtensionPoints to send the event-messages belonging to that *ExtensionPoint*. A larger difference between the two patterns emerges in the way they influence the system. Using component interceptor pattern each interceptor has full access to the application because it executes within the same context. Communication with *StandardComponents* from within *ExtensionComponents* could use normal function-calls just like any other part of the system. This differs from the event distribution pattern where the *ExtensionComponents* execute in a separate environment outside the context of the *StandardComponents*. Any interaction between *ExtensionComponents* and *StandardComponents* needs to go through an external interface. Depending on the type of system and the requirements for interaction this requires the development of some sort of (webservice-)API for the *ExtensionComponents* to use.

The second factor of implementation effort, the effort required to implement ExtensionComponents, affects both patterns. In the COMPONENT INTERCEPTOR PATTERN the implementation requires the development of an interceptor, which executes the correct behaviour when certain conditions are met. The EVENT DISTRIBUTION PATTERN requires the development of ExtensionComponents, which listen for the right messages and execute the correct functionality when certain conditions are met.

VII. Conclusion

Within this paper the COMPONENT INTERCEPTOR PATTERN and the EVENT DISTRIBUTION PATTERN are compared in terms of security, performance, scalability, maintainability and implementation effort. Both patterns offer a solution for dynamically adapting functionality of an online software product, both do so in different ways.

The COMPONENT INTERCEPTOR PATTERN performs less in terms of *scalability*, because the interceptors can not scale independently of the application. When scaling up in terms of number of servers, the interceptors need to be available to all servers. Related to this issue, the *maintainability* of the COMPONENT INTERCEPTOR PATTERN is also less than that of the EVENT DISTRIBUTION PATTERN. This is caused by the fact the interceptors can not be decoupled from the rest of the system, creating a software product which will be difficult to maintain. The EVENT DISTRIBUTION PATTERN offers more isolation in terms of *security* than the other pattern, but requires more processing and network resources in terms of *performance*. Related to *implementation effort*, the COMPONENT INTERCEPTOR PATTERN is easier to implement, because no message broker or related services are required. Please see Table I for an overview of the evaluation of both patterns. Plus and minus signs are used to indicate whether a characteristic is positive or negative. Keep in mind all scores are relative scores compared to the other pattern.

In general, the COMPONENT INTERCEPTOR PATTERN serves best for adapting functionality of small projects, where the EVENT DISTRIBUTION PATTERN is better for large projects, considering the quality attributes described in this paper. For future work we are currently setting up larger evaluation sessions in which different patterns will be evaluated using experts. The evaluation of patterns is particularly difficult, because you shoud evaluate an abstract solution instead of a specific implementation. We are working on a structured method for comparing sets of patterns and making use of the implicit knowledge of experts. By doing this, we aim at evaluation the *solution*, instead of just an *implementation*.

Acknowledgment

The authors would like to thank Allard Buijze and Koen Bos for helping in reviewing the results of the research.

References

[1] A. Dubey and D. Wagle, “Delivering software as a service,” *The McKinsey Quarterly*, vol. 6, 2007.

[2] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger, “Multi-tenant databases for software as a service: schema-mapping techniques,” in *Proceedings of the ACM SIGMOD international conference on Management of Data*. ACM, 2008, pp. 1195–1206.

[3] S. Jansen, G. Houben, and S. Brinkkemper, “Customization realization in multi-tenant web applications: case studies from the library sector,” *Lecture Notes in Computer Science*, pp. 445–459, 2010.

[4] C. Alexander, S. Ishikawa, and M. Silverstein, *A pattern language: towns, buildings, construction*. Oxford University Press, USA, 1977, vol. 2.

[5] M. Jaring and J. Bosch, “Representing variability in software product lines: A case study,” *Software Product Lines*, pp. 219–245, 2002.

[6] J. Bayer, S. Gérard, O. Haugen, J. Mansell, B. Moller-Pedersen, J. Oldevik, P. Tessier, J. Thibault, and T. Widen, “Consolidated product line variability modeling,” 2006.

[7] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl, “Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications,” in *Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems*. IEEE Computer Society, 2009, pp. 18–25.

[8] J. Kabbedijk and S. Jansen, “The role of variability patterns in multi-tenant business software,” in *Proceedings of the Joint 10th Working IEEE/IFIP Conference on Software Architecture and 6th European Conference on Software Architecture Companion*. ACM, 2012, pp. 143–146.

[9] M. Svahnberg, J. Van Gurp, and J. Bosch, “A taxonomy of variability realization techniques,” *Software: Practice and Experience*, vol. 35, no. 8, pp. 705–754, 2005.

[10] A. Benlian and T. Hess, “Opportunities and risks of software-as-a-service: Findings from a survey of it executives,” *Decision Support Systems*, vol. 52, no. 1, pp. 232–246, 2011.

[11] A. Hevner and S. Chatterjee, “Design science research in information systems,” *Design Research in Information Systems*, pp. 9–22, 2010.

[12] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[13] W. Van der Aalst, A. ter Hofstede, and M. Weske, “Business process management: A survey,” *Business Process Management*, pp. 1–12, 2003.

[14] B. Carpenter, G. Fox, S. Ko, and S. Lim, “Object serialization for marshalling data in a java interface to mpi,” in *Proceedings of the ACM Conference on Java Grande*. ACM, 1999, pp. 66–71.

[15] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Professional, 2003.

[16] R. Weinreich, T. Ziebermayr, and D. Draheim, “A versioning model for enterprise services,” in *21st International Conference on Advanced Information Networking and Applications Workshops*, vol. 2. IEEE, 2007, pp. 570–575.

TABLE I: Overview of both Dynamical Functionality Adaptation Patterns

	Component Interceptor Pattern	Event Distribution Pattern
Security	- Extension components execute within application scope.	+ Isolation of extension components and full traceability of actions by extension components.
Performance	+ Direct execution of extension components.	- Network overhead for calling extension components. - The broker system requires extra resources.
Scalability	- No independent scaling of extension components. - Does not scale to high number of extension components.	+ Independent scaling of extension components. + Extension components cannot delay standard components. - Requires scalable message-broker system.
Maintainability	- Tight coupling of extension components.	+ Loose coupling of extension components.
Implementation Effort	+ Direct communication with standard components. + Access to all data by design.	- Requires the setup of a message broker system. - Requires a separate mechanism to communicate with the application.

Using Patterns to Move the Application Data Layer to the Cloud

Steve Strauch, Vasilios Andrikopoulos, Uwe Breitenbücher, Santiago Gómez Sáez, Oliver Kopp, Frank Leymann
Institute of Architecture of Application Systems (IAAS),
University of Stuttgart, Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de

***Abstract*—Cloud services allow for hosting applications partially or completely in the Cloud by migrating their components and data. Especially with respect to data migration, a series of functional and non-functional challenges like data confidentiality arise when considering private and public Cloud data stores. In this paper we identify some of these challenges and propose a set of reusable solutions for them, organized together as a set of Cloud Data Patterns. Furthermore, we show how these patterns may impact the application architecture and demonstrate how they can be used in practice by means of a use case.**

***Keywords*—Data layer; Cloud applications; Data migration; Cloud Data Patterns; Cloud data stores.**

I. Introduction

Cloud computing has become increasingly popular with the industry due to the clear advantage of reducing capital expenditure and transforming it into operational costs [1]. To take advantage of Cloud computing, an existing application may be moved to the Cloud or designed from the beginning to use Cloud technologies. Applications are typically built using a three layer architecture model consisting of a presentation layer, a business logic layer, and a data layer [2]. The presentation layer describes the application-users interactions, the business layer realizes the business logic and the data layer is responsible for application data storage. The data layer is in turn subdivided into the Data Access Layer (DAL) and the Database Layer (DBL). The DAL encapsulates the data access functionality, while the DBL is responsible for data persistence and data manipulation. Figure 1 visualizes the positioning of the various layers.

Each application layer can be hosted using different Cloud deployment models. Possible Cloud deployment models, also shown in Figure 1, are: Private, Public, Community, and Hybrid Cloud [3]. Figure 1 shows the various possibilities for distributing an application using the different Cloud types. The "traditional" application not using any Cloud technology is shown on the left of the figure. In this context, "on-premise" denotes that the Cloud infrastructure is hosted inside the company and "off-premise" denotes that it is hosted outside the company.

In this work, we focus on the lower two layers of Figure 1, the DAL and DBL layers of the application. Application data is typically moved to the Cloud because of, e. g., Cloud bursting, data analysis or backup and archiving. Using Cloud technology leads to challenges such as incompatibilities with the database layer previously used or the accidental disclosing of critical data by, e. g., moving them to a Public Cloud. Incompatibilities in the database layer may refer to inconsistencies between the functionality of an existing traditional database layer, and the functionality and characteristics of an equivalent Cloud Data Hosting Solution [4]. For instance, the Google App Engine Datastore [5] is incompatible with Oracle Corporation MySQL, version 5.1 [6], because the Google Query Language [7] supports only a subset of the functionality provided by SQL, e. g., joins are not supported. An application relying on such functionalities cannot therefore have its data store moved to the Cloud without deep changes to its implementation. It has to be noted here that, for the purposes of this work, we assume that the decision to migrate the data layer to the Cloud has already been made based on criteria such as cost, effort etc. [8], [9].

The contribution of this paper is the identification of such challenges and the description of a set of *Cloud Data Patterns* as the best practices to deal with them. As defined in [10], a Cloud Data Pattern describes a *reusable and implementation technology-independent solution for a challenge related to the data layer of an application in the Cloud for a specific context*. For this purpose, in the following we present an initial catalog of Cloud Data Patterns dealing with functional, non-functional and privacy-related aspects of having the application data layer realized in the Cloud. The Cloud Data Patterns are geared towards the Platform as a Service (PaaS) delivery model [3]. The presented list of the patterns is a result of our collaboration with industry partners and research projects. We do not claim that the list of patterns is complete and we plan to expand it in the future.

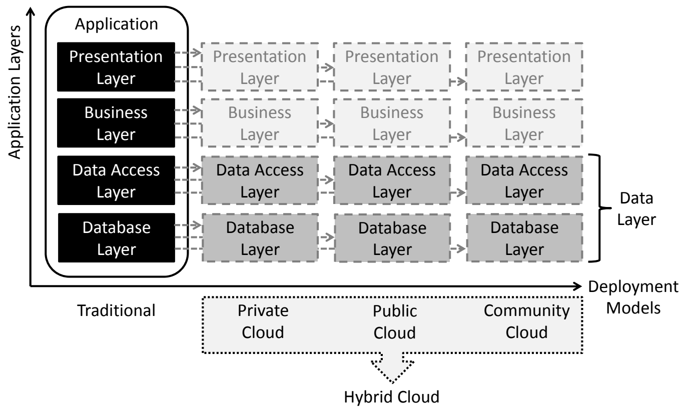


Figure 1: Overview of Cloud Deployment Models and Application Layers

The presentation of the patterns uses the format defined by Hohpe and Woolf [11], consisting of the description of a *context* where the pattern is applicable, the *challenge* posed, external or internal *forces* that impose constraints that make the problem difficult to solve, a proposed *solution* for the challenge, detailed technical issues (as *sidebars*), the *results* of applying the proposed solution in the defined context, an *example* of use, and other patterns to be considered (*next*). A representative *icon* and a graphical *sketch* of the pattern are also provided. In addition, we show how these patterns can be used in practice by means of a use case.

The remainder of this paper starts with providing a motivating scenario (Section II) highlighting some of the challenges that need to be addressed in the following sections. A set of functional Cloud Data Patterns are presented in Section III as best practices for addressing these challenges. Sections IV and V summarize and update some of the patterns we defined for the same purposes in [4] and [10], respectively, but with respect to Quality of Service and data confidentiality. Section VI discusses the impact of applying these patterns to the application layers and evaluates them in practice using the motivating scenario of Section II. A presentation of related work is contained in Section VII; conclusions and future work in Section VIII.

II. Motivating Scenario

For purposes of an illustrative example let us consider the case of a health insurance company in Germany. As a result of the increase of the numbers of its clients, the company stores their data in two data centers in different parts of Germany. The data centers, covering geographical regions A and B, respectively, form a private Cloud data hosting solution that offers a uniform access point and view of the data to the various applications used by the employees of the company. The company is required to provide access to an external auditor to the financial transactions processed by the company. The auditor essentially executes a series of predefined complex queries on the financial transactions data at irregular intervals and reports back to the company and the responsible authorities their findings. The health insurance company however is also obliged by law to protect the personal data privacy and the confidentiality of the medical record of its clients. For this purpose, the company takes special care to anonymize the results of the queries executed by the auditor in order to ensure that no client information is accidentally exposed.

Providing the external auditor with direct access to the database of the company raises a series of concerns about a) ensuring the security of the company-internal data, and b) the performance of the company systems, as an indirect result of the unpredictable additional load imposed by the complex queries executed by the auditor. As a solution to these issues, it is proposed to use a public Cloud data hosting solution provider and migrate a consistent replica of the financial transaction records to the public Cloud, stripped of any personal data. The auditing company would then be able to retrieve the necessary information without burdening the company systems. Such a migration to the Cloud however, even if only partial, requires addressing different kinds of challenges: confidentiality-related (ensuring that it is impossible to recreate the medical records and other personal information of the company clients using the data in the public Cloud), functionality-related (providing both all the necessary data and the querying mechanisms for the auditor to operate as required), and non-functional (ensuring that the partial migration does not encumber in any way the performance of the company systems). The following sections discuss a series of Cloud Data Patterns that address these issues.

III. Functional Patterns

Cloud data stores can be considered as appliances where a fixed set of functionality is provided [12]. Cloud data stores include SQL and NoSQL solutions [13], [14]. Each solution is geared towards a specific application domain and therefore does not come with all possible features. Furthermore, the offered functionalities may be configurable but not extensible. Functional Cloud Data Patterns provide solutions for these challenges. More specifically:

A. *Data Store Functionality Extension*

The *Data Store Functionality Extension* pattern adds a missing functionality to a Cloud data store.

Context: A Cloud data store does not inherently support all functionalities usually offered by a traditional data store. For instance, the Cloud data store might not support data joins. The choice of which data store to use is fixed by the application requirements or contractual obligations and therefore it is not possible to replace the data store with an equivalent one offering the missing functionality.

Challenge: How can a Cloud data store provide a missing functionality?

Forces: The missing (but required) functionality might be implemented in the business layer. An example of missing functionality are joins. Implementation of the missing functionality on a higher application layer requires all data to be retrieved from the database layer and leads to increased network load.

Solution: A component implements the required functionality as an extension of the data store, either by offering an additional functionality, or by adapting one or more of the existing functionalities offered by the data store. The extension component is placed within the Cloud infrastructure of the Cloud data storage. A low distance (in terms of network performance) ensures low latency between the extension and the data store.

Sidebars: The additional or extended functionality code has to be wrapped into an application, which can be hosted in the Cloud. The access to the data store of the Cloud provider from this application is done via the API supplied by the provider. The code in the data access layer has to be adjusted accordingly, denoted by “Data Access Layer*” in Figure 2 case (a). This means that each data access call using the required functionality has to be replaced by a call to the component implementing the corresponding data store functionality extension.

Results: The Cloud data store functionality is extended. Assuming all additional functionality required (and not provided by the Cloud data store) can be implemented within the component implementing the functionality extension, there is no adjustment or modification of the business layer required.

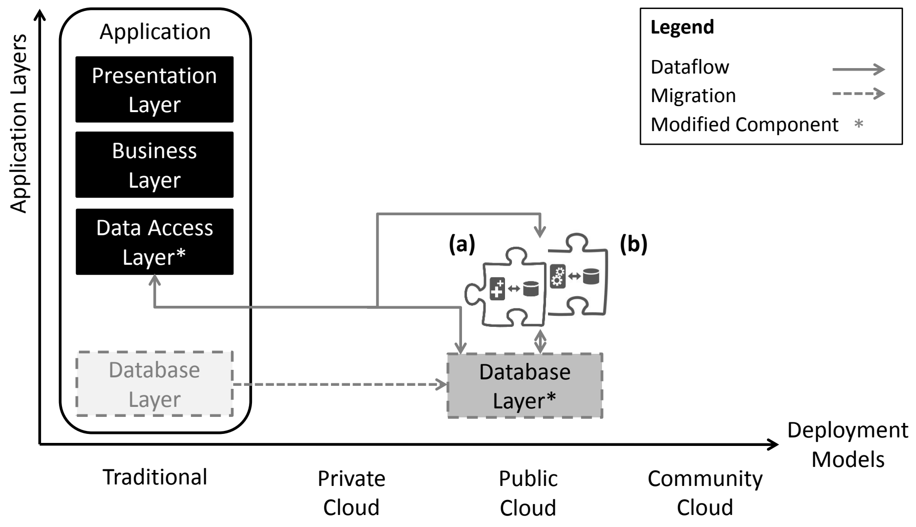


Figure 2: Sketch of (a) Data Store Functionality Extension and (b) Emulator of Stored Procedures

Example: The database layer of an application built on Oracle Corporation MySQL version 5.1 [6], is moved to Google App Engine Datastore [5] by Google Inc. As a result, the database layer functionality is incompatible, because the Google Query Language [7] supports only a subset of the functionality provided by SQL, e. g., join functionality is not supported. As the application requires join functionality, this additional functionality has to be provided by the component implementing the join functionality.

Next: In case functionality for stored procedures has to be added, the "Emulator of Stored Procedures" pattern has to be considered.

B. Emulator of Stored Procedures

The *Emulator of Stored Procedures* pattern is a special case of the Data Store Functionality Extension pattern see case (b) in Figure 2, where an extension component is built outside the data store, containing a set of predefined groups of commands to be executed by the data store. While this is a very common mechanism in traditional data stores, many Cloud data stores do not support it natively. Due to its ubiquity and usefulness we therefore define a separate pattern for it.

Context: Stored procedures "are application programs that execute within the database server process" [15]. A Cloud data store does not inherently support stored procedures as most traditional data stores do. Changing the provider of the Cloud data store might not be an option, because of other advanced features provided or due to specific customer requirements.

Challenge: How can a Cloud data store not supporting stored procedures provide such functionality?

Forces: To keep network traffic low, the number of requests to the database layer should be minimized. Thus, the stored procedure code should not run on-premise, but within the Cloud infrastructure of the Cloud data store.

Solution: An emulator of stored procedures is placed within the Cloud infrastructure of the Cloud data storage. A low distance (measured in terms of network performance) ensures low latency between the emulator and the data store and reduces communication overhead.

Sidebars: The stored procedure code has to be wrapped into an application, which can be hosted in the Cloud. The access to the data store of the Cloud provider is done via the API supplied by the provider. The code of the data access layer has to be adjusted accordingly, denoted by "Data Access Layer*" in Figure 2 case (b). This means that each call to the stored procedure has to be replaced by a call to the emulator.

Results: Instead of emulating the stored procedure functionality in the business layer, the functionality is provided as an application in the Cloud. The number of requests from the data access layer to the database layer is reduced as the work is done by the stored procedure emulator. Assuming the data transfer between nodes in the provider's Cloud infrastructure is free and the stored procedure emulator is hosted there, then costs are also reduced.

Example: The database layer of an application built on Microsoft SQL Server with stored procedures is moved to Microsoft SQL Azure Database. As Microsoft SQL Azure does not support full text search stored procedures [16], this functionality has to be emulated.

Next: In case more functionality has to be added, the "Data Store Functionality Extension" pattern has to be considered.

IV. Non-Functional Patterns

We previously investigated non-functional patterns with focus on providing solutions for ensuring an acceptable Quality of Service (QoS) level by means of scalability in case of increasing data read or data write load [17]. In the following, we provide an overview on these non-functional patterns focusing on their *context*, *challenge*, and *solution*. When considering the data rather than the database system, there are two scaling options available: vertical and horizontal data scaling. Vertical data scaling can be achieved by moving the data to a more powerful database system, which offers better performance, advanced functionalities, or both. Horizontal data scaling is based on partitioning the data according to functional groups [18]. Examples for functional groups are European customers and American customers. Each functional group may be itself distributed among different database systems to increase search speed. This method is also called *sharding* [19].

A. Local Database Proxy

The *Local Database Proxy* enables read scalability by requiring a master/multiple slave model and forwarding read requests to any read replica.

Context: A Cloud data store does not inherently support horizontal scalability for data reads. When the data read load of the application permanently increases, e. g., due to increased user acceptance and usage, a mechanism for horizontally scaling read requests is required. Additionally, the business logic of processing user requests is also moved to the Cloud.

Challenge: How can a Cloud data store not supporting horizontal data read scalability provide that functionality?

Solution: The Cloud data store is configured using a single master/multiple slave model. The master handles data writes and the slaves are used as replicas serving read requests only. In case the application has to deal with stale data, the replication of data may be lazy. A proxy component is locally added below each data access layer. All requests from each data layer are routed through the respective proxy. The proxy routes data read requests to any slave and write requests to the master.

B. Local Sharding-Based Router

The *Local Sharding-Based Router* enables read and write scalability by requiring the independent splitting and distributing of data into functional groups and forwarding read and write requests to the corresponding shard.

Context: A Cloud data store does not inherently support horizontal scalability for data reads and writes. When the data read load of the application permanently increases, e. g., due to increased user utilization, a mechanism for horizontally scaling read requests to the data store is required. Furthermore, a permanent high data update rate of the application requires also horizontally scaling for data writes. The business logic of processing user requests is moved to the Cloud.

Challenge: How can a Cloud data store not supporting horizontal data read and write scalability provide that functionality?

Solution: The data to be stored in the Cloud are split horizontally. This means that tables with many rows are split into several data stores. Each data store is assigned a distinct number of rows of the original table. This technique is called "sharding" [19]. A dedicated sharding-based router is added locally below each data access layer. All requests from each data layer are routed through the respective sharding-based router. The local sharding-based router forwards data read and write requests to the appropriate Cloud data store.

V. Confidentiality Patterns

In our previous work [10], we presented Cloud Data Patterns for confidentiality. They deal with data that have to be kept private and secure, commonly referred to as "critical data". In the following, we present and update these patterns focusing on their *context*, *challenge*, and *solution*.

A. Confidentiality Level Data Aggregator

Critical data can be categorized into different confidentiality levels. As data are not always categorized by confidentiality, or categorized using different confidentiality categorizations, the confidentiality level has to be harmonized. The *Confidentiality Level Data Aggregator* provides one confidentiality level for data from different sources with potentially different confidential categorizations on different scales.

Context: The data formerly stored in one traditionally hosted data store is separated according to the different confidentiality levels and stored in different locations. The business layer is separated into the traditionally hosted part processing the critical data, and the part hosted in the public Cloud processing the non-critical data. As the application accesses data from several data sources, the different confidentiality levels of the data items have to be aggregated to one common confidentiality level. This builds the basis for avoiding disclosure of critical data by passing it to the public Cloud.

Challenge: How can data of different confidentiality levels from different data sources be aggregated to one common confidentiality level?

Solution: An aggregator retrieves data from all Cloud data stores. The aggregator is placed within the Cloud infrastructure of the Cloud data storage with the highest confidentiality level. Since it must be able to process data with the highest confidentiality level, it may not be placed where data with a lower level of confidentiality reside. As a consequence, the aggregator has to be placed in a location where the demands of the highest confidentiality level are fulfilled.

B. Confidentiality Level Data Splitter

The *Confidentiality Level Data Splitter* splits data according to pre-configured privacy levels. This is required when an application writes data to multiple data stores with different confidentiality levels.

Context: The data formerly stored in one traditionally hosted data store is separated between data stores with different confidentiality levels. As the application writes data to several data stores, the data have to be categorized and split according to their confidentiality level. This builds the basis for avoiding disclosure of critical data when storing them in the public Cloud.

Challenge: How can data of one common confidentiality level be categorized and split into separate data parts belonging to different confidentiality levels?

Solution: A splitter is placed within the infrastructure of the data access layer of the application. Thus, additional data movement, network traffic, and load can be minimized. The splitter writes data to all Cloud data stores. As the splitter processes data with the highest confidentiality level, it has to be placed in a location where the demands of the highest confidentiality level are fulfilled.

C. Filter of Critical Data

The *Filter of Critical Data* ensures that no confidential data are disclosed to the public. The filter enforces that no data leaves the private Cloud by filtering out critical data.

Context: The private Cloud data store contains both critical and non-critical data. To prevent disclosure of the critical data, it has to be enforced that the critical data do not leave the private Cloud. The logic implemented in the business layer is split into one part processing critical and one part processing non-critical data. The party implementing and/or hosting the business logic for processing the non-critical data cannot be trusted.

Challenge: How can data-access rights be ensured when moving the database layer into the private Cloud together with a part of the business layer, as well as a part of the data access layer to the public Cloud?

Solution: A filter for the critical data is placed within the infrastructure of the private Cloud data store. All requests to the private Cloud data store have to be directed to the filter. The private Cloud data store is only reachable through the filter. Requests for critical data originating off-premises are denied by the filter.

D. Pseudonymizer of Critical Data

The *Pseudonymizer of Critical Data* implements pseudonymization. Pseudonymization is a technique to provide a masked version of the data to the public while keeping the relation to the non-masked data in private [20]. This enables processing of non-masked data in the private environment when required.

Context: The private Cloud data store contains critical and non-critical data. The business layer is partially moved to the public Cloud and needs access to data. The logic implemented in the business layer is split into one part requiring critical data, and one where critical data in pseudonymized form is sufficient for processing. The party implementing and/or hosting the business logic for processing pseudonymized data may not be trusted. Furthermore, passing critical data may be restricted by compliance regulations. It also is required to be able to relate the pseudonymized data processing results from the public business layer back to the critical data.

Challenge: How can a private Cloud data store ensure passing critical data in pseudonymized form to the public Cloud?

Solution: A pseudonymizer of data is placed within the infrastructure of the private Cloud data storage. All requests to the private Cloud data storage have to be directed to the pseudonymizer. The private Cloud data storage is only reachable by the pseudonymizer. Results of requests for critical data originating off-premises are pseudonymized.

E. Anonymizer of Critical Data

The *Anonymizer of Critical Data* implements anonymization [20]. Anonymization is a technique to provide a reduced version of the critical data to the public while ensuring that it is impossible to relate the reduced version to the critical data.

Context: The private Cloud data store contains both critical and non-critical data. The business layer is partially moved to the public Cloud and needs access to data. To prevent disclosure and misuse, the critical data are anonymized before being passed to the public Cloud. The logic implemented in the business layer is split into one part requiring critical data, and one where critical data in anonymized form are sufficient for processing. The party implementing and/or hosting the business logic for processing the anonymized data cannot be trusted. It is not required to be able to relate the anonymized data processing results from the public business layer back to the critical data.

Challenge: How can a private Cloud data store ensure passing critical data in anonymized form to the public Cloud?

Solution: An anonymizer is placed within the infrastructure of the private Cloud data store. All requests to the private Cloud data store have to be directed to the anonymizer. The private Cloud data store is only reachable through the anonymizer. Results of requests for critical data originating off-premises are anonymized.

For more details on these patterns, the interested reader is referred to [10]. In the following sections we combine these patterns with the ones we defined in Section III and Section IV in order to demonstrate how they can be used in practice.

VI. Cloud Data Patterns in Practice

Table I provides an overview of the impact created by the application of the patterns presented in the previous sections to the various application layers. Table I distinguishes between the layer the patterns are supposed to be realized in, and the ones that may require additional modifications as a result of applying them.

As the functional patterns are used in order to add additional functionality to a Cloud data store, they are realized in the database layer. In case the extended functionality should be used for a data request, the request has to go through the realization of the functional pattern. Thus, adaptations of the data access layer are also required.

Non-functional and confidentiality patterns are supposed to be realized in the data access layer. The confidentiality patterns Confidentiality Level Data Aggregator, Filter of Critical Data, Pseudonymizer of Critical Data, and Anonymizer of

TABLE I: Relation between Cloud Data Patterns and Application Architecture Layers

Cloud Data Patterns / Application Layers	**Business Layer**	**Data Access Layer**	**Database Layer**
Data Store Functionality Extension	⊘	△	◇
Emulator of Stored Procedures	⊘	△	◇
Local Database Proxy	⊘	△◇	△
Local Sharding-Based Router	⊘	△◇	△
Confidentiality Level Data Aggregator	△	△◇	⊘
Confidentiality Level Data Splitter	⊘	△◇	⊘
Filter of Critical Data	△	△◇	⊘
Pseudonymizer of Critical Data	△	△◇	⊘
Anonymizer of Critical Data	△	△◇	⊘

Legend: ⊘ has no impact on, ◇ is realized in, △ requires adaptations to

Critical Data require also adaptations of the business layer of the application. This is because, by realizing the corresponding patterns, the business logic might not have the same view on the data as before since the business layer has to deal with data in aggregated, filtered, pseudonymized or anonymized form.

As patterns are related to each other — to be considered as a whole and to be composable [11], we have chosen the form of a piece of a puzzle for the pattern icons. Whether two or more Cloud Data Patterns are composable depends on the semantics and functionality of each of the patterns. Moreover, the specific requirements and context of the needed solution effect whether a composition of patterns is required. Thus, we do not claim that all Cloud Data Patterns are composable with each other. A deeper investigation under which conditions a composition of Cloud Data Patterns is possible, and what are the resulting semantics, is required. The investigation and results leading to a Cloud Data Pattern language are part of our future work.

In the following, we discuss how the functional, non-functional, and confidentiality patterns can be used in practice based on the motivating scenario introduced in Section II. Figure 3 provides an overview of the realization of the scenario using Cloud Data Patterns. More specifically, in order to provide horizontal scalability for reads and writes to the data of the clients and their transactions, the Local Sharding-Based Router pattern is used. The data are separated between the two data centers according to geographical location.

The Google App Engine Datastore is chosen for outsourcing the storage of the data on financial transactions, configured accordingly. The client information and their corresponding medical records are critical data to be kept only in the company's private Cloud. Financial transactions information will appear both in the private and in the public Cloud. In order to keep both the data stored in the private data store and the one outsourced to the public Cloud consistent, data updates and inserts should be done in parallel to both the private and the public part of the database layer. However, only a part of the financial data is necessary for the auditing (e. g., client names can be removed) and the remaining can be pseudonymized before moving to the public Cloud (e. g., bank account numbers replaced by serial IDs). A composition of the Filter of Critical Data and the Pseudonymizer of Critical Data is used to fulfill both requirements. The filter is configured so that only data on financial transactions pass it. After passing the filter, the information on financial transactions is pseudonymized before it is stored in the public Cloud. Storage of data on the auditing company side can be done either in a private Cloud or in the traditional manner; this is out of the scope of our discussion.

Due to the challenges identified in the motivating example regarding the data access of the auditing company, the query results must not contain any relations concerning clients and their corresponding medical records or personal data. Therefore, this information has to be completely deleted before passing the query results to the auditing company (instead of simply obfuscating this relation by using pseudonymization). In addition, the queries to be executed by the auditor have to be agreed upon in advance. For these purposes, the realization uses a composition of the Anonymizer of Critical Data and the Emulator of Stored Procedures patterns. The emulator is used to predefine and restrict the data queries allowed to be executed by the auditing company. The Anonymizer of Critical Data additionally ensures the removal of any critical information from the results of the queries. A combination of functional, non-functional, and confidentiality patterns can therefore be used in tandem to address the requirements of the use case posed by the motivating scenario.

VII. RELATED WORK

Pattern languages defining reusable solutions for recurring challenges in architecture have been first proposed by Christopher Alexander [21]. A series of well-established patterns have been previously identified concerning, e. g., software engineering [22], enterprise integration [11] and application architecture [2]. Such general works do not consider building or migrating the database layer in the Cloud. Nevertheless, we reuse the pattern format defined by Hohpe and Woolf [11] for describing our Cloud Data Patterns.

Petcu [23] proposes Cloud usage patterns for Cloud-based applications based on existing use cases. Fehling et al. [24] and Pallmann [25] provide high-level architectural patterns to design, build, and manage applications using Cloud services. None of these works discusses patterns for building and/or moving the data layer to the Cloud. Adler [12] provides contributions regarding best practices for scalable applications in the Cloud. In this paper we reuse some of the results presented in [12] to form the non-functional patterns presented in Section IV.

ARISTA Networks, Inc. [26] provides seven patterns for Cloud computing of which only one (the Cloud Storage pattern)

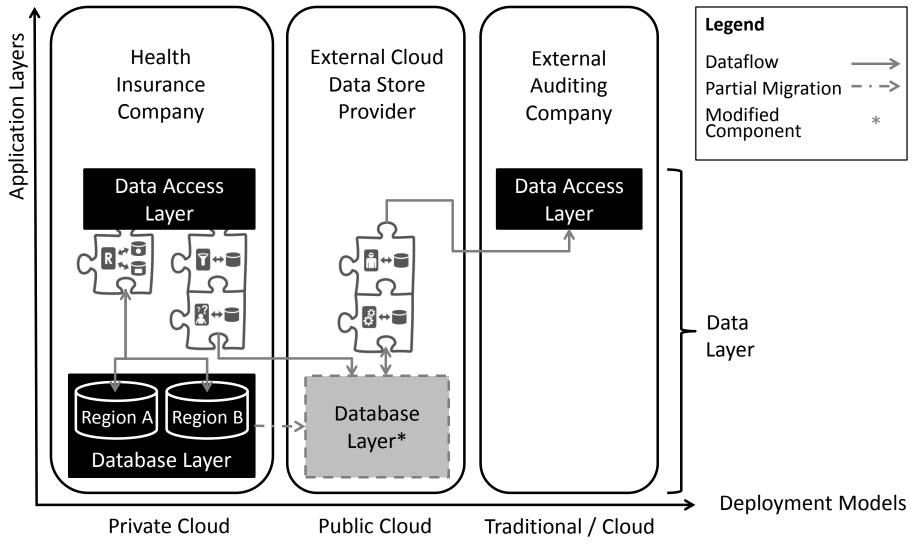


Figure 3: Realization of motivating scenario using Cloud Data Patterns (Data Layer)

deals with data in the Cloud. Nock [27] provides patterns for data access in enterprise applications, without however treating Cloud data stores in the same manner as we do.

Schumacher et al. [28] present reusable solutions for securing applications, but do not deal with data pseudonymization, data anonymization, and data filtering. Hafiz [29] presents a privacy design pattern catalog consisting of nine patterns achieving anonymity by mixing data with data from other sources instead of providing a general pseudonymization, anonymization, or filtering pattern. Creese et al. [30] consider design patterns for data protection of Cloud services. Romanosky et al. [31] describe privacy patterns applicable for online interactions. Schumacher [32] introduces an approach for mining security patterns from security standards and presents two patterns for anonymity and privacy. These works do not consider building a data layer in the Cloud or migrating an existing one there; some of the mechanisms identified however (e. g., pseudonymization) are reused in the Cloud Data Patterns we propose.

Finally, Schuemmer [33] presents patterns filtering personal information to establish boundaries for interactions between users utilizing collaborative systems. Our patterns are more general in the sense that they are not limited to filtering of personal data.

VIII. Conclusions and Future Work

This work presented a set of reusable solutions to face the challenges of moving the data layer to the Cloud or designing an application using a data store in the Cloud. The challenges and proposed solutions were organized as a non-exhaustive catalog of Cloud Data Patterns focusing on the PaaS delivery model. These patterns are the result of our collaboration with industry partners and research projects. Patterns for functional, non-functional, and confidentiality issues were discussed and shown how they can be combined in order to address a use case in practice.

The presentation of the patterns focused on the design issues, rather than the underlying technical challenges, in order to ensure their applicability across different technological platforms. This means that issues requiring a deeper technical insight, like for example scalability, are not covered sufficiently in the scope of this work. Nevertheless, these are issues we are currently looking into. For example, in the discussion in Section VI, implementing the Local Sharding-Based Router as a single component may result in a bottleneck for scalability, or even to a complete failure of the data access/database layer connection. A possible scalability enabling mechanism, and a counter-measure to single points of failure is to implement each pattern using a hot-pool of pattern realizations in the Cloud. A hot-pool consists of multiple instances of the realization component and a watchdog. Such issues and their possible solutions are investigated as part of a larger scale evaluation of our patterns using an industrial case study. Toward this direction, we also plan to formalize a general composition method of Cloud Data Patterns and expand our catalog with identified patterns presented here.

Acknowledgments

The research leading to these results has received funding from projects 4CaaSt (grant agreement no. 258862) and Allow Ensembles (grant agreement no. 600792) part of the European Union's Seventh Framework Programme (FP7/2007-2013), and the BMWi-project Cloud-Cycle (01MD11023). We thank Tobias Unger for his valuable input.

References

[1] M. Armbrust *et al.*, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, 2009.

[2] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[3] P. Mell and T. Grance, "Cloud Computing Definition," *National Institute of Standards and Technology*, July 2009.

[4] S. Strauch, O. Kopp, F. Leymann, and T. Unger, "A Taxonomy for Cloud Data Hosting Solutions," in *Proceedings of the International Conference on Cloud and Green Computing (CGC '11)*. IEEE Computer Society, Dezember 2011, pp. 577–584.

[5] Google, Inc., “Google App Engine Datastore,” 2011.

[6] Oracle Corporation, “MySQL,” 2011, http://www.mysql.com 31.03.2013.

[7] Google, Inc., “Google App Engine GQL Reference,” 2011, https://code.google.com/intl/en/appengine/docs/python/datastore/gqlreference.html 31.03.2013.

[8] B. C. Tak, B. Urgaonkar, and A. Sivasubramaniam, “To Move or Not to Move: the Economics of Cloud Computing,” in *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud’11. Berkeley, CA, USA: USENIX Association, 2011.

[9] M. Menzel and R. Ranjan, “CloudGenius: Decision Support for Web Server Cloud Migration,” in *Proceedings of WWW ’12*. New York, NY, USA: ACM, 2012, pp. 979–988.

[10] S. Strauch, U. Breitenbücher, O. Kopp, F. Leymann, and T. Unger, “Cloud Data Patterns for Confidentiality,” in *Proceedings of CLOSER’12*. SciTePress, April 2012, pp. 387–394.

[11] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[12] B. Adler, “Building Scalable Applications In the Cloud: Reference Architecture & Best Practices, RightScale Inc.” 2011, http://www.rightscale.com/info_center/white-papers/building-scalable-applications-in-the-cloud.php 31.03.2013.

[13] C. Strauch, “NoSQL Databases,” February 2011, http://www.christof-strauch.de/nosqldbs.pdf 31.03.2013.

[14] Stefan Edlich, “List of NoSQL Databases,” July 2011, http://nosql-database.org 31.03.2013.

[15] P. A. Bernstein, *Principles of Transaction Processing (Morgan Kaufmann Series in Data Management Systems)*, 2nd ed. Morgan Kaufmann, 2006.

[16] Microsoft, “System Stored Procedures (SQL Azure Database),” August 2011, http://msdn.microsoft.com/en-us/library/ee336237.aspx 31.03.2013.

[17] S. Strauch, V. Andrikopoulos, U. Breitenbücher, O. Kopp, and F. Leymann, “Non-Functional Data Layer Patterns for Cloud Applications,” in *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom’12)*. IEEE Computer Society Press, Dezember 2012, pp. 601–605.

[18] K. Küspert and J. Nowitzky, “Partitionierung von Datenbanktabellen,” *Informatik-Spektrum*, vol. 22, pp. 146–147, 1999.

[19] J. Zawodny and D. Balling, *High Performance MySQL: Optimization, Backups, Replication, Load-balancing, and More*. O’Reilly & Associates, Inc. Sebastopol, CA, USA, 2004.

[20] Federal Ministry of Justice, “German Federal Data Protection Law,” December 1990, http://www.gesetze-im-internet.de/bdsg_1990/ 31.03.2013.

[21] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language. Towns, Buildings, Construction*. Oxford University Press, 1977.

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

[23] D. Petcu, “Identifying Cloud Computing Usage Patterns,” in *2010 IEEE International Conference on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*. IEEE, October 2010.

[24] C. Fehling, F. Leymann, R. Retter, D. Schumm, and W. Schupeck, “An Architectural Pattern Language of Cloud-based Applications,” in *Proceedings of PLoP’11*. ACM, October 2011.

[25] D. Pallmann, “Windows Azure Design Patterns,” 2011, http://www.windowsazure.com/en-us/develop/net/architecture/ 31.03.2013.

[26] ARISTA Networks, Inc., “Cloud Networking: Design Patterns for Cloud-Centric Application Environments,” January 2009.

[27] C. Nock, *Data Access Patterns: Database Interactions in Object Oriented Applications*. Prentice Hall Professional Technical Reference, February 2008.

[28] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating Security and Systems Engineering*. Wiley, 2006.

[29] M. Hafiz, “A Collection of Privacy Design Patterns,” in *Proceedings of PLoP’06*, New York, NY, USA, 2006.

[30] S. Creese, P. Hopkins, S. Pearson, and Y. Shen, “Data Protection-Aware Design for Cloud Services,” in *Proceedings of CloudCom’09*, 2009, pp. 119–130.

[31] S. Romanosky, A. Acquisti, J. Hong, L. F. Cranor, and B. Friedman, “Privacy Patterns for Online Interactions,” in *Proceedings of PLoP’06*. ACM, 2006.

[32] M. Schumacher, “Security Patterns and Security Standards,” in *Proceedings of the 7th European Conference on Pattern Languages of Programs (EuroPLoP)*, July 2002.

[33] T. Schuemmer, “The Public Privacy–Patterns for Filtering Personal Information in Collaborative Systems,” in *Proceedings of the Conference on Human Factors in Computing Systems (CHI’04)*, 2004.

A Factor Model Capturing Requirements for Generative User Interface Patterns

Stefan Wendler, Danny Ammon, Ilka Philippow, Detlef Streitferdt
Software Systems / Process Informatics Department
Ilmenau University of Technology
Ilmenau, Germany
{stefan.wendler, danny.ammon, ilka.philippow, detlef.streitferdt}@tu-ilmenau.de

Abstract — **The lowering of efforts for the adaptation of GUI dialogs to changing business processes is still a worthwhile goal. In this context, user interface patterns (UIPs) have been introduced in the development of user interfaces to increase both usability and reusability. Originally derived from human computer interaction patterns, UIPs are generative and thus have to be formalized. Recent approaches for model-based GUI development employ UIPs with specific notations. These UIP concepts have not yet been evaluated on the basis of a stringent set of criteria. We elaborate detailed requirements for generative UIPs. The resulting influence factor model is used to assess recent UIP approaches and identify open issues.**

Keywords — user interface patterns; model-based user interface development; HCI patterns; graphical user interface.

I. Introduction

A. Motivation

Domain. Nowadays, companies heavily rely on systems that offer a vast support for the activities defined in business processes. To serve their purpose effectively, those business information systems provide graphical interaction dialogs to various users. Besides the enormous effort to be taken into the specification of the business processes and related requirements, there are also considerable high costs involved in the development of GUI dialogs the user interacts with in order to process certain activities of the business process. In addition, those dialogs need to be matched with their currently assigned workflow derived from the respective business processes. As business information systems must be changed over time, the need to keep business processes, application kernel functions and the GUI, that provides the dialogs based thereupon, in correspondence [1] has arisen.

Problem. Approaches have been proposed that aim at raising both efficiency and reuse by applying model- and pattern-based concepts for the development of GUIs and dialog structures derived from task models. The different concepts have not been verified yet. Currently, there is no detailed set of requirements, which can be used as foundation to assess the pattern concepts employed for GUI generation.

B. Objectives

We review the state of the art of model-based development processes employing generative user interface patterns (UIPs) and present answers to following questions:

- What requirements have to be addressed by a general definition for generative UIPs applied for GUIs?
- What are the capabilities and limitations of current generative UIP concepts concerning reusability and variability?

Our main focus lies on the last question, so we formulate requirements for a UIP definition on the presentation level. After we analyze current UIP issues, we apply a customized Global Analysis [2] to derive the factors, which bear the great impacts on the definition and application of generative UIPs. As a result, we continue and detail our previous work [3][4] on initial requirements associated to generative UIPs.

C. Structure of the Paper

The next section provides a brief overview to GUI development and UIPs. In Section III, we establish a factor model that captures major requirements for UIPs. The model-based development approaches are described in Section IV and are analyzed on the basis of the factor model. In Section V, we express our findings and conclusions.

II. Related Work

A. Graphical User Interface Development

GUI architecture patterns. Besides the requirements and software architecture to be changed harmonically, a new implementation of dialogs induces high costs as reuse of existing code is hardly possible. More precisely, the GUI system may be composed of proven architecture patterns that enable separation of concerns and reduced dependencies like the Quasar client architecture [8], but these kinds of patterns are restricted to non-visual aspects of the GUI like event and data processing or the communication with a workflow system. As far as visual and closely associated interaction design aspects are concerned, the common patterns do not posses the means to offer the desired aspects of reuse. Moreover, the usability is crucial for dialogs, as it affects how quickly users are able to learn to use new features of the GUI and how efficiently they will perform reoccurring tasks. Usability also is not covered by architectural patterns.

GUI-generators. Generators have been applied for a longer period now and could not fill the gap, since they can only cover dialogs that allow a realization based on fixed layout and interaction definitions. Besides the visual and interactive aspect, GUI-generators often were based on information provided by the domain model, so that task models or other process definitions could not be sourced for the generation of dialogs with acceptable usability.

B. User Interface Patterns

To overcome the high efforts and permit higher reusability along with proven usability, patterns of human computer interaction (HCI) have been integrated as model artifacts in model-based development. In that environment HCI patterns had to be formalized in order to obtain a machine processable format. Called generative patterns by Vanderdonckt and Simarro [6], a new form of pattern has emerged based on descriptive HCI patterns. Commonly, these patterns are named User Interface Patterns (UIPs).

Reusability. UIPs as generative patterns are to be deployed as reusable entities in GUI development. By specifying dialog parts abstractly (visual parts and interaction) as well as parameters for variability, UIPs should facilitate the reuse and automated generation of GUI dialogs. Configured accordingly, the UIPs would be instantiated to target contexts. This way, a GUI system should be compiled by the selection and combination of chosen UIPs. Key features of this approach shall be the variable application of UIPs to any appropriate context and their ability to form hierarchies of further cascading UIP instances. The latter could form a context-specific composition of already specified appearance and behavior qualities, which would be quantitatively adapted to the context when instantiated.

Issues. The application of UIPs for GUI generation has successfully been probed by past research [1][11][14][19][24]. As HCI patterns need to be augmented for automatic deployment, the main issue of finding a suitable formalization format, which offers a feasible definition of generative UIPs, has arisen. Current approaches propose different UIP concepts combined with tools, which propagate the instantiation of the abstract UIP entity for various contexts and thus an increase in reuse. Nevertheless, reusability is still restricted to a limited set of UIPs, which can be deployed without having to consider all variability aspects [3]. The potential variations for view, interaction and in particular the control aspect are so extensive that they need to be further detailed by a set of comprehensive criteria.

III. REQUIREMENTS FRAMEWORK

UIP definition. The specification of UIPs is impaired by a fundamental problem that persists in the lack of a dedicated definition for this generative artifact. Many sources have been published on HCI or GUI related patterns, but these either presented no or did not converge towards a unified definition. We stick to our drafted definition in [3] and use the term User Interface Pattern (UIP) that addresses the generative form ready to be instantiated to a certain GUI context. So, a UIP is settled in close proximity to architecture and code artifacts assuming presentation responsibilities.

Approach. To overcome the disunity concerning the definition and features of UIPs, we develop a system of requirements that is able to express the conception of UIPs independently from any employment in modeling frameworks and tools. We apply the Global Analysis [2], as requirements for UIPs are rather general. So, we refine them according to their impact on the generative UIP artifact definition. The background and an initial factor model have been developed in [4], which is detailed in the following.

A. Criteria for User Interface Patterns

As outlined in [3], sufficient solutions for pattern-based GUI development have to meet basic criteria. Firstly, they must enable reusability in the context of vast variability of stored patterns. Secondly, facilities must permit to compose several patterns to form a hierarchy of GUI components - an attribute that is not common for all kinds of software patterns. Lastly, the instantiation into varying user interface paradigms, platforms and types should be possible.

The first two criteria are relevant for our scope and we will decompose them in our factor model as we progress towards Section III.E. For now, the factor "Reusability of UIPs" is defined, which is composed by the three factors "Structural composition ability", "Behavioral composition ability" and "Variability of UIP instances". The split nested factors are motivated by the following distinction. A single UIP may be reused for many contexts and for that purpose, certain variability concerns have to be met that are covered in the next section. Besides, a combination of more than one UIP may be reused. In that case, both the structural and behavioral definitions should be adaptable to the desired context. Section III.C treats these composition ability factors.

B. Variability of User Interface Patterns

MVC analogy. If one UIP is variably instantiated, implementations of given architecture components evolve and eventually differ in certain aspects. For this reason, the architectural pattern of model-view-controller (MVC) is used to describe the UIP adaptability for different contexts [3]. An UIP adaptation changes the actual view structure, data types for the view parts and the control serving visual and application event handling of a certain architecture instance.

Variability factor. The above mentioned variability concerns affect various contents of an instance of a certain adaptable UIP. The content is materialized by the two aspects *view* and *interaction* in the factor model. Each sub-factor of *variability* is operationalized by an aspect. Besides, the *variability* factor influences a second dimension, which describes the moment in time, when the UIP adaptation takes place. Thus, the *configuration* factor details *variability.*

C. Aspects of User Interface Patterns

Purpose. Originally, we described three aspects of UIPs to detail our definition in [3]. We pointed out the differences between a concrete specification of a GUI unit, the abstracted formalization of a UIP and its instances. Here, we summarize the aspects to further evolve the factor model.

View. By the stereotype but abstract *view* of a UIP, selection, arrangement and types of user interface controls (UI-Controls) are defined. With its abstract definition the "view aspect" preserves the applicability of a UIP to various contexts and should not rely on certain GUI frameworks, hence a UIP must be able to be transformed to desired platforms. Through the "view aspect", UIPs can be categorized into simple and composite patterns. Simple UIPs, like a simple search [10], consist of a fixed set of UI-Controls, while composite UIPs, like an advanced search [10], contain even other UIPs. Therefore, the "Structural composition ability" is operationalized by the "view aspect". To define the *visual element structure* of a UIP, a developer may source both UI-Controls and already defined UIPs.

Interaction. A user always perceives and performs interactions with instances of a certain UIP in the same way. Combined with the *view*, the *interaction* forms the general purpose of a UIP and so, both aspects constitute the reusable entity and distinguish UIPs from mere UI-Control compositions. With *interaction* states, data handling and presentation related events are defined by referring to *view* contents. Moreover, a UIP may demand for structural *view* states that are determined at run-time by user inputs.

Control. Composite UIPs, as defined above, actuate in- and outputs depending on the defined selection, instantiation, and configuration of their child UIPs. Sections III.D and III.E treat how *control* operationalizes "Behavioral composition ability" and in this regard details the *interaction* of several UIPs in one *view structure unit*. Depending on the *variability configuration* dimension, a dynamic *control* may be needed where child UIPs are selected and instantiated at runtime. The following section covers this case.

Reusability factor. *View*, *interaction* and *control* aspects operationalize the before-mentioned *reusability* factor. All three factors ensure either the *composition abilities* or *variability* of UIPs. The reuse of single UIPs for different contexts is achieved by abstraction in both the structure of the *view* and the dynamics of the *interaction* as well as *parameters* that provide instance-specific information.

D. *Architecture Experiments*

Architecture. For the GUI architecture, we assume a structure to be established in analogy to Figure 1, which was derived from [9] and altered for our scope. Notably is the distinction of three controllers for presentation, dialog and task. The *PresentationController* queries data from technical *GUI Framework* objects, receives technical events from them, adapts the *DialogVisuals* accordingly and finally forwards events relevant for the application state to the *DialogController*. The responsibility of the latter is to implement application logic, query data from and send data to the *ApplicationKernel* after selection based on the *Model* data. Additionally, the *DialogController* decides on the lifecycle of the *Dialog*, as it evaluates the state of the *Model* and events received from the *View*. Acting as a factory, the *DialogConfiguration* builds the *Dialog* composition unit, and for that purpose, communicates with the *TaskController*, which initiates the creation or deletion of dialogs.

The architecture is detailed, since a *Dialog* can be based on composite UIPs. A child UIP affects the *View* component only, while the superior one triggers *DialogController* actions, when new sub-dialogs or data must be loaded. Thus, the factor model lists presentation and dialog action-binding.

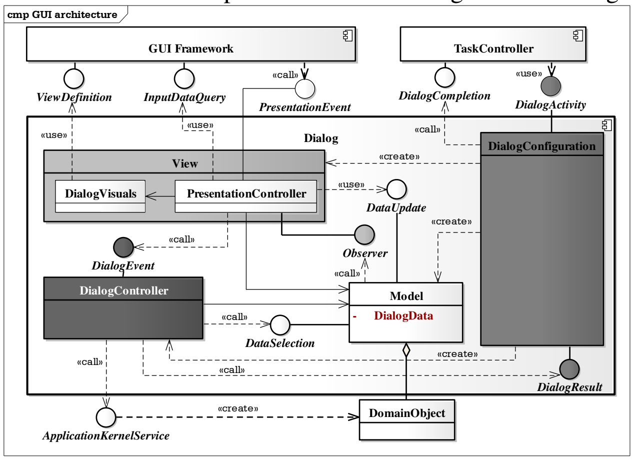


Figure 1. GUI architecture reference model

Experiments. We presented two architecture concepts to implement UIPs specified with UIML (User Interface Markup Language) in [5]. The main findings of our experiments were that UIPs supporting the criteria in III.A could not be formalized as single artifacts with UIML. This was due to the complex example, which required for a "control aspect" with dynamic *configuration*.

The advanced search UIP [3] holds a certain number of search criteria, each demanding a certain UIP type, e.g., a price range and a date represent two different search criteria. The states such a composite UIP can adopt cannot be enumerated by a static specification as they depend on user input or another context not known at design-time. Figure 2 illustrates an example of an advanced search UIP instance.

Firstly, the object to be searched is selected and secondly, attributes are offered for search criteria depending on the choice. The architecture is affected, as new UIPs and UI-Controls are instantiated for the *DialogVisuals*. Additionally, the *PresentationController* actions and scope are altered.

In [24] and [25] run-time awareness of UIPs is mentioned, but not further outlined. As outlined in Section III.B, respective impacts of UIP *configuration* were included. Finally, we discovered two possible workarounds for composite UIPs, which govern the lifecycle of other sub-ordinate UIPs and thus demand for the "control aspect".

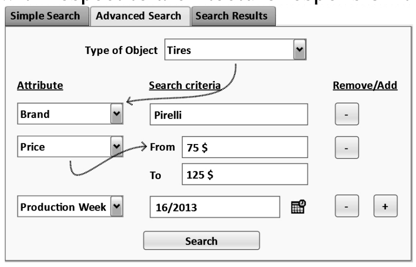
UIP context parameters. Firstly, the UIP specification language should permit parameters essential for an instantiation to varying contexts. This decoupling of UIPs from concrete GUI definitions has already been considered by the model-based approaches, which are assessed in Section IV. Without such parameters only invariant but most UIPs simply could not be formalized at design time [4][5].

Virtual user interface. Secondly, UIPs could be split into several atomic UIPs, which would compose a dialog on demand of the dynamic *control* aspect behavior. The atomic UIPs, being mostly invariant and mainly variable concerning data types and the number of structure elements, could be instantiated during run-time by a virtual user interface architecture [7]. This option would demand for manual realization or a DSL for *DialogController* and *View* creation.

E. *Influence Factor Model*

The influence factor model continuously has been supplemented during the previous sections and its final shape is depicted by Figure 3. The method applied is described in [4]. We cut-out factors not to be considered here.

UIP definition. The main *definition* factor is decomposed by the three aspects derived from Sections III.B and III.C. These are intended to identify, group and separate the impacts with respect to architecture responsibilities.



Type of Object: Tires

Attribute	Search criteria	
Brand		Pirelli
Price	From	75 $
	To	125 $
Production Week		16/2013

Figure 2. Exemplary advanced search [10] [4]dialog

View impacts. The impacts of the "view aspect" are concentrated on the *DialogVisuals* component. They are refined by two factors. "View definition" demands for the creation of stereotype visual structures composed of UI-Controls or even UIPs. As these impacts resemble static elements of a UIP definition, the second factor requests for parameters to be defined for them. In detail, they need to be named, enumerated and ordered, arranged in layout and customized by style in order to enable *variability* of single UIP instances.

Interaction impacts. The *interaction* impacts seem to be primarily focused on the *PresentationController*. In fact, the "Visual element structure states definition" impact depends on actual *View* structure composition and so, a point of view. Hence, the following distinction is made:

Firstly, when UI-Controls are the only components contained in the visual structure of an individual UIP, several states may have to be defined, which describe alternative *Views*. So, the first impact requires the definition of states a *PresentationController* has to ensure. For example, a UIP may formalize the choice of just two options out of many available, as it is sketched in Figure 4. Consequently, the possible states, e.g., activations, deactivations or toggling collapsible panels [10] of the *visual element structure* have to be specified by the UIP. Moreover, the defined *view structure elements* need to be bound to presentation related actions that trigger changes in states or data to be displayed. The "Presentation action-binding" foresees this binding. In detail, a certain UI-Control has to be configured to trigger a change in state of already defined visual elements of the same scope, e.g., deactivate a delivery address (when it is the same as billing address), assumed that the toggle button or checkbox belongs to the same UIP specification unit.

Secondly, superior UIPs of a composition need to specify an outside view on the sub-ordinate UIPs in order to change or instantiate new sub-UIPs dynamically. For instance, this is required when the user triggers the attribute combobox or buttons on the right hand side of Figure 2, which change states of criteria rows. Accordingly, when a UIP defines a composition of UIPs, then the lower situated UIPs constitute the *view structure elements*. Therefore, their outside view states have to be governed by the superior UIP. In this case, the *control* related impacts become relevant.

Control impacts. The impacts associated with *control* mainly apply to UIP compositions and affect both the *PresentationController* and *DialogController*. Several UIPs may define the *DialogVisuals* altogether. In that case the actions of the *PresentationController* are scattered among the individual UIP specifications as each one governs its own part of *View* separately. One UIP is to be defined as a supreme entity to control the other UIPs visual states or lifecycles. This way, a *hierarchical control flow* for presentation is to be established.

The UIP formalization has to enable the combination of various UIPs with the option to reuse their individual *view state* and *structure definition*. Thus, the *encapsulation of UIPs* demands for the autonomy of each UIP unit. As a consequence, UIPs need to define an interface to report their changes in state to superior UIPs. For this purpose, UIP *intercommunication events* need to be defined that allow for plugging in UIPs in a flexible way. However, UIPs still need to be isolated from each other in order to maintain a flexible composition and exchange options. According to events, they have to be distinguished as the architecture differentiates *PresentationController* and *DialogController*. Since the UIPs principally may be combined in any fashion to build composite UIPs, it is essential that one can define a differentiated perception for UIP originated events. On the one hand, one must specify, which UIPs events will trigger a change in sub-ordinate UIPs *view structure*. On the other hand, one has to define the UIP, which provokes application relevant events that are to be forwarded to the *DialogController*. For instance, a button of an online shopping dialog may trigger to copy billing address data to delivery address data fields of that dialog. Another button in a button bar may confirm the entire shopping process, so that data is validated and delivery address is checked. So, there is a need for "Dialog action-binding". The latter could also be associated to *interaction*, but we decided that this impact has a stronger relation to UIP compositions, when the superior UIP has to filter events from sub-ordinate UIPs and respectively forward them to the *DialogController*.

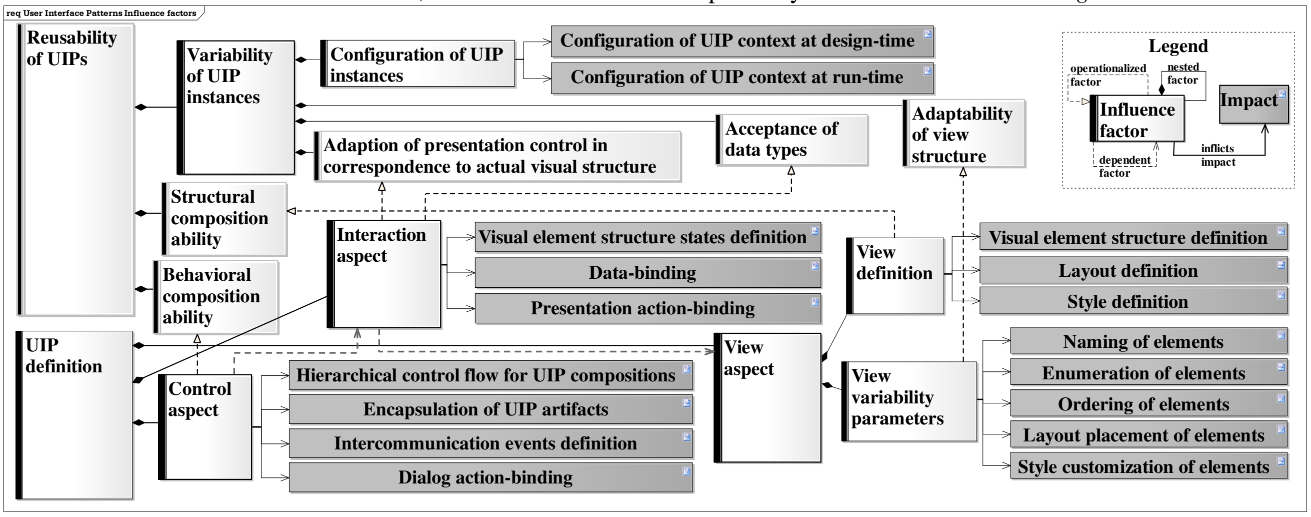

req User Interface Patterns Influence factors
Reusability of UIPs
Variability of UIP instances
Configuration of UIP instances
Configuration of UIP context at design-time
Configuration of UIP context at run-time
Adaption of presentation control in correspondence to actual visual structure
Acceptance of data types
Adaptability of view structure
Legend
operationalized factor
nested factor
Influence factor
Impact
dependent factor
inflicts impact
Structural composition ability
Behavioral composition ability
Interaction aspect
Visual element structure states definition
Data-binding
Presentation action-binding
View definition
Visual element structure definition
Layout definition
Style definition
UIP definition
View aspect
View variability parameters
Naming of elements
Enumeration of elements
Ordering of elements
Layout placement of elements
Style customization of elements
Control aspect
Hierarchical control flow for UIP compositions
Encapsulation of UIP artifacts
Intercommunication events definition
Dialog action-binding

Figure 3. Influence factors identified for the UIP analysis

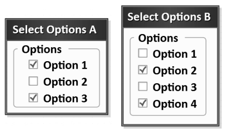


Figure 4. Checkboxes for the choice of two options

IV. Evaluation of Model-Based Development Processes

Recently, model-based development processes for GUIs employing UIPs or similar artifacts have been proposed. Based on available sources, we investigate what generative UIP concept they have incorporated. Afterwards, we review the capabilities and limitations of the respective concepts. More precisely, we consider what impacts of the factor model in Section III.E are supported or inspired.

A. *Annotated Task Models - Queen's University Kingston*

Harmonic evolution. To restrain the disharmonic evolution of business processes, application kernels and finally user interfaces, Zhao et al. [1] proposed the generation of GUI dialogs on the basis of task model specifications. They applied "usability practices and UI design principles to guide transformations" in order to ensure a better usability of generated solutions. So, task activities were annotated with information about roles, data in- and output. By parsing the augmented task model and applying rules, tasks were automatically segmented. For each segment, windows of a dialog model were derived, so that tasks handling the same data were kept together. This way the dialog structure and its transitions were created.

Task patterns. A fixed set of HCI patterns - called task patterns and based on collections like [10] - was mapped to specific task type segments on the basis of similar naming between both. During the transformation phases, a set of rules was applied for task- and dialog-modeling. For the presentation model, each occurring data type within the respective task was mapped to a certain UI-Control. Thus, a harmonic balance between grouped tasks, stereotype HCI-pattern assignments and windows with a reduced UI-Controls was propagated. On this basis, consistency between changed task models and GUI should be achieved by re-performing the transformation steps.

Factor support. Analyzing the relations to our factors, we found out that the only impacts to be mentioned were the following: For "Visual element structure definition", the UIPs were implicitly and strictly assigned by window or dialog rules according to the information provided in the mapped task-segment. Only a limited set of "task-patterns" was introduced so far. A free composition of UIPs was not possible or even aimed at. In addition, no fancy UI-Controls like separators, progress bars, sliders etc. were to be included for *view structure* definition. Thus, the *DialogVisuals* were statically dictated by a limited set of model dependencies.

Concerning "Layout definition", the general layout already was determined by the dialog model rules as there were Editor, Viewer and Dialog windows. Therefore, the meta-model for presentation was limited to very basic abstract UI-Controls and did not allow for custom UI-Controls arrangement like the separation of mandatory from optional data or a user specific grouping of data.

As far as *variability* and thus "Configuration of UIP context at design-time" are affected, the *DialogVisuals*, *PresentationController* and *Model* (data) were generated on the basis of GUI-generators. In sum, there hardly was any variability for the patterns aside from "data-binding" and the strict automated rules. In this regard, the definition of own presentation related patterns and usability principles was considered as future work.

Questions. The formalization of abstract UI-Controls or task patterns and their instantiation for certain contexts has not been outlined yet. How the mapping of task-patterns and tasks is done also remains as a question.

Summary. From our point of view, the approach of annotated task models combined with a mapping to task-patterns resembles a pure GUI-generator solution. However, this generator has much enhanced capabilities compared to single GUI-generators as it supports a much greater requirement basis: The task names drive the selection of a matching task pattern that is composed by certain usability rules. More important, the process does not need manual intervention and can be repeated when business processes have changed. Starting with the task model, the developer is able to initiate the update for both dialog and presentation model. Although, the solution promises great automation, it is not as flexible as the other UIP-based solutions. Its suitability for a wider range of task types, the customization of the uniform look & feel and the proposed future work of integration of own task patterns and UI design principles should be considered.

B. *Patterns in Modeling - University of Rostock*

PIC introduction. Forbrig et al. presented their development environment that employed UIPs in many consecutive sources. They described an approach also being based on task-models [11]. Dialog graphs were manually created with the DiaTask tool performing the steps of defining views, assigning tasks to those views and finally creating transitions between views. Then views were translated to windows for a WIMP paradigm [11] platform deriving the abstract user interface (AUI). For each interactive task defined, buttons were created as UI-Controls of a view within the AUI. The transition of the AUI to CUI (concrete user interface) was performed by manual refinement with the XUL-E tool. In this step, buttons from AUI could be replaced by other UI-Controls or even Pattern Instance Components (PICs). This way, the abstract windows with their buttons served as UIP-placeholders for manual replacement. In this regard, the tool-chain achieved to maintain the initial connection between UI-Controls and the task of the original task-model. Both AUI and CUI were formally described with an enhanced XUL format as the final output. Hence, PICs drafted an early approach to formalized HCI patterns, as they only supported a set of five patterns [12]. Moreover, they allowed for the simultaneous replacement of more than one button and had task control data for specific tasks attached, which enabled a more customized processing of the respective task or domain data.

PIM. Continuing the work on pattern integration into model-based processes, a new modeling framework was presented in [13] along with the PIM (Patterns In Modeling) tool. The pattern application areas were expressed as "UI Multi Model" (task, dialog, presentation and layout). The PIM was intended to apply and manage patterns for the four defined models. Mainly, the focus was laid on the task model and an approach for pattern integration therein. Other models were not yet supported, but the steps for pattern instantiation using a wizard were drafted. Firstly, instance structures had to be specified, describing how often model fragments are duplicated. Secondly, the variables defined by model fragments had to be assigned. To express model fragments, a research into formalization options for the model patterns - especially presentation and layout - was conducted.

PIC revised. After two years, an overview of earlier work was provided in [16]. The focus once more lay on task-patterns to derive dialog navigation structures. It was outlined that the pattern instantiation process could not be fully automated. Therefore a "combination of automatisms and human designer" was propagated. So, a semi-automatic approach to creation of dialog graphs on the basis of task models and respective pattern application was introduced. Thus, the AUI was generated from dialog graphs containing views as placeholders for UIP integration. For comparison, the approach by Zhao et al. [1] fully relied on automated derivation of dialog structures. As before, the CUI was manually refined by replacement of UI-Controls. XUL-E as a tool would permit the refinement of view structures within generated navigation dialogs and their correspondence to the tasks. Manual customization and instantiation of UIPs was suggested by relying on PICs as formal HCI pattern representations, which already resembled context-specific instances of the respective patterns [17].

Factor support. Although it was proven that the instantiation of invariant UIPs was possible, this step was restricted to the replacement pre-determined UI-Controls. Concerning **"**Visual element structure definition", the presentation model included UIPs implicitly as they were intended to be changed and adapted manually. For instance, the content area of each of the wizard dialog windows [11] had to be customized once more via the replacement mechanism. The generator created an initial abstract design like the buttons in the mail client example. Thus, the wizard pattern was strictly defined and could be adapted only in limited ranges to the context as the textfields could be replaced by other PICs or UI-Controls. Additional manual adjustments were necessary, e.g., to remove the next button in the last dialog "Apply". Thus, *variability* depended on manual rework. Lastly, not all kinds of UIPs were supported.

In sum, the "Configuration of UIP context at design-time" relied on the PICs "pre-arranged as components." [11]. From the wizard-example, we assume that there existed an explicit dependency to the task information serving implicitly as UIP-instance parameters. Thus, one could freely decide on what parameters to be used for particular UIP-instances, as this was the case for the approach by Zhao et al.

The "Layout definition" was determined by the PICs, which probably consisted of a strict layout (content area of a dialog, wizard button bar) and always instantiated the same button configuration (Prev, Next, Finish).

For "View variability parameters" the PIM-Tool approach [13] suggested an instantiation process, which could have inspired our parameter impacts: The *view structure definition* and variable assignments were introduced. Also, a hierarchical refinement of an entity by structured patterns concentrated on one of the four models or a mixed selection of them could be learned from this approach. Therefore, the following requirements were also inspired by Forbrig et al.:

The "Hierarchical control flow for UIP compositions" and the "Dialog action-binding" had been drafted. Based on published work, we support the assumption, that UIPs must not interfere with application states since those are to be determined by tasks. According to the "Intercommunication events definition" and after following the vision established by Forbrig et al., one could come to the conclusion that standard-events were quite relevant to plug-In UIPs for altering tasks or to allow UIPs for various task-combinations. After all, the idea of replacement is also important since UIPs should be exchangeable in dialog placeholders in order to enable a change in *view structure* but not in application workflow. Therefore, UIPs need to be replaceable and universal in shape and the impact "Encapsulation of UIP artifacts" may be inspired as well. We vote that the ability to build a cascade of UIPs is important because artifact details or their modules and matching project requirements are hardly the same in different projects. Hence, specific interpretations and instances are of the essence.

Questions. The main emphasis was put on task modeling and the application of patterns on that context. How PICs would be instantiated and applied to contexts is not clearly outlined. It is also questionable how a PIC was successfully shaped to be abstract and universally deployable.

Summary. The approach with rich tool support investigated on the feasibility of "patterns in modeling" [13] and backed or could have inspired some of our factors impacts. The PIM-Tool voted for a combination of model-based and pattern-based approach. This implicated and required a UIP base model to increase reuse and lessen efforts for linking and model integration.

Both Zhao and Forbrig et al. followed a similar approach as they progressed towards the combination of model- and pattern-based development to ensure cost-effectiveness and the application of patterns for the sake of good usability. Forbrig et al. put more emphasis on the pattern aspect. In this respect, they developed tools and customized formal languages for the individual models. However, besides tool support automation could not be increased to the desired level and manual refinements in interaction with the tools had to be performed. For instance, task models had to be shaped to accommodate PICs after the derivation of dialog graphs. Finally, specific variants of arbitrary UIPs could be modeled with the tools and thus greater variability could be achieved compared to Zhao et al.

C. UsiPXML - University of Rostock

Following the former PIM approach, another pattern application framework for UIPs was presented in [14]. The models were further elaborated here, as layout and presentation were intended to refine the AUI and thus enable the transition to a CUI by instantiation of common solutions encapsulated by respective patterns. To organize the patterns of all four models, the "User Interface Modeling Pattern Language" was introduced as a pattern language. The CUI, where UIP instances were to be integrated next, still needed manual adaptation work.

Continuing towards formalization of UIPs, they presented UsiPXML (User Interface Pattern Extensible Markup Language) as an enhanced UsiXML (USer Interface eXtensible Markup Language) pattern specification language for all four models. To provide both context information for proper usage of a pattern and "implementational information" [15] for automated processing, UsiPXML incorporates PLML (Pattern Language Markup Language) for description and UsiXML as generative part. The PIC concept of older sources is not mentioned here. In contrast, the UsiXML enhancements are further elaborated in [15]. The new pattern notation followed the PIM pattern instantiation steps and thus featured structure attributes, which would determine how many times (min, max) an element within the pattern is instantiated. In addition, variables were incorporated to define mandatory placeholders for values, which could be governed via assignments and applied for various purposes. The former defined how variables would be evaluated. Pattern references, a third feature, would specify sub-pattern-relationships for refinement.

However, UsiPXML is no longer mentioned in subsequent sources again focusing on PICs. Finally, the goals to be achieved with UsiPXML were relativized in [17] as they stated PIC "is called instance component, since we consider the template to be already an instance of the pattern that is described through this component. We are aware of the fact that, due to their nature, not all known HCI patterns can be treated as or translated into an algorithm or a PIC."

Factor support. For the "Visual element structure definition" presentation patterns were applied to define *view structures*. Concerning UIP compositions, the patterns were always presented in isolation and never in entirety, so the real capabilities cannot be judged.

Separate patterns were dedicated to the "Layout definition" impact. As it was not clearly outlined, where they could be included in the hierarchy of pattern instances, the flexibility of the solution cannot be assessed as well.

As far as "View variability parameters" are concerned, structure attributes as well as variables and assignments were invented. Those parameters would permit the deactivation of certain pattern structure parts [15] by "set" assignments.

The "Data-binding" was also realized by the "set" assignment, so that an implicit mapping of data types to abstract UI-Controls was possible. This way the developer did not have to decide for each domain object attribute what kind of UI-Control or UIP to instantiate in a form.

A hierarchical structure of patterns was employed, so patterns could be combined via a pattern interface. More precisely, the variables of higher order patterns could be passed to the pattern interface of lower patterns in order to allocate their variable definition. For vast flexibility in *pattern composition ability*, such a pattern interface could have been arranged for potential reusable patterns, but this is not further mentioned.

The pattern interface, variables and assignment facilities might have been useful to empower "Hierarchical control flow for UIP compositions" and the "Encapsulation of UIP artifacts", but due to missing examples and language specifications, these cannot fully be judged. However, the variables were not standardized for certain pattern types, so they depend on the individual pattern model fragment and their evaluation by the assignments. So, a superior UIP needs to know about implementation details of sub-ordinate UIPs. That is why the *encapsulation* eventually might be broken.

Inspired by the realization of the "Unambiguous Format" [15] pattern, the advanced search criteria rows of Figure 2 could be defined in an abstract manner by UsiPXML, but it has to be answered how they could be requested during run-time. Eventually, the realization of "Configuration of UIP context at run-time" remains unsolved.

Questions. At first we ask, how presentation and layout patterns are merged in a generated window. Both are "CUI Model Fragments" [15] and in that source the patterns are only shown separately but not integrated. As far as UsiXML is reused here, UsiPXML should have inherited some of its weaknesses [3][4]. For instance, how could UI-Control types be platform independently described when UsiXML uses a strict set of types for UI-Controls? How did UsiPXML allow for the description of all four 4 models when UsiXML cannot describe presentation and especially layout models separately? For a better assessment of these issues, we miss code examples of UsiPXML.

Summary. This solution may be a great enhancement concerning the expression ability of generative UIPs. Yet, it is overshadowed by many open issues concerning impact details, which have not been presented yet. So, this approach could not accurately be assessed by us. Moreover, this approach is limited to UIPs being able to be specified at design time. A UIP dynamically morphing during run-time as in Figure 2 most likely cannot be defined with known UsiPXML facilities. Lastly, the occurrence of sub-patterns was the only considered relationship so far. "Inter-Model" [24] patterns have not been considered yet.

D. PaMGIS - University of Augsburg

HCI Pattern language. Engel et al. [20][21][22] state that current UIP-collections do not reflect the need to structure the UIPs to certain aspects, which would enable to select and judge them independently from domain or their relationships. They express that the abstraction and organization criteria are not satisfying. Starting with advice, how to structure a UIP language properly [18], Märtin et al. gradually advanced to their own concepts for UIP instantiation. The rules of a global entry point, allowed and not allowed links within the pattern hierarchy, should guide the user of the pattern collection, so that he would have to start with a rather "abstract pattern for the general problem class" [18] and consequently follow the same abstractions searching the pattern hierarchy for a solution. It should be avoided to oversee a potential useful pattern and isolate individual patterns. The concept was applied in later sources.

PaMGIS. An entirely new modeling architecture was presented in [19], named "pattern-based modeling and generation of interactive systems (PaMGIS)" and neglecting the very recent work of PIM, UsiPXML and respective four pattern types of the University of Rostock. A central pattern repository would hold patterns for the following modeling stages. Firstly, an "abstract application model is generated" (AAM) by interpreting a set of potential input models (task, user, device, context). Secondly, "a semi-abstract application model" (SAAM) is generated. During this step, the patterns might be instantiated. For this purpose, patterns were composed of both descriptive and generative information. In detail, the generative part introduced an <automation> XML tag allowing the parameterization of the respective <element>, which served as the container and layout unit of the UIP. The <children> tag referenced child UIPs or UI-Controls, governed their number, ordering and position in relation to the parent UIP. This mechanism was based on the <element> tag and the therein defined attributes of the respective sub-UIP or UI-Control. The superior UIP could select from the lower specified attributes.

The approach of PaMGIS was further outlined in [22] by Engel. He stated that the process was based on the enhancement of information derived from fully-fledged task-models and unique pattern models. Patterns would be applied for both the extended AAM and the SAAM. Furthermore, he mentioned that the framework contained a repository for UI-Controls as lowest units in the UIP hierarchy, which would be mapped to target platforms.

Joined by Forbrig, Engel and Märtin presented further information about the PaMGIS framework and the DTD applied for the generative <automation> tag of UIPs in [23]. By example, they outlined the unique way of structuring UIPs based on [18]. Therein, main categories resembled technically shaped patterns appropriate for the current GUI structure element, e.g., a panel or button-bar as sub-patterns.

Factor support. The XML specification defined by the <automation> DTD is closely related to "Visual element structure definition". In general, UIPs are supported as composites. They always define the inclusion of child patterns, since even UI-Controls are regarded as patterns. Their ordering is explicitly determined and constraints are allowed as well as optionals. However, the composite patterns only approach is unfavorable, since one cannot decide on what are composite and what are atomic units of reuse. For instance, a panel is often to be used as an atomic unit in Figure 2. The advanced search UIP defines its own tree of elements or reuses entire UIPs. Not the included panel should decide on that. Anyway, one cannot use a panel without children definition in PaMGIS, since this would result in an empty panel as well as a breach in layout definition hierarchy. The UIP hierarchy is designed in a way, that UIP definitions cannot traverse more than one level at once. So the structure parameters would be limited to a certain levels scope. Single UIPs were too strictly bound to the hierarchy, as they always would have to determine about sub-ordinate UIPs. The leveling would be too strict and one-dimensional, so that one can only include a certain UIP with its respective children and not without them. For Figure 2 this would implicate, a specialized set of panels had to be formalized. Many specialized versions of a panel would have to be created, because the children hardly would be reusable in other contexts. In sum, the visual options are detailed, but high efforts for formalization are needed, as there would be a high amount of UIPs and branches in hierarchy.

As there is no dedicated layout pattern, "Visual element structure definition" and "Layout definition" are merged in UIP definitions. The Layout is governed by the superior UIP, which refers to parameters provided by the children and provides values for them. Therefore, layout attributes are explicitly maintained by children. This may be a drawback compared to layout patterns used in UsiPXML, hence for changes in layout each single UIP instance has to be touched.

Concerning "View variability parameters", there are no dedicated parameters for the view structure, as each pattern instance has to be declared explicitly to be included. For layout, naming and ordering, the respective attributes have to be assigned with certain values.

Questions. Consequently, each pattern, that reuses others, needs to define them as children. As Seissler et al. [24] have found out, the UIP hierarchy may be inflexible or does not permit all possible combinations of UIPs to form new UIP compositions. So the UIPs may indeed be very statically linked among each other. For instance, the panel in [19] can only be instantiated with the two buttons, since this pattern has declared them as children. It is questionable whether for each pattern instance the <automation> has to be defined over and over, or if one is assisted by a tool. Since the pattern instance configuration was not described, it is not clear, how the occurrences of children (min and max) are configured and how this impacts their order in layout.

In addition, the concepts for *data and action binding* have not been presented yet. Moreover, the intended realization of *control aspect* impacts is not clear. This is of the essence for the fine grained pattern structure and so, each UIP instance is composite.

Summary. Due to above issues, the *variability* of this approach can hardly be assessed. Along with missing concepts for the *control aspect*, the generic and fine-grained UIP categorization approach is arguable and has to be proved. Both framework and process of PaMGIS were only drafted by available sources. Therefore, the scope for AAM and SAAM model generation stages were not outlined as the application of patterns was only mentioned for the SAAM.

E. Encapsulated UIML - University of Kaiserslautern

Reflection of recent approaches. Seissler et al. shortly reflect previous approaches and present their rather new pattern application framework in [24]. Concerning PaMGIS, they claim, separation of concerns was compromised, since layout information was implicitly included in the generative part of <automation> (anchor attribute) and this way, layout and presentation structure were mixed up. In addition, the pattern language suggested was "very fine-grained (and complex)" and thus contradicted the idea that patterns would cover a broader view on the problem. Regarding UsiPXML, it is described as "one of the more mature approaches", but also has a weak spot, since links between individual patterns were rated as rather static. Finally, it was implied that UIP compositions could not be built flexibly.

Process. Within their process, they suggest the "Use Model" for tasks, "Dialog Model" for the states of view and finally a "Presentation Model" to express certain interaction objects and their layout. For each model patterns could be defined. The patterns were classified according to their relationships on the model layer and to each other. Single patterns do stand alone; "intra-model" patterns reference sub-patterns of the same model and "inter-model" patterns reference patterns of other models and may include both other kinds. In contrast to UsiPXML, separate notations were being used for every model. The presentation used UIML and both held structure (UI-Controls) and layout. Rather than deriving dialog graphs from tasks, they defined infinite state machines for dialogs to be interpreted at run-time.

Pattern instantiation. A "generative pattern solution" consisted of the three parts "Pattern Specification Interface" (PSI), "Pattern Interface Implementation" (PII) and "Model Fragments" [24]. The PSI offered instance parameters of two types, as there were variables and constraints (data type, min, max and default) to be defined for each model fragment. A selection acted as a special variable to enable a choice out of more than one data option. Furthermore, model fragments constituted the core solution (e.g., UIML for presentation) and thereby a non-altered notation. The enhancements were limited to the PSI and PII. The latter is realized via XSLT and allowed the specification of four basic operations. It put the parameters to effect on the core part: The structure of a model fragment might be altered by add, remove and replace operations. The assign operation passed parameters to the corresponding model fragment attributes in order to assign data to defined variables. After selection and instantiation, patterns were integrated to be finally interpreted.

For future work the tool-chain has to be developed, the pattern notation is to be tested according to its formalization capabilities and lastly, a refinement of inter-pattern relationships is to be sought after.

In a more recent source, Breiner et al. [25] once more introduce their model framework, but add the conclusion that HCI patterns are difficult to integrate in model-based processes, since they missed a "lingua franca or modeling standard". They outline the process of pattern formalization and add that a pattern commonly features both fixed and adaptable content. In the future, the automation of pattern instantiation and integration shall be investigated. Another aspect, aimed at in future, focuses on how to determine and consider user capabilities during GUI creation at run-time.

Review of criticism. To begin with, we consider their way of argumentation for criticism on other approaches. In principle, Seissler et al. do not provide information on requirements allowing for a comparison with the other approaches. According to their valuation, UsiXML has least weaknesses. We wonder, what a direct comparison between their and the UsiPXML approach would result in.

According to PaMGIS, they regard the mix-up of layout and presentation patterns as unfavorable. A separation might be irrelevant, since layout patterns in PaMGIS would always serve as a container in the final hierarchy. The UsiPXML separation may eventually be mixed up in the same model as it seems (both are rooted as "CUI Model Fragment"). It is arguable, whether layout patterns are an aspect and thus can be applied almost anywhere at a certain stage in PaMGIS pattern language. It might be no help keeping layout separately in this kind of pattern hierarchy. In this respect, the fine grained structuring of patterns for PaMGIS has been criticized, too. There might be too many levels of decomposition, but Märtin and Roski suggested starting to search in the highest hierarchy in order to preserve all options. However, from the statements by Seissler et al. about UsiXML keeping core models encapsulated an indirect critic about PaMGIS can be uttered: PaMGIS merges model information to create AAM and subsequently the SAAM. Thereby, it was not mentioned if and how backwards links, as Forbrig et al. have propagated, are established.

Factor support. Seissler et al. have drafted their thoughts on "View variability parameters". They follow the idea to incorporate parameters on a very general level, as those are not categorized as structure, layout related information. Instead, they define parameters individually and ad-hoc for each model fragment. This way, parameters are clearly bound to the core pattern contents and are dependent on tool algorithms, like this is the case for UsiPXML and PaMGIS. Using the four operations supported by the PII, versatile modifications on the model fragment similar to the capabilities of UIML 4.0 template handling can be achieved. In addition, they augment the UIML features with parameters, since they state "PICs might be interpreted as attributed templates that can be instantiated" [24]. Therefore, they may have realized all impacts of the "view aspect".

As far as "Configuration of UIP instances" is concerned, this may be realized for design time only. They are aware of the need to configure UIPs at run-time [25] and thus support the respective impact. Nevertheless, they did not present a concept, how parameters could be changed at run-time.

Questions. Since Seissler et al. propose the PSI, the "Encapsulation of UIP artifacts" seem to be realized. As the parameters were also not standardized in analogy to UsiXML, this impact finally might not be met. A superior UIP requires information about the variables roles in the actual "Model Fragment" and their handling by the PII. In addition, it was not presented how UIPs may be composed.

Summary. This approach is very promising, but not easy to valuate, since no full UIP has been presented as working example. In addition, they fail to argue deeply for thoughts on the instantiation mechanism and pattern notation. Facing UsiPXML, their approach seems not to be backed entirely by their criticism. Despite this, their pattern categories may trade off, since they are more oriented towards pattern inter-relationships. However, it is arguable if categories by Märtin et al. will work in complex examples as well or will prove to be too atomic for a high usability in pattern composition. Seissler et al. strive for many goals such as dialog transitions, presentation model UIML fragments to be interpreted at run-time and maybe re-configurable at run-time. Up to now, a comprehensive proof of concept has not been given.

V. Conclusion and Future Work

Results. We presented an overview of recent approaches to generative UIP deployment within model-based development. Different researchers proposed their own model frameworks and UIP formalization techniques. Our analysis revealed that they either could not cover every factor (especially the UIP configuration at run-time) or have significant issues to be solved. A reason for that may be found in missing criteria to guide the applied concepts and UIP representations. In our opinion, a sufficient notation for UIPs has yet to be developed or refined based on available approaches. Until now, there have been no efforts for standardization concerning a unified UIP specification. In contrast, UIML and UsiXML both have emerged as strong options for GUI specification. Whether they can serve as a basis to develop a language dedicated to the specification of generative UIPs, remains an open research question.

Achievements. We refined our earlier work [3][4][5] and elaborated a detailed requirements model for the analysis of UIP formalization and instantiation aspects. As we found strong support or inspiration by the other approaches, the established factor model can be used for their verification.

Limitations. We did not consider devices, environments [21] or user skills [25] for UIPs. The categorization of UIPs [18], their descriptive relationships [20], their mapping to tasks [16], as well as their instantiation for paradigms different than WIMP also were not covered.

Future work. We strive to communicate the requirements for UIPs more deeply and in more detail. Other researchers involved in UIP related topics may reassess their aims and capabilities on the basis of the presented factors. They are sincerely invited to suggest improvements. Our analysis solely is based on the sources included in references. A deeper comparison of the approaches could be initiated by contacting the respective authors to honestly ask for current tools or UIP notations to be evaluated in a practical study.

REFERENCES

[1] X. Zhao, Y. Zou, J. Hawkins, and B. Madapusi, “A Business-Process-Driven Approach for Generating E-commerce User Interfaces,” Proc. 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07), 2007, Springer LNCS 4735, pp. 256-270.

[2] C. Hofmeister, R. Nord, and D. Soni, Applied Software Architecture. Boston, Addison-Wesley, 2000.

[3] D. Ammon, S. Wendler, T. Kikova, and I. Philippow, “Specification of Formalized Software Patterns for the Development of User Interfaces,” Proc. 7th International Conference on Software Engineering Advances (ICSEA 12), Nov. 2012, Xpert Publishing Services, pp. 296-303.

[4] S. Wendler, I. Philippow, “Requirements for a Definition of generative User Interface Patterns,” Proc. 15th International Conference on Human-Computer Interaction (HCII 13), July 2013, in press.

[5] S. Wendler, D. Ammon, T. Kikova, and I. Philippow, “Development of Graphical User Interfaces based on User Interface Patterns,” Proc. 4th International Conferences on Pervasive Patterns and Applications (PATTERNS 12), July 2012, Xpert Publishing Services, pp. 57-66.

[6] J. Vanderdonckt and F. M. Simarro, “Generative pattern-based Design of User Interfaces,” Proc. 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems (PEICS 10), June 2010, ACM, pp. 12-19.

[7] E. Denert and J. Siedersleben, “Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur,“ Informatik Spektrum, 23(4), Aug. 2000, Springer, pp. 247-257.

[8] M. Haft, B. Olleck, “Komponentenbasierte Client-Architektur,” Informatik Spektrum, 30(3), June 2007, Springer, pp. 143-158.

[9] J. Siedersleben, Moderne Softwarearchitektur - Umsichtig planen, robust bauen mit Quasar. 1st ed. 2004, corrected reprint, Heidelberg, dpunkt, 2006.

[10] M. van Welie, “A pattern library for interaction design,” http://www.welie.com 20.01.2013.

[11] A. Wolff, P. Forbrig, A. Dittmar, and D. Reichart, “Tool Support for an Evolutionary Design Process using Patterns,” Proc. Workshop: Multi-channel Adaptive Context-sensitive Systems (MAC 06), May 2006, pp. 71-80.

[12] R. Rathsack, A. Wolf, and P. Forbrig, “Using HCI-Patterns with Model-based Generation of Advanced User-Interfaces,” Proc. MoDELS'06 Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 06), Oct. 2006, CEUR Workshop Proc. Vol-214.

[13] F. Radeke, P. Forbrig, A. Seffah, and D. Sinnig, “PIM Tool: Support for Pattern-driven and Model-based UI development,” Proc. 5th International Workshop on Task Models and Diagrams for Users Interface Design (TAMODIA 06), Oct. 2006, Springer LNCS 4385, pp. 82-96.

[14] F. Radeke and P. Forbrig, “Patterns in Task-based Modeling of User Interfaces,” Proc. 6th International Workshop on Task Models and Diagrams for Users Interface Design (TAMODIA 07), Nov. 2007, Springer LNCS 4849, pp. 184-197.

[15] F. Radeke, “Pattern-driven Model-based User-Interface Development”, Diploma Thesis in Department of Computer Science, University of Rostock.

[16] A. Wolff and P. Forbrig, “Deriving User Interfaces from Task Models,” Proc. Workshop: Model Driven Development of Advanced User Interfaces (MDDAUI 09), Feb. 2009, CEUR Workshop Proc. Vol-439.

[17] A. Wolff and P. Forbrig, “Pattern Catalogs using the Pattern Language Meta Language,” Electronic Communication of the European Association of Software Science and Technology, vol. 25, 2010.

[18] C. Märtin and A. Roski, “Structurally Supported Design of HCI Pattern Languages,” Proc. 12th International Conference on Human-Computer Interaction (HCII 07), July 2007, Springer LNCS 4550, pp. 1159-1167.

[19] J. Engel and C. Märtin, “PaMGIS: A Framework for Pattern-Based Modeling and Generation of Interactive Systems,” Proc. 13th International Conference on Human-Computer Interaction (HCII 09), July 2009, Springer LNCS 5610, pp. 826-835.

[20] J. Engel, C. Herdin, and C. Märtin, “Exploiting HCI Pattern Collections for User Interface Generation,” Proc. 4th International Conferences on Pervasive Patterns and Applications (PATTERNS 12), July 2012, Xpert Publishing Services, pp. 36-44.

[21] J. Engel, C. Märtin, and P. Forbrig, “HCI Patterns as a Means to Transform Interactive User Interfaces to Diverse Contexts of Use,” Proc. 14th International Conference on Human-Computer Interaction (HCII 11), July 2011, Springer LNCS 6761, pp. 204-213.

[22] J. Engel, “A Model- and Pattern-based Approach for Development of User Interfaces of Interactive Systems”, EICS 2010 Proc. ACM SIGCHI symposium, June 2010, ACM, pp. 337-340.

[23] J. Engel, C. Märtin, and P. Forbrig, “Tool-support for Pattern-based Generation of User Interfaces,” Proc. 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems (PEICS 10), June 2010, ACM, pp. 24-27.

[24] M. Seissler, K. Breiner, and G. Meixner, “Towards Pattern-Driven Engineering of Run-Time Adaptive User Interfaces for Smart Production Environments,” Proc. 14th International Conference on Human-Computer Interaction (HCII 11), July 2011, Springer LNCS 6761, pp. 299-308.

[25] K. Breiner, G. Meixner, D. Rombach, M. Seissler, and D. Zühlke, “Efficient Generation of Ambient Intelligent User Interfaces,” Proc. 15th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (KES 11), Sept. 2011, Springer LNCS 6884, pp. 136-145.

Three Patterns for Autonomous Robot Control Architecting

Carlos Hernández, Julita Bermejo-Alonso, Ignacio López and Ricardo Sanz
Autonomous Systems Laboratory
Universidad Politécnica de Madrid, Spain
Emails: carlos.hernandez@upm.es, jbermejo@etsii.upm.es, ignacio.lopez@upm.es, Ricardo.Sanz@upm.es

***Abstract*—Construction of robotic controllers has been usually done by the instantiation of specific architectural designs. The ASys design strategy described in this work addresses the synthesis of custom robot architectures by means of a requirements-driven application of universal design patterns. In this paper we present three of these patterns —the Epistemic Control Loop, the MetaControl and the Deep Model Reflection patterns— that constitute the core of a pattern language for a new class of adaptive and robust control architectures for autonomous robots. A reference architecture for self-aware autonomous systems is synthesized from these patterns and demonstrated in the control of an autonomous mobile robot. The term "autonomous" gains a deeper significance in this context of reflective, pattern based controllers.**

Keywords*— *Patterns; autonomous systems; robot controllers; reconfiguration; model-based systems; meta-control.

I. Introduction

Control architectures for autonomous mobile robots have a long and heterogeneous history [1]–[3]. In some of these cases, robot controllers have been built from scratch as ad-hoc solutions without any underlying systematic software architecture. The architectural foundation is implicit and the effort is focused on specific, concrete, needed functionalities and component technologies to provide them. These elementary functionalities are then deployed, integrated and operated over a minimal integration platform to generate the robot control system [4].

An architecture-centric approach is strongly needed in robotics. Focusing on architecture means focusing on the structural properties of systems that constitute the more pervasive and stable properties of them [5]. Controller architecture —the core set of organizational aspects— most critically determines the capabilities of robots, resilience in particular. Robot mission-level resilience is to be attained by maximizing architectural adaptivity from a functional perspective [6]. In this vein, the *Autonomous Systems* Programme (ASys) tries to leverage a model-based [7], architecture-centric, process for autonomous controller construction.

This paper describes some developments in this direction in the form of *reusable design patterns*. The paper is organized as follows: Section II describes the use of design patterns as an architectural strategy; section III describes the three target design patterns; section IV describes the reference architecture generated using these patterns; sections V and VI contain a roadmap for future work and conclusions.

A. The ASys Research Programme

The ASys Programme [8] is a long term research effort of very simple purpose: develop domain-neutral technology for building custom autonomy in any kind of technical system. In this context, *autonomy* has a broader meaning that the regular use of the term in autonomous mobile robots [9]. ASys pursues the identification of core architectural traits that enable a system to handle any kind of uncertainty, whether environmental or internal. It is not just a quest for achieving robust movement planning technologies in uncertain environments but *robust teleonomy for unreliable systems in uncertain environments*. Adaptation is a key issue in autonomous robotics [10] to enable the coping with environmental changes and with internal faults. Architectures enabling dynamic fault-tolerance is an important aspect of the work presented in this paper.

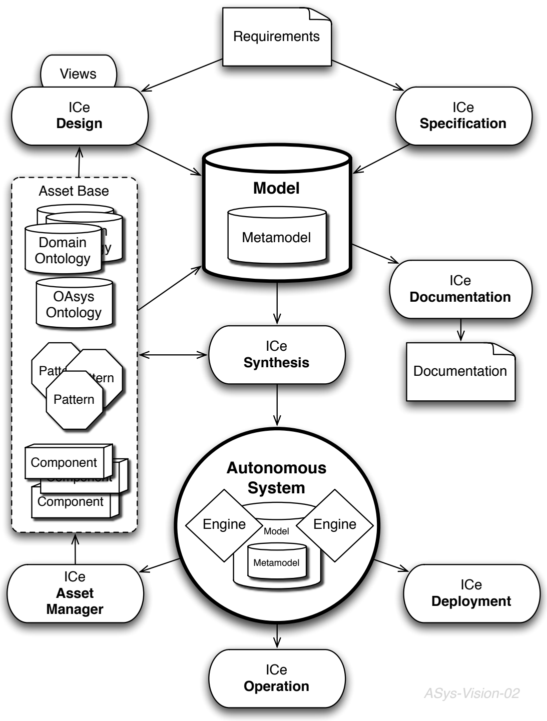


Figure 1. The ASys model-centric systems engineering process. ICe stands for *Integrated Control Environment*.

The mainstream direction of the ASys Programme comes from a simple observation: there exists a class of competence that may maximize system autonomy: cognition. We can observe it when technical systems do overcome the unexpected beyond what was technically planned and built into them at design time. It is the idiosyncratic competence of McGyvers or what is shown in the *Apollo XIII* movie. Systems can be more adaptive by making them exploit design knowledge at run-time. The ASys strategy is simple: *build any-level autonomy systems by using cognitive control loops to make systems that can engineer themselves*. A controller of maximal robustness will be able to redesign itself while fielded.

We try to generalize and downsize engineer's capabilities to the level of atomic, resilient subsystems in all kinds of operational conditions in technical systems [11]. Machines that *deeply know* themselves —their structure, their functions, their missions— will be the mission-level robust machines we need for the future, according to our vision of *self-aware machines* [12].

B. ASys Design Principles

This research into general artificial autonomy is driven by some *fundamental design principles* that structure the research and development of our technologies [13]. Three of them are of special importance to the work described in this paper:

- **Model-based control:** Cognition is the core competence to develop into robots; a cognitive control loop is based on the exploitation of explicit models of the system under control [14].
- **Metacontrol:** Teleological robustness —the stubborn prosecution of mission goals— is achieved by means of control loops handling disturbances. When these can happen in the controller itself we need metacontrollers deal with them [15].
- **Break the run-time divide:** There are *design* models that engineers use to build a technical artifact and *run-time* models that reflective systems may use during their operation. Using the same models for both will break the design/run-time divide and leverage the full potential of model-driven development [16] at run-time.

These principles are further developed in section III, where we explain how we have reified them in the three patterns that are the core content of this paper. The final objective of this work is the provision of generalized adaptation mechanisms by means of run-time reflection in advanced, real-time cognitive architectures.

II. A Pattern-based Strategy

The ASys strategy for building autonomous control systems is the exploitation of reusable assets over architectures defined by means of *design patterns* [17] (see Figure 1). This paper describes the construction and use of three patterns — assets in the *Asset Base*— and their use in the synthesis of an autonomous robot.

A. A Pattern-based Design Approach

A *design pattern* [18] is a reusable solution to a recurring problem. Design patterns are usually not complete designs for whole systems but descriptions of partial designs that offer a solution template of problem solving strategies that may be instantiated for concrete problems. In principle, patterns capture best practices and anti-patterns capture worst practices: things to do versus things to avoid when designing or implementing specific applications [19].

The final objective of this work is the creation of a generative pattern language to support the construction of intelligent integrated controllers for autonomous systems.

B. A Pattern Schema

Below we briefly describe the different sections of the *pattern schema* [20] that has been used in this paper:

- *Name*: The name of the pattern.
- *Aliases*: Patterns are usually not new; most of them have been discovered and used elsewhere, esp. in the controls domain.
- *Example*: A use case of the pattern; a possible application of the pattern in a real situation.
- *Context*: Contextual information regarding the potential application of the pattern.
- *Problem*: The problem that the pattern tries to solve.
- *Solution*: The form of the solution that the pattern provides.
- *Structure*: An architectural description of the pattern using roles and relations between roles.
- *Dynamics*: How system activity happens as sequences of role activations.
- *Related patterns*: Other patterns related with this, by structure, by way of use or because they are applied at the same time to a system.
- *References*: Bibliographic references for the pattern.

III. Three patterns

The focus of this paper are *three design patterns* that reify some of the ASys principles for the design of autonomous systems (see Table I). These patterns have been integrated in the OM Reference Architecture for the development of robust controllers for autonomous robots (see Section IV). Two of the sections are almost identical for the three patterns; they share a common *context* and are *closely related*:

`Context` Development of robust control architectures for autonomous systems; in the current drive toward increased complexity and interconnection and with a need of augmented dependability. The design strategy of these systems has to address not only the problem of the uncertainty of the environment, but also of the uncertainty arising from

TABLE I. THREE ASYS DESIGN PATTERNS

Acronym	Name	Content
ECL	*Epistemic Control Loop*	To exploit world knowledge in the performance of situated action.
MC	*MetaControl*	A controller that has another controller as control domain.
DMR	*Deep Model Reflection*	To use the system engineering model as self-representation.

the system itself because of faults or unforeseen, emergent behaviors resulting from the interplay of their components, or their connection with other systems [21]. More traditional approaches such as fault-tolerant control based on redundancy are too expensive and not efficient. The universality of the problem demands a general approach rather than specific solutions for certain applications that are difficult to transfer to other domains.

Related patterns These three patterns can be said to constitute a micro *Pattern Language*, sharing a common context of application and being conceived so as to apply them jointly for the development of control architectures for autonomous robots.

The next three sections describe the three patterns using the schema presented in section II-B.

A. *The Epistemic Control Loop Pattern*

Name Epistemic Control Loop (ECL).

Aliases RCS node, PEIS loop, OODA loop.

Example The navigation control system of an autonomous mobile robot.

Problem Sometimes controllers are required to implement a closed-loop strategy using an explicit model of the plant —the controlled system, e.g. the mobile robot—, with the possibility to also incorporate feed-forward action or predictive control, by providing design scalability to seamlessly incorporate different algorithms in the same control process.

Solution The Epistemic Control Loop pattern defines a loop that exploits world knowledge —i.e. a model of the plant— in the performance of situated action (see Figure 2). This loop is a variant of *Feedback* loop pattern in classical control [17], but in which the sensory input is used to update an explicit representation of the plant, i.e. the *Model*, through a *Perception* process. This model contains both the instantaneous state of the plant and more permanent general knowledge about it. It is the explicitness of this last static knowledge what differentiates the ECL from other control patterns. In these other cases, the static knowledge —i.e. the plant model—, which is application-dependent, is assumed static, being embedded into the controller together with the control algorithm, so it is not possible to change or incorporate an element to the control schema without entirely re-implementing it. With an explicit model bearing all the information used in the different elements of the

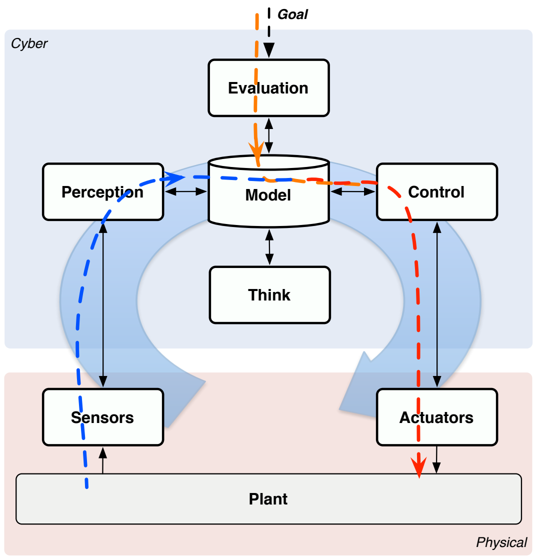


Figure 2. The Epistemic Control Loop Pattern structure. Thin arrows show structural connections between roles, with the arrow head indicating the direction of the data-flow, whereas thicker dashed arrows show the basic flow of information that leads to action generation.

control, the ECL design allows for changing the algorithm of any element of the control, or incorporating a new one, without modifying the rest.

Structure The ECL pattern proposes an structural separation of controller roles. The *Perception* process in ECL consists of the processing of the available input form the *Sensors* to update the estimation of the plant state contained in the *Model*. The *Evaluation* process evaluates the estimated state in relation with the current *Goal* of the loop —a generalization of the error signal of classical feedback control. The *Control* is responsible for generating proper action by using the evaluation result, the information about possible actions contained in the *Model*, and action-generation knowledge, for example planning methods. The action is then sent to the *Actuators* that execute it. The *Think* process include additional reasoning activities operating on the *Model*, e.g. to improve the state estimation, together with any operations that involve the manipulation of knowledge, such as consolidation or prediction. All application specific knowledge is contained in the *Model*. It is accessed and manipulated by the rest of the processes through standard interfaces. This can be implemented using the *Database Management System* pattern, for example.

Dynamics The ECL defines a cyclic operation in which each cycle follows the perceive-reason-act sequence, although the *Model* serves as a decoupling element that prevents the blocking of the operation caused by a failure in any of the steps. The *Perception* process in ECL corresponds to the first step and may include other reasoning operations as described previously. The *Evaluation* process then generates value from the current estimated state in the *Model*. In the last phase of the loop the *Controller* produces the most appropriate action based on the information available.

`Related Patterns` The ECL pattern is rooted on well established control patterns: feedback [17], model-based predictive control [22] or model-based control [23].

`References` The RCS (Real-time Control System) node [24] defines similar functions that are required in a real-time intelligent control unit. The PEIS (Physically Embedded Intelligent System) loop [25] also considers the aggregation of distributed control components with different functionalities. Boyd's OODA Loop (Observe, Orient, Decide, and Act) [26] is a concept originally applied to the combat operations process, often at the strategic level in both the military operations. It is now also often applied to understand commercial operations and learning processes.

B. The MetaControl Pattern

`Name` MetaControl (MC).

`Aliases` Meta Architecture (HUMANOBS project).

`Problem` The MetaControl pattern addresses the problem of designing a control system that is self-aware, *i.e.* it understands its mission, in the sense of detecting when its behavior diverges from its specification; it understands itself, meaning that it can reason about how its own state realizes a certain functional design in order to fulfill its mission; and it can reconfigure itself when required to maintain its behavior convergent towards its mission fulfillment.

`Solution` MC proposes a separation of the control system into two subsystems (see Figure 3): the *Domain Subsystem*, which consists of the traditional control subsystem responsible for sensing and acting on the plant so as to achieve the domain goal given to the system —e.g. move the mobile robot to a certain location, grab a certain object with a robotic hand. . . —; and the *Metacontrol Subsystem*, which is a control system whose plant is in turn the *Domain Subsystem*, and whose goal is the system's mission requirements.

`Structure` The two subsystems in which the control system is to be divided operate in different domains. The Domain Subsystem operates in the application domain, and could be patterned after any arbitrary control architecture, for example the navigation architecture proposed in [27] for a mobile robot. The pattern imposes the following requirements on the Domain Subsystem: i) its implementation has to provide a *monitoring* infrastructure, providing data at run-time about the processes and elements realizing the domain control, ii) some redundancy, not necessarily physical but more interestingly analytical, in the sense of having alternative designs to realize some functions [28], and iii) the implementation platform shall include mechanisms for *reconfiguration* to exploit that redundancy.

`References` The separation of the domain control and the meta-control is at the core of AERA, the architecture for autonomous agents developed in the HUMANOBS project

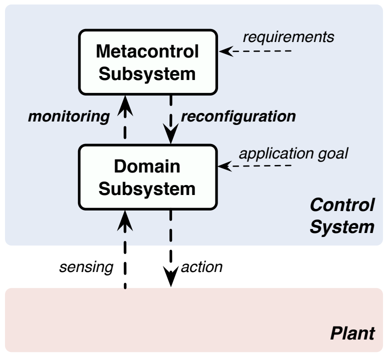


Figure 3. The structure proposed by the MetaControl Pattern.

(Humanoids the Learn Socio-Communicative Skills by Imitation, www.humanobs.org). The issue of metacontrol is also discussed related to reconfiguration of control systems in [29], and in supervisory control in fault-tolerant systems [28].

C. The Deep Model Reflection Pattern

`Name` Deep Model Reflection (DMR).

`Problem` This pattern addresses the problem of how to use the engineering model of a control system as a self-representation, so the system can exploit it at run-time to adapt its configuration in order to maintain its operation converging to its goal.

`Solution` Develop a metamodel capable of explicitly capturing both i) the static engineering knowledge about the system's architecture and functional design, and ii) the instantaneous state of realization of that design. This *Functional Metamodel* has to be machine readable to be usable by a model-based controller. A mapping from the languages used to design the system to this metamodel is necessary, in order to generate the run-tim model of the system from the engineering model of it. Automatic generation is possible if both conform to a formal metamodel, and a transformation between both metamodels exists (Figure 4).

`References` Metamodeling is a core topic in the domain of software modeling [30]. Functional modeling has been addressed in many disciplines, for example in the control of industrial processes [31].

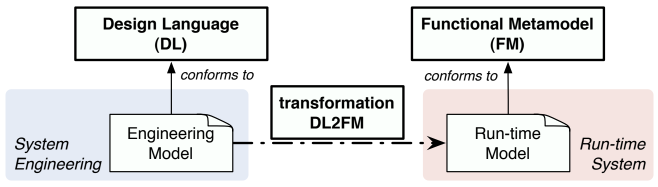


Figure 4. The roles involved in the Deep Model Reflection Pattern.

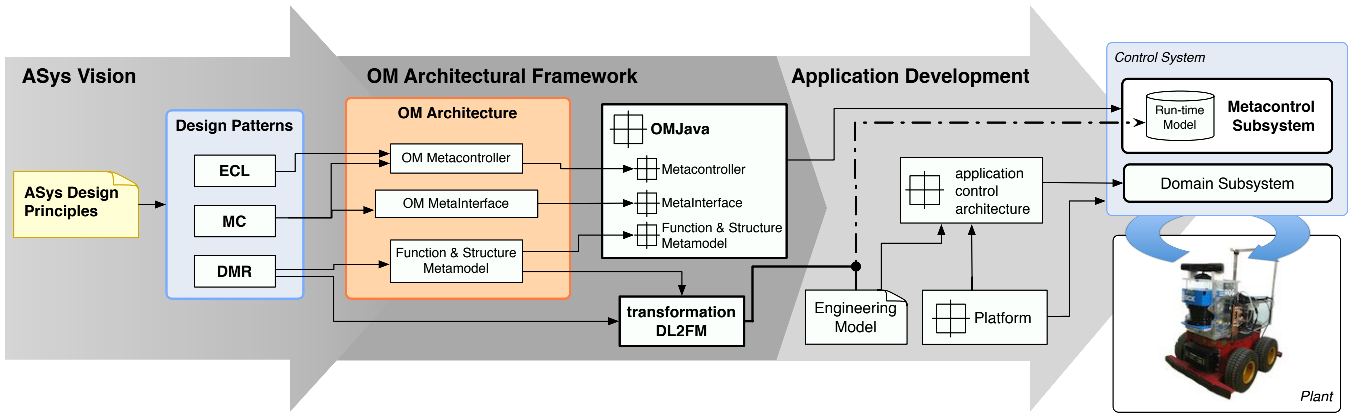


Figure 5. The methodological path followed in the development of the OM Architecture, with the core assets for ASys resulting from it.

IV. The OM Architecture

The three patterns presented provide partial solutions to the problem of designing robust control architectures for autonomous systems. They can be used in the construction of more complete architectures, as is the case of the Operative Mind (OM) reference architecture, a complete and general solution, synthesized by integrating the three patterns and realizing them in reusable software.

The conceptual and methodological path to OM is depicted in Figure 5. From left to right: a principled approach to robust cognition and self-awareness is captured as a set of design patterns; they are applied to synthesize the OM Architecture and build an application independent implementation as a Java package; it is then used, together with platform specific available assets, to implement the control architecture of a mobile robot.

A. *OM Architectural Assets*

OM offers a set of interrelated engineering assets for the development of specific control architectures (see Fig. 6):

1) MetaInterface: The application of the MetaControl pattern to our reference architecture has resulted in an interface that specifies the contract between the Domain and Metacontrol Subsystems' implementations.

2) Metacontroller: The Metacontroller is a refinement of the MetaControl pattern that specifies the design of the Metacontrol Subsystem by application of the ECL pattern. It defines an structure of two nested ECL loops: *ComponentsECL* realizes a servo-control loop of the configuration of the Domain Subsystem, to which it is connected for sensing and acting through the MetaInterface. The ComponentsECL goal is to keep a certain desired configuration, given by the action of the outer loop, the *FunctionalECL*, whose sensory input is the current configuration as estimated by the ComponentsECL and its goal is the system specification. The FunctionalECL evaluates the observed configuration by determining how well it realizes the functions designed to address the application requirements —*i.e.* the *mission*—, which is the goal of the FunctionalECL loop, and acts by producing a reconfiguration when necessary. For their operation, both ECL loops rely on a shared model which captures the engineering knowledge about the domain subsystem.

3) Function & Structure Metamodel: To design the knowledge that the OM Metacontroller exploits for control purposes —*i.e.* its *Model*— we have applied the DMR pattern. This pattern prescribes the explicit use of design-time models. For that purpose we have used an ontological approach to modeling [32]: we have compiled all the necessary concepts required for the explicit representation of the structural and the functional aspects of a system, to later formalize it in the Function & Structure Metamodel. The metamodel contains elements to account for the two referred aspects of an autonomous system:

Structure elements: concepts to represent the configuration of the system in terms of components and their connections.

Functional elements: concepts that capture the teleology of the system, in the sense of Lind [31] of representing the roles the designer intended for the components of the system to achieve the objectives of the system. These representations constitute design solutions for the required functionality.

The metamodel has been specified in UML, and a Platform Specific Model (PSM) has been implemented in Java.

B. *Use of OM in an autonomous mobile robot*

This section describes the application of the OM Reference Archiecture to develop the control architecture of a mobile robot capable of robustly perform standard navigation tasks.

1) The patrol robot testbed: A patrolling mobile robot testbed has been used to validate the OM Architecture. This application was selected because it involved heterogeneous components, both hardware and software, and had a sufficient level of complexity to prove for generality.

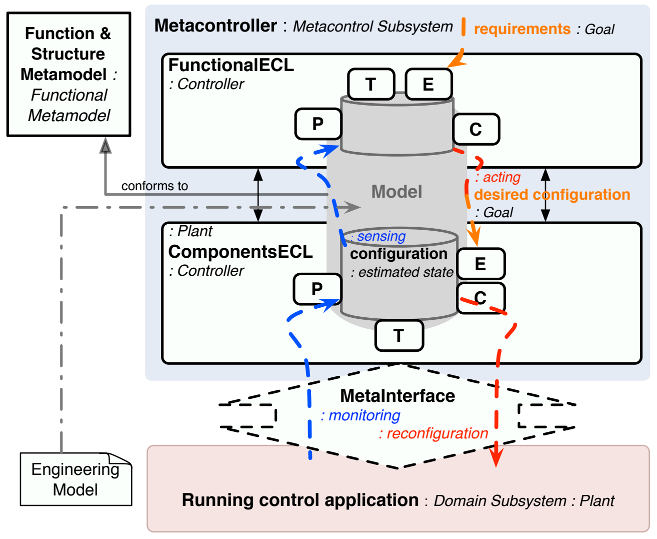

Function & Structure Metamodel : Functional Metamodel
Metacontroller : Metacontrol Subsystem
requirements : Goal
FunctionalECL : Controller
T
E
P
C
Model
conforms to
: acting
desired configuration
: Goal
: Plant
ComponentsECL : Controller
: sensing
configuration : estimated state
E
P
C
T
MetaInterface
: monitoring
: reconfiguration
Engineering Model
Running control application : Domain Subsystem : Plant

Figure 6. The interplay of the main elements of the OM Architecture Model in the operation of a metacontroller. Functional and Components loops are patterned after the ECL (hence including **P**erception, **T**hink, **E**valuate and **C**ontrol activities around a **M**odel). The roles that each element plays are written in italics after colon marks.

The basic use case consists of the robot (a Piooner 2AT8 platform instrumented with a SICK LMS200 laser sensor, a Kinect and a compass) navigating through an indoor office environment to sequentially visit a number of given waypoints. The environment has no dynamic obstacles and an initial map is provided. Some runtime deviations from it are possible, *e.g.* chairs may have moved, in order to provide for realistic levels of uncertainty. Two scenarios were envisaged to test the benefits of the application of the OM Architecture in terms of robustness and autonomy: i) a temporary failure of the laser driver, ii) a permanent fault in the laser sensor. In both cases the system should be able to detect it and reconfigure its organization to adapt to the current state of affairs, in order to maintain the mission.

2) Platform development: The platform selected to implement the domain control system is ROS [4] for several reasons: i) its middleware infrastructure provides with mechanisms for *monitoring* and *reconfiguration* as prescribed by the `MetaControl` pattern, ii) its computation model of nodes that publish and subscribe to message channels or topics fits in a component-based architecture model, being thus modelable with our Function & Structure Metamodel; and iii) there are open source ROS implementations of components for navigation of mobile robots available.

For the Domain Subsystem of the control architecture we have used the ROS navigation stack [27], which we have tuned for our Pioneer mobile platform, and complemented with other available ROS packages for the robot Kinect and Laser sensors, and additional ROS necessary for the patrolling mission.

The metacontroller is implemented using the OMJava package. It provides a domain independent and multiplatform implementation of the OM Architectural Model (see Figure 5), including a complete implementation of the Function & Structure Metamodel, the OM Metacontroller, and a Java specification of the MetaInterface.

In order to integrate the OMJava implementation of the Metacontroller with the ROS-based navigation control architecture, an application-independent ROS implementation of the MetaInterface has been developed as a set of ROS nodes. These nodes, together with another one which wraps the OMJava Metacontroller as a ROS node, constitute a PSM of the OM Architecture for the ROS platform.

3) Application development and validation: So far we have described the implementation of: i) a Java library which provides an implementation of OM Metacontroller (OMJava), ii) a ROS PSM of the OM Architecture (OMrosjava), which includes ready-to-deploy OM-based metacontrol assets for any robot application implemented with ROS.

Thanks to the architectural model-based approach, to implement the concrete testbed it is only necessary to generate the application model according to the Function & Structure Metamodel, in order to provide the Metacontroller with explicit knowledge about the system: i) its mission —core functionality required—, ii) its structure —its components and their properties—, and iii) its functional design —available design solutions that realize the core functionality through certain configurations of its structure.

Following we briefly describe how the application independent OM processes we have developed exploit the aforementioned knowledge in each of the scenarios envisaged:

Scenario 1: the Metacontroller detects the failure of the laser driver and repairs the component it by re-launching the software process. Only the ComponentsECL intervenes, since the solution is achieved at the structure level.

Scenario 2: this time it is not possible to relaunch the laser driver because the error is due to a permanent fault in the laser device. the ComponentECL detects this and the situation scales to the FunctionalECL loop. The functional failure caused by the unavailability of the laser driver is assessed, and an alternative configuration that fulfills the functionality required to maintain the mission is generated. This configuration makes use of the Kinect sensor instead of the laser. The new configuration is commanded to the ComponentsECL, which reconfigures the navigation systems accordingly. In summary, the robot redesigns its control architecture using available components.

V. Future Work

Our current pattern language contains only a small number of patterns centered on core ASys issues. It is necessary to complete it with more common, practical patterns to achieve a full *generative pattern language* [33].

The current implementation of the ICe —the Integrated Control Environment— is just a collection of engineering resources atop the Rational Software Development Platform. Our current effort is to use the Eclipse RCP to create a specific IDE for the ASys engineering process. There is an ongoing work in the automation of some of the transformation processes. For example, concerning the current implementation of the OM Architecture Model, the automatic transformation from the engineering design language to our Function & Structure metamodel is still under development. The MDD transformations necessary to complete the ICe toolchain are still in early stages.

The Functions & Structure metamodel now only models basic structural aspects of control systems. It is necessary to incorporate behavioral aspects to our current metamodel so the Metacontrol will be able to handle function-centric, dynamical issues in the Domain Subsystem. It is necessary to improve the metamodel by including the full ECL and OASys [34] concepts to further specify functionality. An specially important ongoing work is the self-closure of the MC pattern: the application of the MetaControl pattern to the Metacontrol Subsystem, so it becomes also part of the Domain Subsystem it controls.

The ambition of ASys is of universality; and hence there is a need for domain generalization, i.e. the extension of theoretical concepts and technological assets to other domains. Current efforts are centered around *autonomous robots* and *continuous process control systems* [11], but other technological domains are under consideration —e.g. utilities, telecoms or maintenance systems. Even more, while the patterns described in this paper are technological designs for autonomous artifacts, their content may find strong biological roots in animal cognition [35]. In this sense, the ASys research programme may have impact not only in how engineers build autonomous robots, but also in how cognitive scientists understand the mind [36].

VI. Conclusions

This work shows a pattern-based approach to the construction of sophisticated, self-aware control systems in the domain of autonomous robots. The three patterns — Epistemic Control Loop (ECL), MetaControl (MC), and Deep Model Reflection (DMR)— offer valid reusable design assets for the implementation of custom architectures for autonomous systems.

The development of the testbed application demonstrated the benefits of following a pattern-based approach in the implementation of a resilient control architecture for a robot.

The patterns, as instantiated in the OM Architecture, were easily applicable thanks to the availability of a domain neutral implementation (OMjava). From it, the production of the ROS platform-specific model was straightforward, and only slightly hampered by the lack of a formal Platform Definition Model for ROS. Considering strictly only the development of the testbed application, the addition of our reference architecture produced only a minor extra-effort when compared with a standard development of the control architecture for the mobile robot.

The three patterns offered solutions to very different problems both at design time and runtime, but as the OM Reference architecture and the OMjava realization demonstrate in the testbed, they are easily integrable and can successfully collaborate in generating better system architectures.

The pattern-based, model-centric approach to the construction of autonomous controllers proposed by ASys can offer possibilities —both for engineering and run-time operation— that go well beyond current capabilities of intelligent autonomous robots. In this direction, the application of our OM Architecture, rooted on the three design patterns described in the paper, has provided the robot with deep runtime adaptivity based on a functional understanding, hence demonstrating enhanced robust autonomy through cognitive self-awareness.

Remember that the ASys programme seeks *robust teleonomy for unreliable systems in uncertain environments*. The model-based, self-aware adaptivity approach supported by these patterns departs from conventional robust control approaches, offering a more open-ended pathway for system adaptation.

Acknowledgements

The authors would like to thank the European Commission for grant HUMANOBS (Humanoids the Learn Socio-Communicative Skills by Imitation).

References

[1] N. Nilsson, “A mobile automaton: An application of artificial intelligence techniques,” AI Center, SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025, Tech. Rep. 40, Mar 1969, sRI Project 7494 IJCAI 1969.

[2] R. A. Brooks, “A robust layered control system for a mobile robot,” IEEE Journal of Robotics and Automation, vol. 2, no. 3, 1986, pp. 14–23.

[3] M. Lindstrom, A. Oreback, and H. I. Christensen, “Berra: a research architecture for service robots,” in IEEE International Conference on Robotics and Automation, 2000. Proceedings. ICRA ’00., San Francisco, CA, USA, April 2000, pp. 3278–3283 vol.4.

[4] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in ICRA Workshop on Open Source Software, 2009.

[5] M. Shaw and D. Garlan, Software Architecture. An Emerging Discipline. Upper Saddle River, NJ: Prentice-Hall, 1996.

[6] W. Houkes and P. Vermaas, Technical Functions. On the Use and Design of Artefacts. Springer, 2010.

[7] J. Miller and J. Mukerji, “MDA guide v1.0.1,” Object Management Group, Tech. Rep. omg/03-06-01, 2003.

[8] R. Sanz and M. Rodríguez, "The ASys Vision. Engineering Any-X autonomous systems," Universidad Politécnica de Madrid - Autonomous Systems Laboratory, Technical Report ASLAB-R-2007-001, 2007.

[9] W. Meeussen, E. Marder-Eppstein, K. Watts, and B. P. Gerkey, "Long term autonomy in office environments," in ICRA 2011 Workshop on Long-term Autonomy, IEEE. Shanghai, China: IEEE, 05/2011 2011.

[10] B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic, "A domain-specific software architecture for adaptive intelligent systems," Software Engineering, IEEE Transactions on, vol. 21, no. 4, Apr 1995, pp. 288–301.

[11] M. Rodriguez and R. Sanz, "Development of integrated functional-structural models," in 10th International Symposium on Process Systems Engineers, Salvador, Brasil, August 16-20 2009, pp. 573–578.

[12] C. Hernández, I. López, and R. Sanz, "The operative mind: a functional, computational and modelling approach to machine consciousness," International Journal of Machine Consciousness, vol. 1, no. 1, June 2009, pp. 83–98.

[13] R. Sanz, I. López, M. Rodríguez, and C. Hernández, "Principles for consciousness in integrated cognitive control," Neural Networks, vol. 20, no. 9, 2007, pp. 938–946.

[14] R. C. Conant and W. R. Ashby, "Every good regulator of a system must be a model of that system," International Journal of Systems Science, vol. 1, no. 2, 1970, pp. 89–97.

[15] C. Landauer and K. L. Bellman, "Meta-analysis and reflection as system development strategies," in Metainformatics. International Symposium MIS 2003, ser. LNCS. Springer-Verlag, 2004, no. 3002, pp. 178–196.

[16] L. Balmelli, D. Brown, M. Cantor, and M. Mott, "Model-driven systems development," IBM Systems journal, vol. 45, no. 3, 2006, pp. 569–585.

[17] R. Sanz and J. Zalewski, "Pattern-based control systems engineering," IEEE Control Systems Magazine, vol. 23, no. 3, June 2003, pp. 43–60.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, ser. Addison-Wesley Professional Computing Series. New York, NY: Addison-Wesley Publishing Company, 1995.

[19] J.-M. Perronne, L. Thiry, and B. Thirion, "Architectural concepts and design patterns for behavior modeling and integration," Math. Comput. Simul., vol. 70, no. 5-6, Feb. 2006, pp. 314–329.

[20] R. Sanz, A. Yela, and R. Chinchilla, "A pattern schema for complex controllers," Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference, vol. 2, Sept. 2003, pp. 101–105 vol.2, .

[21] N. G. Leveson, Engineering a Safer World Engineering a Safer World: Systems Thinking Applied to Safety. The MIT Press, 2012.

[22] A. Pike, M. Grimble, A. O. M.A. Johnson, and S. Shakoor, The Control Handbook. CRC Press, 1996, ch. Predictive Control, pp. 805–814.

[23] P. M. van den Hof, C. Scherer, and P. S. Heuberger, Model-Based Control: Bridging Rigorous Theory and Advanced Technology. Springer, 2009.

[24] J. S. Albus and A. J. Barbera, "RCS: A cognitive architecture for intelligent multi-agent systems," Annual Reviews in Control, vol. 29, no. 1, 2005, pp. 87–99.

[25] A. Saffiotti, M. Broxvall, M. Gritti, K. LeBlanc, R. Lundh, J. Rashid, B. Seo, and Y. Cho, "The PEIS-ecology project: vision and results," in Proc of the IEEE/RSJ Int Conf on Intelligent Robots and Systems (IROS), Nice, France, 2008, pp. 2329–2335.

[26] F. P. Osinga, Science, Strategy and War: The Strategic Theory of John Boyd. Routledge, 2007.

[27] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The office marathon: Robust navigation in an indoor office environment," in Robotics and Automation (ICRA), 2010 IEEE International Conference on, may 2010, pp. 300 –307.

[28] M. Blanke, M. Kinnaert, J. Lunze, and M. Staroswiecki, Diagnosis and Fault-Tolerant Control. Springer-Verlag Berlin, 2006.

[29] J. L. de la Mata and M. Rodríguez, "Accident prevention by control system reconfiguration," Computers & Chemical Engineering, vol. 34, no. 5, 2010, pp. 846 – 855.

[30] T. Kühne, "Matters of (meta-)modeling," Software and System Modeling, vol. 5, no. 4, July 2006, pp. 369–385.

[31] M. Lind, "Modeling goals and functions of complex industrial plants," Applied Artificial Intelligence, vol. 8, 1994, pp. 259–283.

[32] J. Bermejo-Alonso, R. Sanz, M. Rodríguez, and C. Hernández, "An ontological framework for autonomous systems modelling," International Journal on Advances in Intelligent Systems, vol. 3, no. 3, 2010, pp. 211–225.

[33] D. Brugali and K. Sycara, "Frameworks and pattern languages," ACM Computing Surveys, March 2000.

[34] J. Bermejo-Alonso, "OASys: An ontology for autonomous systems," Ph.D. dissertation, Departamento de Automática, Universidad Politécnica de Madrid, 2010.

[35] C. Hernández, R. Sanz, J. Gómez-Ramirez, L. S. Smith, A. Hussain, A. Chella, and I. Aleksander, Eds., From Brains to Systems. Brain-Inspired Cognitive Systems 2010. Springer, 2011.

[36] R. Sanz, "Machines among us: Minds and the engineering of control systems," APA Newsletters - Newsletter on Philosophy and Computers, vol. 10, no. 1, 2010, pp. 12–17.

A Method for Directly Deriving a Concise Meta Model from Example Models

Bastian Roth, Matthias Jahn, Stefan Jablonski
Chair for Databases & Information Systems
University of Bayreuth
Bayreuth, Germany
{bastian.roth, matthias.jahn, stefan.jablonski} @ uni-bayreuth.de

***Abstract*—Creating concise meta models manually is a complex task. Hence, newly proposed approaches were developed which follow the idea of inferring meta models from given model examples. They take graphical models as input and primarily analyze graphical properties of the utilized shapes to derive an appropriate meta model. Instead of that, we accept arbitrary model examples independent of a concrete syntax. The contained entity instances may have assigned values to imaginary attributes (i.e., attributes that are not declared yet). Based on these entity instances and the possessed assignments, a meta model is derived in a direct way. However, this meta model is quite bloated with redundant information. To increase its conciseness, we aim to apply so-called language patterns like inheritance and enumerations. For it, the applicability of those patterns is analyzed concerning the available information gathered from the underlying model examples. Furthermore, algorithms are introduced which apply the different patterns to a given meta model.**

Keywords-meta model derivation; meta model inference; conciseness of meta models; pattern recognition; language patterns; inheritance

I. Introduction

Manually creating domain-specific languages (DSL), especially with a concise meta model as abstract syntax, is a complex task [1]. Besides an abstract syntax, a typical DSL also consists of a concrete syntax and a set of semantic rules (constraints) [1]. In this paper, the focus lies on the abstract syntax defined by a meta model. For defining such a meta model, new development methods have emerged. Those methods focus on deriving or inferring a meta model from a given set of example models [2, 3]. However, they only marginally consider the conciseness aspect of the resulting meta model (if at all). According to [4], this is a very important quality criteria of meta models. Therefore, our primary goal is to obtain a meta model with a high degree of conciseness. To achieve this, a typical solution is to apply language patterns like single inheritance, multiple inheritance and enumerations to a constellation of meta model entities (for more information about conciseness see section III).

Since the resulting meta model should represent the abstract syntax of a DSL another important goal of our approach is to derive a meta model which highly corresponds to concepts describing the domain. Hence, we have to gain the domain entities' instances from the model examples. Such instances directly can be modeled when using the Open Meta Modeling Environment (OMME) introduced by Volz et al. in [5]. Consequently, the paper at hand originates in the context of OMME. In the following sub section, we shortly explain the relevant characteristics of this platform.

A. The Open Meta Modeling Environment

OMME is an Eclipse-based meta modeling tool [6] that allows developers to define their own modeling language. It goes far beyond the capabilities of competing tools with respect to its support for advanced language patterns (e.g. Powertypes [7]). Its implementation is based on the Orthogonal Classification [8] and uses Clabjects [9] for representing concepts of a model (the term "concept" in the context of OMME always means a Clabject). Hence, OMME provides a Linguistic Meta Model (LMM) and interprets (meta) models at runtime in order to emulate a concrete textual syntax.

Below, we predominantly limit ourselves to concepts which can act as both, types and instances. As a type (also called a meta concept), a concept declares attributes whereas as an instance (also called an instance concept), a concept contains assignments each of which may be associated with an attribute. If such an association exists the target attribute must be declared by the type (meta concept) of the assignment's owner. Attributes and assignments can be divided into literal and referential ones depending on their respective type. OMME supports the following literal types: boolean, integer, double and string. In our understanding, enumerations are regarded as literal types, too. That is tolerable because enumerations can also be represented by integers with a highly restricted range of values. Each defined concept, however, may be used as a referential type. While modeling using the LMM, the applicable language patterns can be selected according to a user's needs (e.g., enabling or disabling multiple inheritance). Below, each suchlike configuration is called a modeling context.

B. Fundamental assumption on equally named elements

The most important assumption we take is that equally named elements (types of concepts on the one hand, assignments and attributes on the other hand) always relate to the same semantic object at domain side. One could imagine a meta model containing two different concepts each with exactly one string attribute labeled as `owner`. When trying to make this meta model more concise, both concepts are deemed to be candidates for generating a common super concept because of the two equally named attributes.

This assumption is mandatory. Otherwise, neither a meta model can be derived from one or more example models nor the conciseness of a given meta model can be enhanced. Both approaches presented in section V infer graphical DSLs and follow a comparable principal. They state that two shapes correspond to each other if their graphical properties are identical.

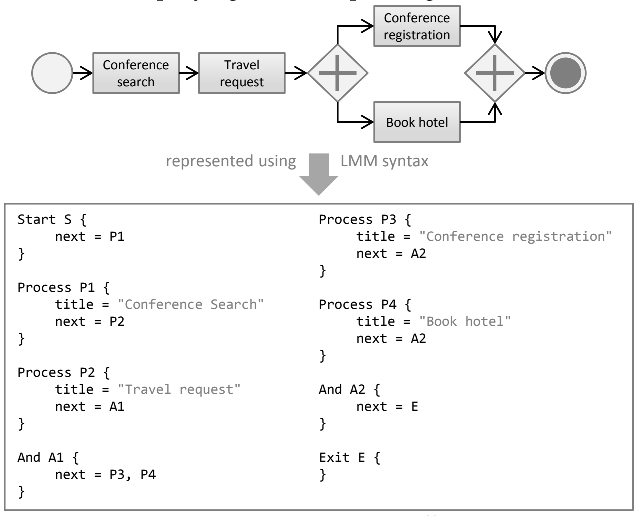
II. Example Model

As an example model we constructed the process shown in Fig. 1 using two different concrete syntaxes. The top part shows the graphical representation with nodes and directed edges. It is just depicted for a better comprehensibility since a graphical process model is easier to understand than a textual one. In the right, the same model is written down using the concrete textual syntax given by the LMM.

Below, we focus on the textual representation because it directly uses constructs of the LMM. Since the LMM syntax is quite similar to the one of popular object-oriented programming languages it is easy to read for software developers and modelers. The mapping rules between both representations lie beyond the scope of this paper, so the following mapping is taken for granted: The circle node on the left is considered as `Start` concept with identifier `S`. It contains an assignment `next` which refers to concept `P1` and represents a successor relationship. `P1` to `P4` are specified as Process concepts. Each of them has a title and also a `next` assignment. `A1` and `A2` represent instances of concept `And`. Both again contain a next assignment. However, assignment `next` of `A1` holds two references to `P3` and `P4`. The last circle on the bottom represents the process models `Exit`. It contains no further assignments. The arrows between the different nodes can be seen as successor relationships which are always mapped to according `next` assignments.

III. Concise meta model using language patterns

One important goal in meta modeling is keeping meta models concise [4]. Therefore, models need to be as small as possible, i.e., they should completely describe their according domain with as few constructs as possible. Achieving this is a general problem when building meta models. For instance, the authors of the newly published version 2.5 of UML have focused on simplifying the corresponding meta model [10].



```
Start S {                              Process P3 {
     next = P1                              title = "Conference registration"
}                                           next = A2
                                       }
Process P1 {
     title = "Conference Search"       Process P4 {
     next = P2                              title = "Book hotel"
}                                           next = A2
                                       }
Process P2 {
     title = "Travel request"          And A2 {
     next = A1                              next = E
}                                      }

And A1 {                               Exit E {
     next = P3, P4                     }
}
```

Figure 1. Example model visualized using two different concrete syntaxes

Making a meta model concise can be accomplished by applying so called language patterns to suitable constellations of meta elements [11]. In literature, it is not exactly specified how a suitable constellation looks like. There are only suggestions in form of best practices or guidelines when to apply a certain pattern (comparable to the applicability of design patterns [12]). Because these guidelines are suggestions they are mostly formulated quite imprecise with a subjective touch. Most guidelines base on domain-specific background knowledge (e.g., the “is a”-statement mentioned in the following sub section A for using single inheritance). In general, such information is not available. Hence, we have to rely on the information provided by the model examples as well as the structure of the derived meta model (i.e., attributes and their referential or literal types).

In the following, three typical language patterns that are supported by OMME and partially many different other modeling frameworks (e.g., EMF, MetaGME, eMOFLON) are presented. For enumerations, we do not elaborate further because their usage is straightforward. They basically allow for restricting the value range of an attribute to a few predefined literals.

A. *Single inheritance*

Single inheritance is a well-known and widespread language pattern stemming from the field of object-oriented programming languages. There, it allows for introducing generalization/specialization hierarchies on classes. The key feature necessary for our approach is that a specialized class inherits all fields of its super class.

The most common rule for introducing a specialization relationship is: if an “is a”-relationship can be identified between two classes [13] (or entities like stated in [14]) the source of this relationship specializes the target. To identify this kind of relationship, background knowledge about the domain is required which cannot be directly expressed through the model example(s). Therefore, we follow the proposal of [15] and interpret a set of shared attributes as indicator for an inheritance relation. In some cases, for a given model example the introduction of a specialization relationship is indispensable. This occurs if an attribute is intended to reference two or more different classes. Then, those referable classes need a common super class which has to be the type of the aforesaid attribute. An example for that is demonstrated in section IV.B step 3. This additional information can only be retrieved from the model examples and not directly from the meta model. That is the case because merely in instances different concepts may be assigned to attributes (according to their respective types). A referential attribute, however, always expects exactly one type.

Another important topic when using inheritance is a rather flat generalization hierarchy. Otherwise, the meta model gets quite complex and thus its comprehensibility suffers.

B. *Multiple inheritance*

Multiple inheritance is often criticized as risky because of potentially occurring problems as stated by Singh in [16] (e.g., name collision and repeated inheritance). Hence, we only utilize multiple inheritance to meet addressability constraints found within the original model example(s). Addressability means the two possible referencing aspects, namely "a concept is referenced by another one" and "a concept refers to another concept". An adequate example can be found in section IV.C.2) where an algorithm is proposed for applying the multiple inheritance language pattern to a given meta model. This restriction protects also from over-generalizing the resulting meta model.

IV. META MODEL DERIVATION

When deriving a meta model from a model example, the directly recognizable constraints need to be softened in some way. Otherwise, solely the provided model example can be re-modeled without any differences. This softening behavior needs to be highly configurable since the statement whether a meta model is concise or not is always subjective. Therefore, our prototypical implementation provides many according parameters which allow for fully customizing the derivation behavior. However, the given default settings represent the notion of a concise meta model based on our experiences and best practices.

In the following, we introduce our direct method for deriving a concise meta model out of one (or more if available) model example(s). Direct method means that we directly work with constructs given by the LMM. In the first instance we refer to concepts, assignments and attributes. The whole method can be divided into two main parts, according to the necessity whether applying language patterns is required or not:

- Bottom-up part: for each found unique type a separate meta concept is created with all required attributes. After that, language patterns are applied that are mandatory for obtaining a valid meta model as defined by the LMM's semantics.
- Conciseness part: analysis of the generated meta model to find constellations of concepts to which further language patterns can be applied. These constellations are identified according to the statements about the particular patterns in section III.

A. *Reusable sub algorithms*

Below, three sub algorithms are presented that are reused at different places. So, their functionality is described once and referenced wherever needed.

1) *Merging a set of types using generalization*

The sub algorithm "merging types using generalization" has the task that for a given set of types, one common super type has to be determined (without moving contained attributes from the input types to a new common super type). Its functionality correlates to the one provided by the model evolution operations "extract super class" and "fold super class" described in [17]. Nevertheless, both operations always base on at least one common feature (in our terms: one common attribute) which is not the case for our algorithm.

The algorithm works as follows: Receiving a set of input types `ITs`, for each type `IT` the routine collects its super types and add them to the set `STs`. Those super types `STs` are analyzed whether each one of their specializations `SPs` (sub types) is contained by the set of input types `ITs`. If so the particular super type `ST` is a merging candidate `C`. After processing the input types, all found candidates `Cs` are merged to one common super type `CST` (disjunction). In case no super type candidates `Cs` are found, a new common abstract super type `CST` is generated. Finally, over `ITs` is iterated again. Thereby all specialization relationships from the type to any candidate `C` are removed. In place of that, a new specialization relationship is inserted from the type `IT` to the new common super type `CST`. As a cleanup, each super type `ST` that is no longer specialized is removed from the meta model. Furthermore, all references to the former super types `STs` (if exists) are replaced by according references to the new common one (`CST`). In addition to this informal description of the algorithm's functionality, Figure 2 gives an overview by means of a corresponding flow diagram.

After performing this algorithm, the resulting common super type may contain attribute duplicates. They may appear when merging several super type candidates to one common super type. Due to reusability reasons, it is not in the scope of this algorithm to resolve this inconsistency. That has to be done afterwards.

2) *Elimination of attribute duplicates*

Another frequently reused sub routine is "eliminating attribute duplicates". This algorithm takes a concept with inconsistent content as input. Inconsistency is enunciated by several equally named attributes which need to be merged to one single attribute.

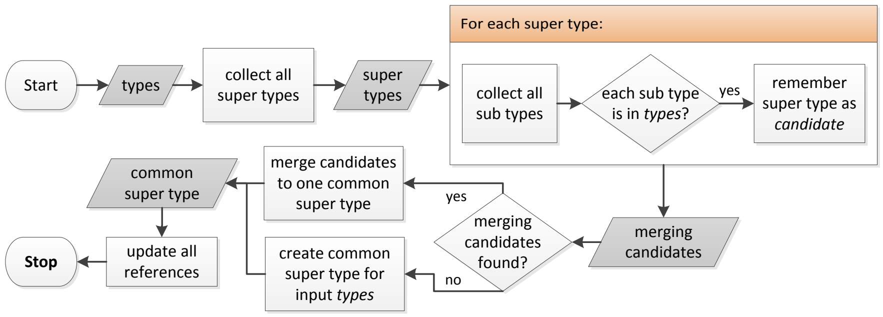

For each super type:
Start
types
collect all super types
super types
collect all sub types
each sub type is in *types*?
yes
remember super type as *candidate*
common super type
merge candidates to one common super type
yes
merging candidates found?
merging candidates
Stop
update all references
create common super type for input *types*
no

Figure 2. Flow diagram for "merging types using generalization" algorithm

Handling different cardinalities is quite easy. They are always widened to flexibility and consequently less limitations (e.g., 1 and 1..* turn to 1..*, 0..1 and 1..* to 0..*).

However, before addressing the type merging part, attributes have to be split up according to their kind, namely referential or literal. It is important to notice again that enumerations are regarded as literal attributes, too. The fork is necessary because referential attributes may lead to an indispensable introduction of a generalization hierarchy.

For instance, imagine a source concept with two equally named attributes whose types are referring to two different target concepts. In order to maintain the possibility of referencing instances of both target concepts within an instance of the source concept, the target concepts need a common super type. Thereby, for a set of equally named attributes the attribute types are extracted. If these types refer to different meta concepts for all those concepts a common abstract super concept is created and specialized by them (using sub algorithm 1)). Afterwards, only one of the original attributes is kept and its type (the referenced meta concept) is changed to the new common super concept. For another concrete example, see Figure 4 and Figure 5.

Eliminating or at least handling several equally named literal attributes happens in a different way. For it, we conceive three alternative strategies which may be configured as mentioned at the beginning of this section. The first one just informs the user about these ambiguities. The second strategy renames all duplicate attributes by means of a predefined rule (e.g., appending ascending numbers to the attributes' names). That also leads to small modifications within the model example(s) because the respective assignments must be updated as well. Using the third and last alternative conforms to our overall intention to a greater extent. We stated above that equally named constructs are considered to correspond to the same artifacts at domain side. Therefore, the third strategy merges the duplicate literal attributes based on type widening. This concept is comparable to the one of popular programming languages like Java and C#. Hereby, we allow type widening for all literal data types (enumerations included). When applying it to two different types then always the one with a greater value range is chosen. The ascendant order of the literal types according to their value range is as follows: boolean, enumeration, integer, double, string.

It may also occur that there are equally named referential and literal attributes at the same time. In this case it is obvious that only the first two strategies are expedient (i.e., inform the user or rename the concerning attributes and assignments). Due to the different inherent intents of literal and referential attributes, merging is not a valid option.

3) *Elimination of multiple inheritance*

Executing the task "eliminating multiple inheritance" is required if some concepts specialize more than one super type but multiple inheritance is not available in the current modeling context. At first, the according algorithm looks for concepts `Ts` which specialize at least two other concepts (set of all super types `STs`). Next, it iterates over all found concepts `STs`. For each concept `ST`, it selects all concerning sub concepts `SPs` that extend one or more super types specialized by `T`. Moreover, algorithm 1) is called by delivering all specializations `SPs` (`T` incl.) as input data.

Merging types this way may lead to attribute duplicates. They have to be eliminated by algorithm 2). Since its execution could again produce more than one super type per concept cyclic invocation of both algorithms may be necessary. This cycle will definitely terminate. At the latest this occurs when one global super type is found which is used as generalization for all other concepts.

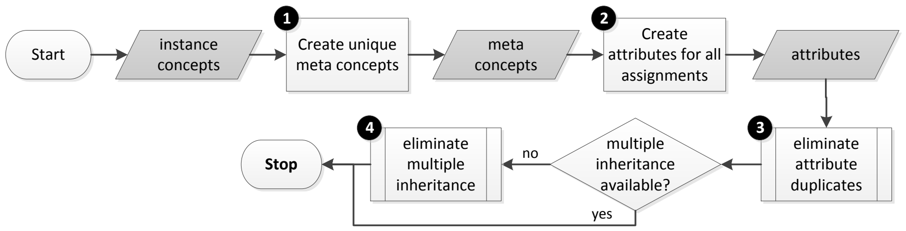
For eliminating multiple inheritance, extending the inheritance hierarchy about a further level is another conceivable solution. However, the solution is not universally valid (like the chosen solution stated above) because it cannot be applied to each constellation of concepts. For instance, that is the case if there are many different attributes which are mutually used within various concepts.

B. *Bottom-up algorithm*

The initial bottom-up algorithm (Figure 3) is considered as obligatory for deriving an initial meta model. For this algorithm, the (instance) concepts of one or more example models are taken as input data. The algorithm itself can be divided into four main steps.

Within the first step, for each uniquely identified type in the model example(s) a separate meta concept is created. Applied to the example from section II the unique meta concepts `Start`, `Process`, `And` and `Exit` are derived.

The second step infers attributes according to the assignments specified in the particular instantiating concepts. Hereby, for each assignment a corresponding attribute is created. This attribute takes over the name, the type and the cardinality of the assignment. In doing so, the cardinality's lower bound is set to 1 if each instance of the same type contains such an assignment, otherwise to 0. The upper one is set to 1 if every time only one value is assigned, else * is chosen. For literal assignments, the type can be directly read off because this recognition task is carried out by the LMM's parser. Handling referential assignments is more complex. If solely one concept is referenced then its type is directly borrowed from it. Otherwise, for each referenced concept its


Start
instance concepts
1
Create unique meta concepts
meta concepts
2
Create attributes for all assignments
attributes
3
eliminate attribute duplicates
multiple inheritance available?
no
yes
4
eliminate multiple inheritance
Stop
Figure 3. Coarse-grained flow diagram for the bottom-up algorithm

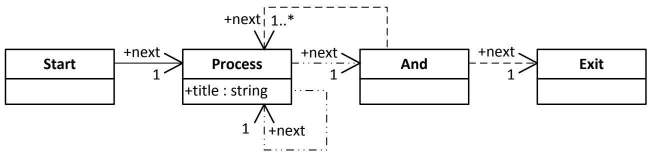


Figure 4. Meta model for the above example after executing step 2

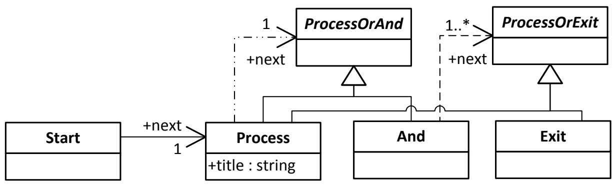


Figure 6. Meta model for the above example after executing step 3

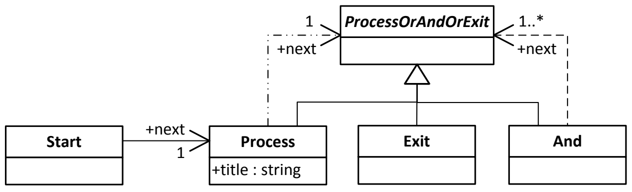


Figure 7. Meta model for the above example after executing step 4

type is detected individually. In case different outcomes occur, for every type a separate attribute is generated.

After finishing step 2, the meta model for the aforementioned example looks like the one depicted in Figure 4. Therein, the two `next` attributes of `Process` as well as the two of `And` must be merged in some way. This is done by invoking sub algorithm IV.A.2).

Thereby, two abstract super concepts are generated, namely `ProcessOrExit` and `ProcessOrAnd`. The so modified meta model is shown in Figure 5. The style of the arrows symbolizing the referential `next` attributes is the same as the style of the arrows which represent their sources in Figure 4.

Step 4 is merely required if multiple inheritance is disabled for the current modeling context. Then `Process` may only specialize one super concept. To achieve this, sub routine IV.A.3) is invoked. When applying it to the meta model from Figure 5, `ProcessOrAnd` and `ProcessOrExit` are merged to `ProcessOrAndOrExit` as depicted in Figure 6. Beyond that, the specialization relationships of `Process`, `And` and `Exit` must now point to `ProcessOrAndOrExit`. The same is true for the referential next attributes which refer to `ProcessOrAnd` respectively `ProcessOrExit`.

C. *Technical applicability of language patterns*

Below, for each supported language pattern a separate conciseness algorithm is presented that applies this pattern to a given meta model. Every conciseness algorithm requires so called "corresponding attributes" as input data. Thereby, two different correspondences need to be distinguished. As stated in the introduction, equally named attributes are intended to have the same meaning according to the particular domain. In other words, different attributes which correspond to each other always carry an identical name. The second correspondence bases upon the first one because sets of such corresponding attributes may be again subsumed to a superior set. In contrast, this correspondence does not base on the attributes' names but on their owners. Hence, two sets of corresponding attributes correspond only if each attribute of one set has a counterpart in the other set which both exhibit the same owner.

Before applying any language pattern, these corresponding attributes have to be determined and the according data structure must be built up. For it, all equally named attributes are put into appropriate sets. Depending on the underlying configuration, the attributes' types and cardinalities are regarded or ignored. Afterwards, the superior sets are created by extracting subsets from the former ones whose attributes meet the aforementioned owner criterion. This calculation task can be simplified by sorting the attributes within the former sets by their owners.

1) *Single inheritance*

The conciseness algorithm that applies single inheritance (Figure 7) can be split up into two variants. The first variant (yes-path) takes one of the input attributes' owner as common super type, whereas using the second variant (no-path) a new common super type is built up.

Choosing the particular variant bases on information gathered in step 1. Herein, the incoming attributes' owners are scanned for a concept which can be taken as common super type. Such a concept must declare all common attributes which can then be inherited by any sub concept (step 3).

In step 4, all corresponding attributes from the sub concepts are moved to the common super concept. This results in an inconsistent meta model because several equally named attributes occur within the super type. Then, step 5 invokes sub routine A.2) which resolves this inconsistency. However, execution of step 5 may bring multiple inheritance to the meta model (see section IV.B and especially Figure 5 for an according example). This potential problem is addressed by step 6 that encapsulates sub routine A.3).

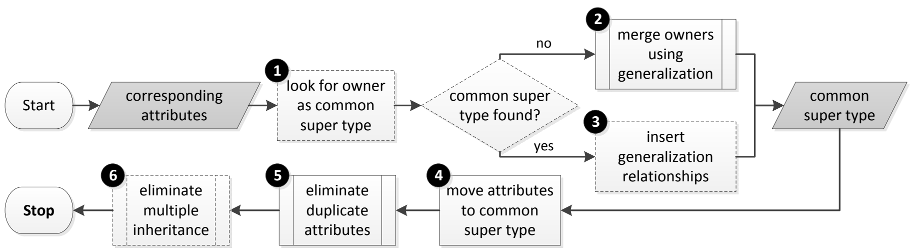

Start
corresponding attributes
1
look for owner as common super type
common super type found?
no
2
merge owners using generalization
yes
3
insert generalization relationships
common super type
4
move attributes to common super type
5
eliminate duplicate attributes
6
eliminate multiple inheritance
Stop

Figure 5. Flow diagram for the conciseness algorithm that applies single inheritance

Following the second variant (no-path) will be done if in step 1 no common super type is found and hence, one has to be determined. Then, step 2 calls the aforementioned sub routine IV.A.1). Applying it (the subsequent steps included) to the final meta model generated by the bottom-up algorithm (section IV.B), all three `next` attributes are delivered as input to the algorithm described above. Since none of the attributes' owners can be used as common super type those owners have to be merged accordingly. This has led to a new super type called `StartOrProcessOrAnd`. Afterwards, the `next` attributes are moved to this new super concept and merged (their common type is set to `ProcessOrAndOrExit`). Now, `Process` as well as `And` specialize two concepts, namely `StartOrProcessOrAnd` and `ProcessOrAndOrExit`. Due to the requirement of using single inheritance, both super concepts are merged to a single concept named `ProcessOrAndOrExitOrStart` and all references to the former ones are updated.

Apparently, `Exit` may also have a successor now, which was not intended by the model example. As explicated in section IV.A.3) (third sub algorithm), that is a negative side effect when restricting to single inheritance. This problem typically is solved by integrating a constraints system. When using a suchlike system, however, the brought constraint language needs to be studied first. All in all, that decreases the comprehensibility of the generated meta model and thus has a negative impact on its conciseness.

2) *Multiple inheritance*

Due to the aforementioned restriction to addressability constraints, the algorithm for applying multiple inheritance only has to consider referential attributes. Consulting the example from section II, concepts of type `Start` may never be "referenced by" any other concept. In doing so, an instance of `Exit` may not be able to have a successor by "referring to" any target concept (via `next`). However, reducing the number of equally named attributes is still our base intent. Keeping those two objectives in mind and applying them to the meta model depicted in Figure 5, the resulting meta model will look like the one visualized by Figure 8. Here, the two different concerns mentioned above ("references by" and "refer to") are implemented by means of a separate generalized concept. The first one is represented by `ProcessOrAndOrExit`, while the "refer to" aspect is established via `StartOrProcessOrAnd`.

Consequently, an appropriate algorithm needs to regard both aspects. However, utilizing the knowledge about the algorithm for applying single inheritance, the solution for multiple inheritance is similar. We directly take the algorithm for single inheritance and remove some superfluous steps.

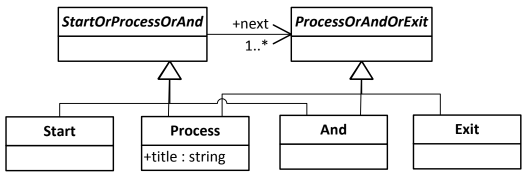


Figure 8. Meta model after applying the multiple inheritance language pattern

These superfluous steps are marked in Figure 7 by a dashed border. So, the resulting algorithm merely contains steps 2, 4 and 5. Besides, it only accepts referential attributes as input.

3) *Enumerations*

The conciseness algorithm for inferring enumerations is simpler than the two for applying single or multiple inheritance. Nevertheless, it requires more information as input, namely all assignments belonging to an attribute or a set of corresponding attributes. The selection whether to choose a single attribute or a set of corresponding attributes must be taken by the user in a previous configuration step. However, this has no impact on the main flow of the algorithm. Using a set of attributes just means to process more according assignments than with only one single attribute. From these assignments, the values are used to determine the resulting enumeration's literals. Hence, only literal attributes of type string are supported as input.

Whether an enumeration is generated or not depends on the diversity of values held by the different assignments. If there are merely a few values which are repeatedly assigned to that attribute(s) a new enumeration is derived. The varied values are taken as unique literals for this enumeration. Accordingly, the assignments have to be updated with the new literal values as well. So, when applying the enumeration language pattern the underlying model examples suffer small modifications. That is why this algorithm has to be executed before running the two others (for single or multiple inheritance).

V. RELATED WORK

As mentioned in the introduction, deriving a meta model from a set of model examples is not a totally new approach. Depending on their purpose, the available related work can be classified into two categories: meta model reconstruction and meta model creation.

Meta model reconstruction stems from the field of grammar reconstruction and grammatical inference [18]. Thereby, many textual sentences (ideally positive and negative samples) are analyzed to infer a grammar [19].

In current research, the Metamodel Recovery System (MARS) is one prominent representative for meta model reconstruction [20]. It receives a set of model samples and transforms them to a representation that can be used by a grammar inference engine. The output of this engine (a grammar) is then converted back to an equivalent meta model. As the title suggests, MARS focuses on the recovery of meta models (e.g., if a meta model got lost). To obtain a meta model which corresponds as much as possible to the original one, a large number of positive model samples is required. Otherwise the resulting meta model is strongly restricted in its capabilities. Since we mostly receive only one or at least a small set of model examples this approach is not practicable for us.

Up to our knowledge, there are only two research groups that generate a meta model by deriving it from very few model examples. BitKit as one representative has a rather different intention [21]. Its authors aim at supporting the pre-requirements analysis of software products by allowing to model in a freeform way just like with general purpose office tools. The resulting meta model is merely a means to an end. Primarily, BitKit semantically combines equally looking elements by deriving a common associated entity. After a meta model is inferred and, for instance, the color of such an element is changed the color of every other (equally looking) element is adapted accordingly. Due to the office tool intention of BitKit, the generated meta model is not intended to be processed in any further way. Consequently, its quality is not considered as well.

Another approach is proposed in [22]. Like BitKit, it is also restricted to graphical DSLs. Nevertheless, we adopt their general idea for applying patterns when inferring a meta model. That meta model (which represents the abstract syntax as stated by the author) highly corresponds to the concrete syntax as well. This correspondency is obvious when investigating another publication of Cho and Gray. In [23] they introduce some design patterns well suited for meta models. However, the presented patterns are very specific for graphical DSLs and hence not universally valid. That can be verified when comparing these patterns to the meta models for visual languages defined in [24]. In contrast to our approach, they mix the two identified main parts (section IV) when inferring a meta model. Hence, applying design patterns is strongly enmeshed in the bottom-up part. Thus, using our conciseness algorithms instead of their proposed “design pattern”-based approach is not possible without great effort.

VI. Conclusion and Future Work

In this paper, we presented a method for directly deriving a concise meta model from a small set of example models. To increase the conciseness of the resulting meta model, language patterns are applied to an appropriate constellation of meta concepts. Due to page limitations, we focused on widespread language patterns like inheritance and enumerations. As mentioned in section I.A, there are further language patterns (e.g. Powertypes) supported by OMME. Thus, we currently develop or extend the above conciseness algorithms for those patterns. Afterwards, we explore design patterns that can be applied similar to the way described above (but not only for visual languages like in existing solutions).

Our approach of automatically applying language patterns to meta concepts can also be reused for refactoring activities in modern IDEs like Eclipse or Visual Studio. Hereby, classes are considered as concepts whereas their fields are regarded as attributes. Taking the same assumptions as described in section I.B and providing appropriate configuration options, the presented conciseness algorithms can be taken for applying particular language patterns to a collection of classes. In future research, we also will deal with this topic in more detail.

References

[1] T. Clark, P. Sammut, and J. Willans, “Applied metamodelling: a foundation for language driven development,” CETEVA, 2008.

[2] H. Ossher, R. Bellamy, I. Simmonds, D. Amid, A. Anaby-Tavor, M. Callery, M. Desmond, J. de Vries, A. Fisher, and S. Krasikov, “Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges,” in *Proceedings of OOPSLA 2010*, vol. 45, 2010, pp. 848–864.

[3] H. Cho, “A demonstration-based approach for designing domain-specific modeling languages,” in Proceedings of SPLASH 2011, 2011, pp. 51–54.

[4] M. F. Bertoa and A. Vallecillo, “Quality attributes for software metamodels,” Proceedings of QAOOSE, 2010.

[5] B. Volz and S. Jablonski, “Towards an open meta modeling environment,” Proceedings of the 10th Workshop on Domain-Specific Modeling, 2010, pp. 17-1–17-6.

[6] B. Volz, Werkzeugunterstützung für methodenneutrale Metamodellierung. PhD thesis: University of Bayreuth, 2011.

[7] J. Odell, Advanced object-oriented analysis and design using UML. Cambridge University Press, 1998.

[8] C. Atkinson and T. Kühne, “Concepts for comparing modeling tool architectures,” Model Driven Engineering Languages and Systems, 2005, pp. 398-413.

[9] C. Atkinson and T. Kühne, “Meta-level independent modelling,” International Workshop Model Engineering in Conjunction with ECOOP 2000, 2000, pp. 12-16.

[10] S. Covert, “OMG’s Unified Modeling Language (UML) Celebrates 15th Anniversary,” 2012. [Online]. Available: http://www.omg.org/news/releases/pr2012/08-01-12-a.htm.

[11] C. Atkinson and T. Kühne, “The role of metamodeling in MDA,” Proceedings of the International Workshop in Software Model Engineering 2002, 2002, pp. 67-70.

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

[13] Microsoft Corporation, “When to Use Inheritance,” 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/27db6csx(v=vs.90).aspx.

[14] M. Gogolla and U. Hohenstein, “Towards a semantic view of an extended entity-relationship model,” ACM Transactions on Database Systems, 1991, pp. 369-416.

[15] R. Elmasri and S. B. Navathe, Fundamentals of database systems, 3. A. Amsterdam: Addison-Wesley Longman, 2000.

[16] G. Singh, “Single versus multiple inheritance in object oriented programming,” ACM SIGPLAN OOPS Messenger, 1994, pp. 30-39.

[17] M. Herrmannsdoerfer, S. Vermolen, and G. Wachsmuth, “An extensive catalog of operators for the coupled evolution of metamodels and models,” Software Language Engineering, pp. 163–182, 2011.

[18] M. Mernik, D. Hrncic, B. R. Bryant, A. P. Sprague, J. Gray, Q. Liu, and F. Javed, “Grammar inference algorithms and applications in software engineering,” in Proceedings of the 9th International Colloquium on Grammatical Inference, 2009, pp. 1–7.

[19] F. King-Sun and T. L. Booth, “Grammatical Inference: Introduction and Survey - Part I,” IEEE transactions on pattern analysis and machine intelligence, vol. 8, 1986, pp. 95-111.

[20] F. Javed, M. Mernik, J. Gray, and B. R. Bryant, “MARS: A metamodel recovery system using grammar inference,” Information and Software Technology, vol. 50, 2008, pp. 948–968.

[21] M. Desmond, H. Ossher, I. Simmonds, D. Amid, A. Anaby-Tavor, M. Callery, and S. Krasikov, “Towards smart office tools,” FlexiTools Workshop, 2010.

[22] H. Cho, J. Gray, and E. Syriani, “Creating visual Domain-Specific Modeling Languages from end-user demonstration,” Modeling in Software Engineering, 2012, p. 22-28.

[23] H. Cho and J. Gray, “Design patterns for metamodels,” Proceedings of SPLASH 2011, 2011, pp. 25-32.

[24] P. Bottoni and A. Grau, “A Suite of Metamodels as a Basis for a Classification of Visual Languages,” in Symposium on Visual Languages and Human Centric Computing, 2004, pp. 83–90.

Remodeling to Powertype Pattern

Matthias Jahn, Bastian Roth, Stefan Jablonski
Chair for Databases & Information Systems
University of Bayreuth
Bayreuth, Germany
{matthias.jahn, bastian.roth, stefan.jablonski} @ uni-bayreuth.de

***Abstract*— Nowadays, models often stand as first class objects in the field of software development. That's why clarity and understandability are important markers of high quality models. Therefore, several patterns exists that can help to improve model quality. However, developing a domain specific language is affected by understanding the domain of interest which often evolves during the development of the software system. This evolution again causes the language to change either. As a consequence of that, meta-modeling patterns are oftentimes inserted in an existing meta-model which results in various adaptions to migrate the system into a valid state. Since the current research has not discovered any techniques to cope with a remodeling to such a pattern these adaptions have to be done manually. Focusing on this challenge, we present in this article an evolution operator that creates a powertype within an existing model and furthermore adapts the other related models simultaneously.**

Keywords-powertype, extended powertype, remodeling to patterns, meta-model evolution, meta-model, deep instantiation

I. MOTIVATION

Today, developers often tend to define a separate modeling language for special parts of the domain of interest. That is especially the case if standard modeling languages do not cope with special application settings. This trend is referred to as domain specific modeling (DSM) and the resulting language is hence called domain specific language (DSL).

A modeling language in general consists of three parts: a definition of an abstract syntax, a definition of a concrete syntax, and a rule set (constraints) [1]. Thereby, meta-models are oftentimes used to express both the abstract and the concrete syntax. Hence, the quality of the resulting language is highly-coupled to the quality of the meta-models describing it. Consequently, these meta-model have to be concise and human-readable.

Therefore, current research has discovered several patterns (in the following called language patterns to distinguish them from design patterns) that enrich meta-models in different aspects, e.g., helping persons of different perspectives in the software development process (e.g., the software developer or the method engineer) to understand the meta-model easier [2] or improving their conciseness [3].

One of these language patterns with the above mentioned benefits is the powertype pattern [4], [5]. However, introducing a powertype pattern into an existing meta-model often results in several manual adaptions in other meta-levels for migrating models to the new meta-model. Hence, such a remodeling to powertype patterns can be a time-consuming and error-prone task [6]. Focusing on this problem, we present below an operator that introduces a powertype pattern into an existing meta-model. Simultaneously, the operator adapts corresponding models into a valid state.

Therefore, in the following section we are going to show the state of the art. Subsequently, we explain the powertype (pattern). After that, we will present an extension for this pattern: the extended powertype. In section V we present the Create-Powertype-For operator which introduces an (extended) powertype pattern into a meta-model. In the subsequent section we provide an example model on which we apply the operator. Finally, we give a conclusion and an outlook to our future work.

II. RELATED WORK

The presented work belongs to the research field of meta-model evolution. The Create-Powertype-For operator changes the (meta-) meta-model and migrates other (meta-) models to become valid to the new meta-model.

In the current research such an approach is called coupled evolution [7]. Since most of the work in this field considers merely two meta-levels the coupled evolution definition is limited to a model and a meta-model. As we do support more than two meta-levels in our modeling environment we extend this definition to arbitrary levels.

Meta-model evolution, in general, faces two main challenges. First, adaptations and changes performed on a meta-model need to be captured [8]. Second, evolving a meta-model might render models as instances of a meta-model invalid, e.g., when attributes are removed or a type within a meta-model is defined to be abstract within an evolution step. Hence, these invalid models have to be migrated which is called co-evolution [9].

According to the work of Herrmannsdorfer et al. [8], approaches for capturing meta model evolution can be categorized into three kinds: state based, change based and operation based approaches. State based approaches store two versions of a model and derive differences between those two versions after changes were actually performed (which is an implementation of the Model Management operator DIFF [10]). Contrariwise, change based approaches record differences at the moment they occur. Operation based approaches are a subclass of the change based approaches since changes on meta models are defined by means of transformation operators before they are actually performed. In today's systems often state based recording is chosen although it is not as powerful as the operation based approach [8]. Our presented approach belongs to the operation based approaches.

Practical application scenarios of the varied approaches can be found in the work of Gruschko et. al. [11] (state based), Aboulsamh et. al. [12] (change based) and Herrmannsdorfer et. al. [8], [13] (operation based).

Similar to the above presented work, Wachsmuth [14] and Herrmannsdorfer et. al. [13] provide an operation set that is used to evolve the meta-model explicitly, i.e., by means of well-defined transformations the user evolves the meta-model stepwise. In consequence, co-evolution can be performed without the need to handle ambiguity which is a challenge for state-based approaches [15].

Up to our knowledge, the current research in meta-model evolution mostly considers common meta-modeling concepts like classes, attributes and relations. Only some approaches (e.g., [13], [14], [16]) also analyze inheritance hierarchies for evolution and explain solution for handling co-evolution. However, there is no approach that considers other language pattern like the powertype pattern, deep instantiation or materialization [17].

Besides handling the evolution itself, handling co-evolution is another important topic in this field of research. To face this challenge, various approaches can be observed: matching of two meta models (see model management [18]), operation based co-evolution and manually specification of migration [15]. An Example for an operation based co-evolution can be found at the work of Wachsmuth [14], within the COPE System [19] and also within this paper.

III. THE POWERTYPE PATTERN

The powertype pattern is a language pattern used to describe that a concept A extends another concept Part (this is called the partitioned type) and at the same time this concept A is an instance of concept Pow (which is then called powertype).

A. *Example*

Below, there is an example of the powertype pattern that shows a simple meta-model (named M2) with two concepts: Tree and TreeKind. The concept Tree stands of course for a tree and TreeKind is a representation for a kind of a tree. Furthermore, a model (M1) is shown with only one concept Maple which stands for a correspondent real world object.

If one wants to model trees there are at least two different views of seeing a maple. On the one hand, this maple is a specialization of the class tree. On the other hand, maple partitions the set of trees because it is a kind of a tree. Hence, maple can be seen as a specialization of tree. To combine these two views, one can introduce the powertype pattern (Figure 1). Then, Tree is partitioned with TreeKind (the powertype) and Maple is an instance of TreeKind and together with that a specialization of Tree.

As a consequence, Maple has two different facets. The first one is the *type* facet that extends Tree and the second one is the *instance* facet, an instance of TreeKind.

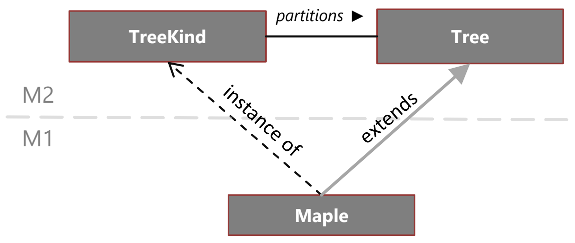


Figure 1. Example of a powertype pattern

Such a mixture of a class and an object is called clabject [20] or concept [21]. The specialization relationship is often not visualized within meta-model diagrams.

IV. THE EXTENDED POWERTYPE PATTERN

One rule of practice in modeling is that all attributes being common in all subclasses are added to the superclass [22]. Other attributes that do not belong to each of the subclasses are not declared in the superclass, in general. Instead, often new subclasses are created that stand between the super- and the subclasses in the inheritance hierarchy. As a consequence, a deep inheritance hierarchy could result which is often seen as bad design [23]. Furthermore, this approach leads to multiple inheritance which sometimes causes problems [24], [25] like the diamond of death.

To avoid this complex inheritance hierarchy, one can use the extended powertype pattern [26], [27]. This pattern enhances the powertype pattern with so called feature attributes.

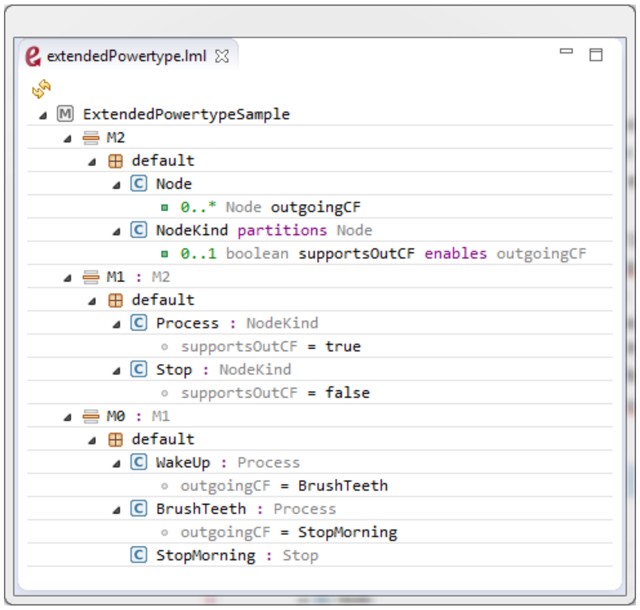
These boolean attributes are declared at the powertype with a link to an attribute of the partitioned type (the enabled attribute). Afterwards, one can decide for each instance of the powertype if an attribute of the partitioned type is inherited or not. If a feature attribute at an instance of the powertype is set to *true* the corresponding enabled attribute of the partitioned type is inherited. Needless to say that if a feature attribute has the value false no attribute is inherited.

Hence, all attributes of the sub-concepts can be collected in the partitioned type and for each sub-concept one can decide the set of attributes that are inherited.

A. *Example*

In Figure 2 a simple graph-based process modeling language with an extended powertype pattern is shown.

To visualize the complete meta-model stack we use a tree editor with syntax similar to object-oriented programming languages. The root of the tree is the whole meta-model stack. The children of that are the different meta-levels. The next higher meta-level which is instantiated by the current level is shown after the colon. Each level again contains at least one or more packages structuring the level. In a package lie concepts (clabjects) and these concepts can have attributes and/or assignments. Again, after the colon all instantiated concepts are listed. Other relations like extends or partitions are also shown together with the corresponding other concept.



```
extendedPowertype.lml
ExtendedPowertypeSample
  M2
    default
      Node
        0..* Node outgoingCF
      NodeKind partitions Node
        0..1 boolean supportsOutCF enables outgoingCF
  M1 : M2
    default
      Process : NodeKind
        supportsOutCF = true
      Stop : NodeKind
        supportsOutCF = false
  M0 : M1
    default
      WakeUp : Process
        outgoingCF = BrushTeeth
      BrushTeeth : Process
        outgoingCF = StopMorning
      StopMorning : Stop
```

Figure 2. Morning Process Example

Furthermore, the deep instantiation counter (also called deep instantiation potency) is displayed, if the value is greater than 1 (see section VI.A). Attributes have a cardinality (0..1, 0..*, 1..*) and an attribute type. Assignments consist of a corresponding attribute and a value for it.

In the meta-meta-model (M2), Nodes are connected with each other using outgoing control flows. This is done with the corresponding outgoingCF attribute at Node. Besides, an extended powertype (NodeKind) is modeled due to the fact that NodeKind has a partitions relation to Node. NodeKind again has a boolean feature attribute supportsOutCF enabling or disabling the outgoingCF attribute of Node.

At level M1, Process and Stop are instances of the powertype. Since a stop interface does not have any outgoing control flows the supportOutCF attribute is set to false whereas the Process attribute is set to true.

Level M0 contains a little model that describes a (spare) morning process. After waking up, the concerning person brushes his/her teeth and then stops the morning process. Since the feature attribute of Stop was set to false setting the value of outgoingCF in StopMorning would cause a validation error.

V. THE CREATE-POWERTYPE-FOR OPERATOR

In the following, we present an Evolution operator that introduces a powertype into an existing (meta-) model and simultaneously adapts the meta-model hierarchy to be valid again.

A. *Operator Process*

In Figure 3 the process of the Create-Powertype-For operator is shown. Therein all steps that need an input from the user are highlighted with black boxes. “The Move concept to upper level” and “the Add instantiation to powertype” steps are also highlighted as they are other complex evolution operators that will be presented below.

Initially, the operator is invoked with a source concept (e.g., chosen by the user). In the following, this concept is called Part as it will be the partitioned typed after the operator has finished. In the next step the operator collects all concepts that specialize Part. This set of concepts (in the following called SCs) is important because all members could potentially be an instance of the newly created powertype.

After that, the user decides which member of SCs will become an instance of the powertype and hence creates a subset of SCs (SubSCs). Then, for each member of SubSCs the specialization relation to concept Part is deleted. Afterwards, each concept of SubSCs is checked whether it is instantiated or not. If one concept is instantiated, concept

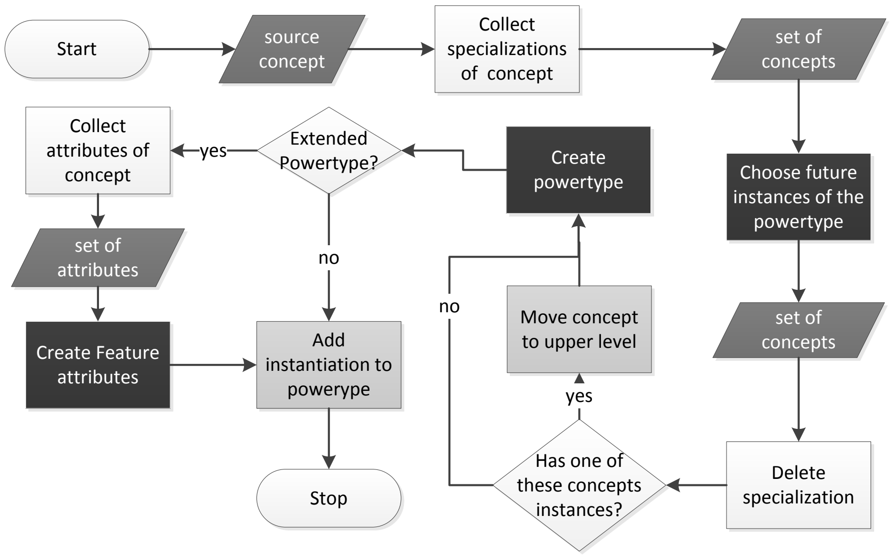

Start
source concept
Collect specializations of concept
set of concepts
Collect attributes of concept
yes
Extended Powertype?
Create powertype
Choose future instances of the powertype
set of attributes
no
no
Move concept to upper level
set of concepts
Create Feature attributes
Add instantiation to powerype
yes
Stop
Has one of these concepts instances?
Delete specialization

Figure 3. Create-Powertype-For operator process

Part and all related concepts (see section V.B) have to be moved to the upper level. Otherwise, it would not be possible to instantiate the instances of the powertype again. Of course, in case a modeling environment does not support several levels this step cannot be done and hence the instantiation has to be deleted.

Then, a new concept (the powertype) is created by the operator. The user specifies the properties of the concept like the name, whether the concept is abstract or final, its visibility, its instantiated concepts (optional), its extended concepts (optional) and its concretely used concepts (for instance specialization see [21]), also optional). The user also must specify whether the concept is an extended powertype or not. The concept Part will then be added to the set of partitioned concepts whereby the *partitions* relation of the powertype is created.

If the created powertype is an extended one the operator collects the attributes of the initially given concept Part. Then, the user chooses the attributes that will get a corresponding feature attribute which will be created in the powertype. Finally, each of the previously chosen concepts (SubSCs) will become an instance of the new powertype using the corresponding operator (see section V.C).

B. THE MOVE-CONCEPT-TO-UPPER-LEVEL Operator

The Move-Concept-To-Upper-Level operator moves, as the name indicates, a concept from a given level upon the next upper level. The process of the operator is shown in Figure 4.

1) Operator Process

The operator gets as input a concept that will be moved one meta-level up.

In the first step the operator tries to get the upper level and checks whether the level exists or not. If not a new level is created and the name of it has to be set. Then the operator changes the level of the given concept to the upper level.

Afterwards, the operator increments the deep instantiation counter of the given concept if the concept is instantiated.

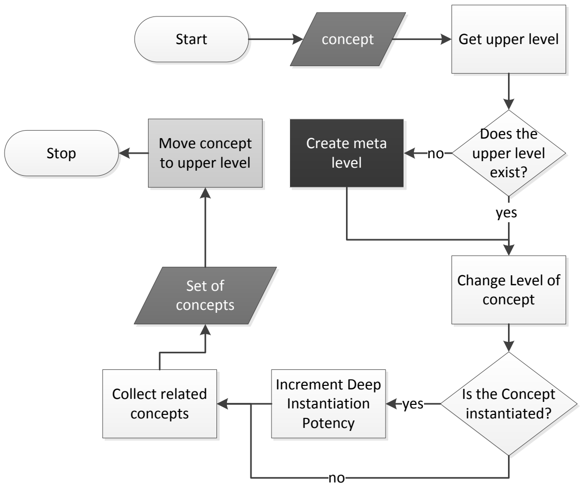


Figure 4. Move-Concept-To-Upper-Level operator process

Changing the value of the deep instantiation counter [28] causes that instances of the concept can instantiate the concept again although they are more than one (exactly two) meta-level lower. If deep instantiation is not supported other techniques like nested meta levels [29] may be used at this point.

For correct migration of the meta-model the operator has to invoke itself recursively on all related concepts. Thus, these concepts are collected in the next step. Related concepts are those concepts that stand in a relationship with the given concept (includes relationships like *extends* (for specialization), *partitions* (for powertype relation) or *concreteUseOf* (for instance specialization) [21]. Thereby, the operator has to detect cycles to avoid an endless loop.

C. The Add-Instantiation-To-Powertype operator

This operator adds an *instanceOf* relation from a given concept to a given powertype. In Figure 5 the process of the operator is presented.

1) Operator Process

Initially, the operator is invoked with a concept (the future instance) and a powertype. If the powertype is not an extended one merely the *instanceOf* relation between the concept and the powertype is created. Thereby, a constraint has to be considered. In case the instance of the powertype is already an instance of another concept this would end in multiple instantiation which breaks, e.g., strict meta modeling [30]. Thus, for such environments the operator has to delete one instantiation.

If the powertype is an extended powertype the operator has to provide a possibility to move the attributes from the given concept to the partitioned type. Therefore, the user has to choose all attributes of the concept that should be moved.

For each reference attributes (the attribute type is a concept) the operator has to check whether the attribute type is a specialization of the partitioned type.

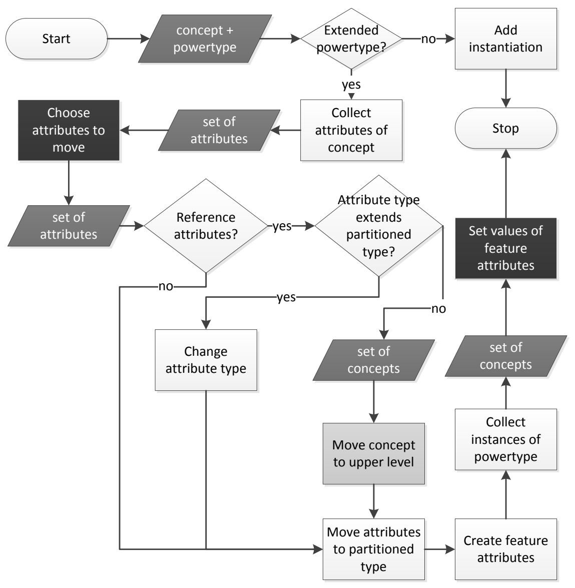

Start
concept + powertype
Extended powertype?
no
Add instantiation
yes
Collect attributes of concept
set of attributes
Choose attributes to move
Stop
set of attributes
Reference attributes?
yes
Attribute type extends partitioned type?
Set values of feature attributes
no
yes
no
Change attribute type
set of concepts
set of concepts
Move concept to upper level
Collect instances of powertype
Move attributes to partitioned type
Create feature attributes

Figure 5. Add-Instantiation-To-Powertype operator process

If so, the attribute type has to be changed to the partitioned type. Otherwise, the referenced concept has to be put one level up because relationships cannot cross levels. Hence the Move-Concept-To-Upper-Level operator is called.

Now the operator can move all selected attributes to the partitioned type.

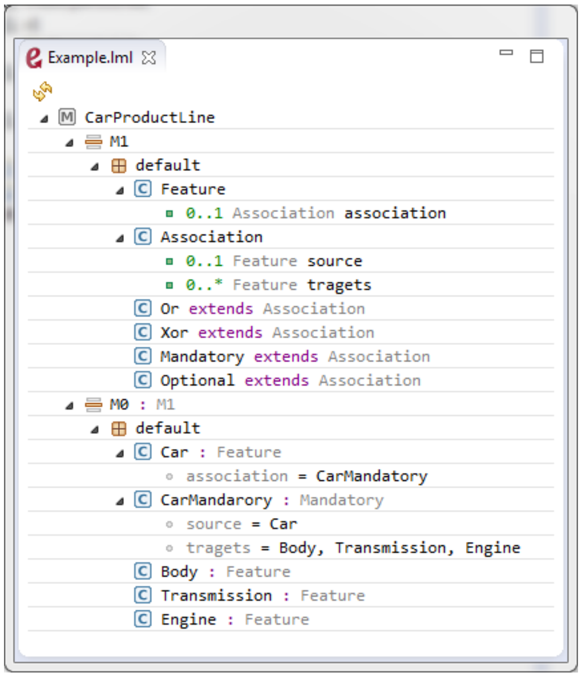
Afterwards, the operator creates corresponding feature attributes in the powertype for the moved attributes. Then, the operator collects all instances of the powertype and the user chooses those ones which should inherit the moved attributes.

Finally, the operator sets according to the user selection before the feature attribute values of all powertype instances. Of course, the value of the feature attributes for the given concept is set to true (since this concept declared the attributes before).

VI. Example

In this section we give an example for the application of the Create-Powertype-For operator. The example shows a simple feature model of a car product line inspired by [3]. This simple feature model gives the opportunity to model Features and link them with the help of Associations together.

Figure 6 shows the complete meta-model stack. Therein M1 is the meta-model for M0. On M1 there are two concepts: Feature and Association. Each Association element connects one Feature element as source and zero or more Feature elements as target. On the other side, Features can refer to zero or one Association. Thus, this relationship is bidirectional.



```
CarProductLine
  M1
    default
      Feature
        0..1 Association association
      Association
        0..1 Feature source
        0..* Feature tragets
      Or extends Association
      Xor extends Association
      Mandatory extends Association
      Optional extends Association
  M0 : M1
    default
      Car : Feature
        association = CarMandatory
      CarMandarory : Mandatory
        source = Car
        tragets = Body, Transmission, Engine
      Body : Feature
      Transmission : Feature
      Engine : Feature
```

Figure 6. Car product line model

Furthermore, the concept Association is specialized in form of the concepts Or, Xor, Mandatory and Optional. Xor and Or can be used to express that at least one of several target features have to be selected. Instances of Optional can set a target whereas instances of Mandatory have to select a target.

Based on M1, there is a model M0 that declares four features (Car, Body, Transmission and Engine) and one association (CarMandatory). These features are linked together with the association so that following constraint is expressed: A car must have a body, an engine and a transmission.

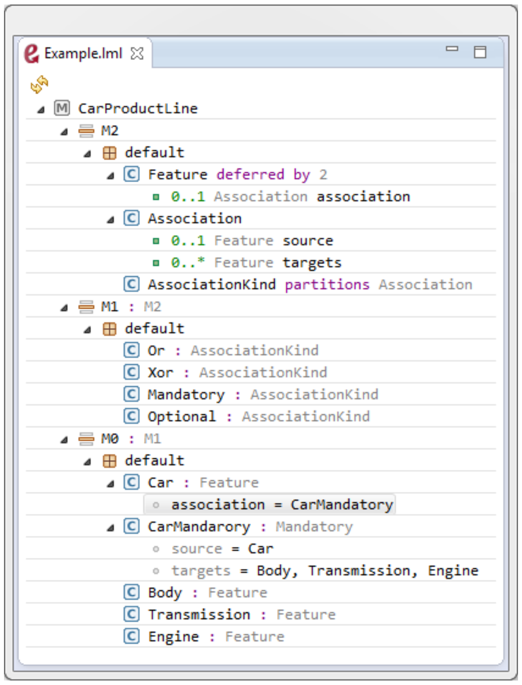
A. *Application of the operator*

Now, we apply the Create-Powertype-For operator to the above introduced model. The result is shown in Figure 7.

First, we select the concept Association and invoke the operator on it. The operator uses the given concept and collects all its specializations since these concepts are candidates for instances of the future powertype. The outcome of this step is a set of four concepts: Or, Xor, Mandatory and Optional.

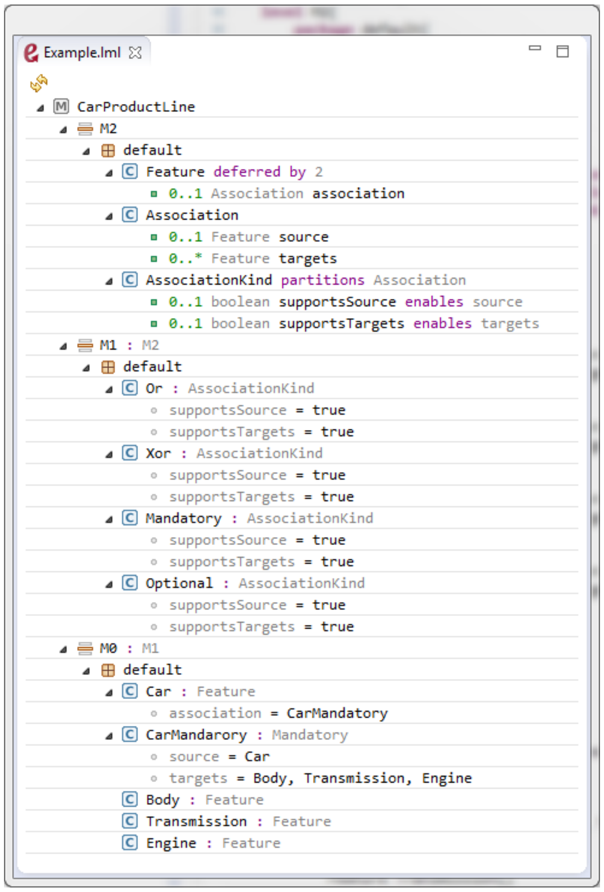
Afterwards, we have to review this set and tell the operator which concepts will become instances of the future powertype. In our example, we choose all of them. Then, the operator checks all selected concepts if they were instantiated before. This is true for Mandatory. Thus, the operator has to move the future powertype to the upper level and invokes the corresponding operator.

Hence, the concept Association is delivered to the Move-Concept-To-Upper-Level operator.



```
CarProductLine
  M2
    default
      Feature deferred by 2
        0..1 Association association
      Association
        0..1 Feature source
        0..* Feature targets
      AssociationKind partitions Association
  M1 : M2
    default
      Or : AssociationKind
      Xor : AssociationKind
      Mandatory : AssociationKind
      Optional : AssociationKind
  M0 : M1
    default
      Car : Feature
        association = CarMandatory
      CarMandarory : Mandatory
        source = Car
        targets = Body, Transmission, Engine
      Body : Feature
      Transmission : Feature
      Engine : Feature
```

Figure 7. The resulting car product line model after application of the operator



```
Example.lml
CarProductLine
  M2
    default
      Feature deferred by 2
          0..1 Association association
      Association
          0..1 Feature source
          0..* Feature targets
      AssociationKind partitions Association
          0..1 boolean supportsSource enables source
          0..1 boolean supportsTargets enables targets
  M1 : M2
    default
      Or : AssociationKind
          supportsSource = true
          supportsTargets = true
      Xor : AssociationKind
          supportsSource = true
          supportsTargets = true
      Mandatory : AssociationKind
          supportsSource = true
          supportsTargets = true
      Optional : AssociationKind
          supportsSource = true
          supportsTargets = true
  M0 : M1
    default
      Car : Feature
          association = CarMandatory
      CarMandarory : Mandatory
          source = Car
          targets = Body, Transmission, Engine
      Body : Feature
      Transmission : Feature
      Engine : Feature
```

Figure 8. Result of the operator with extended powertype

After that, the operator checks if an upper level exists which is false. That’s why it creates a new level that we name M2. Then the operator changes the level of Association to M2. After that, the specialization relation of Or, Xor, Mandatory and Optional is deleted. As Association is not instantiated, no deep instantiation counter has to be changed.

In the next step all related concepts are collected by the operator, which is only Feature (relationship to and from Association). Thus, the operator Move-Concept-To-Upper-Level is again called for Feature.

Because M2 already exists no meta-level has to be created. Subsequently, the meta-level of Feature is changed to M2 and the deep instantiation counter is incremented as it is instantiated in form of Car, Body, Transmission and Engine. Hence, the deep instantiation counter of Feature is now *2* (shown after the keyword *deferred by* in Figure 7). Since Feature has no related concepts because Association is already visited, the Move-Concept-To-Upper-Level operator terminates.

Afterwards, the Create-Concept-For-Powertype operator starts again with creating a new concept that we name AssociationKind and setting the partitions relation to Association.

If we decide to create a “simple” powertype Or, Xor, Mandatory and Optional just become instances of AssociationKind.

Otherwise, the operator collects for each concept (Or, Xor, Mandatory and Optional) all declared attributes. Since none of the concepts have attributes no user selection is needed and no reference attribute is part of the selection.

The operator continues with the creation of the feature attributes for targets and source (supportSource, supportTargets). Since Or, Xor, Mandatory and Optional were specializations of Association the feature attribute values for all concepts are set to true.

The result of creating an extended powertype is shown in Figure 8.

VII. Conclusion

Nowadays, meta-modeling is an often used approach for developing a domain specific language. Since these languages evolve during modeling of the domain of interest it is important to support this evolution to avoid manual migration of models.

Current research has discovered several patterns helping to improve the quality of (meta-) models [3]. Unfortunately, a remodeling of a meta-model to such a pattern is not supported today.

Facing this challenge, we presented in this article an operator that allows introducing a powertype pattern into an existing meta-model hierarchy considering migration of invalid models.

Currently, we have developed an Eclipse-based editor that supports several basic evolution operators like creating levels, packages, concepts and attributes. Furthermore some complex operators like the presented Create-Powertype-For, the Move-Concept-To-Upper-Level and the Add-Instantiation-To-Powertype operator are implemented as well.

In future work we will present complex evolution operators that support other language patterns like deep instantiation [28], materialization [17] or instance specialization [21]. Furthermore, we envision providing a preview of evolution operators similar to refactoring previews in modern IDEs. With the help of these previews, users can compare possible evolution steps.

References

[1] H. Cho, “A demonstration-based approach for designing domain-specific modeling languages,” *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 51–54, 2011.

[2] B. Henderson-Sellers and C. Gonzalez-Perez, “The rationale of powertype-based metamodelling to underpin software development methodologies,” *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling*, vol. 43, pp. 7–16, 2005.

[3] B. Neumayr, M. Schrefl, and B. Thalheim, “Modeling techniques for multi-level abstraction,” *The evolution of conceptual modeling*, pp. 68–92, 2011.

[4] C. Gonzalez-Perez and B. Henderson-Sellers, “A powertype-based metamodelling framework,” *Software and Systems Modeling*, vol. 5, pp. 72–90, 2006.

[5] J. Odell, “Power types,” *Journal of Object-Oriented Programming*, vol. 7(2), pp. 8–12, 1994.

[6] A. Demuth, “Cross-layer modeler: a tool for flexible multilevel modeling with consistency checking,” *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 452–455, 2011.

[7] M. Herrmannsdoerfer, S. Benz, and E. Juergens, “COPE-automating coupled evolution of metamodels and models,” *European Conference on Object-Oriented Programming*, pp. 52–76, 2009.

[8] M. Herrmannsdoerfer and M. Koegel, “Towards a generic operation recorder for model evolution,” *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pp. 76–81, 2010.

[9] D. Di Ruscio, “What is needed for managing co-evolution in MDE?,” *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, pp. 30–38, 2011.

[10] [P. A. Bernstein and S. Melnik, “Model Management 2.0: Manipulating Richer Mappings,” *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 1 – 12, 2007.

[11] B. Gruschko, D. S. Kolovos, and R. F. Paige, “Towards Synchronizing Models with Evolving Metamodels,” *Int. Workshop on Model-Driven Software Evolution held with the ECSMR*, 2007.

[12] M. A. Aboulsamh and J. Davies, “A Metamodel-Based Approach to Information Systems Evolution and Data Migration,” *Proceedings of the 2010 Fifth International Conference on Software Engineering Advances*, pp. 155–161, 2010.

[13] M. Herrmannsdoerfer, S. Vermolen, and G. Wachsmuth, “An extensive catalog of operators for the coupled evolution of metamodels and models,” *Software Language Engineering*, pp. 163–182, 2011.

[14] G. Wachsmuth, “Metamodel adaptation and model co-adaptation,” *European Conference on Object-Oriented Programming*, pp. 600–624, 2007.

[15] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack, “An analysis of approaches to model migration,” *In Proceedings of Models and Evolution (MoDSE-MCCM) Workshop*, pp. 6–15, 2009.

[16] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth, “Language evolution in practice: The history of GMF,” *Software Language Engineering*, pp. 3–22, 2010.

[17] M. Dahchour, A. Pirotte, and E. Zimányi, “Materialization and its metaclass implementation,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, no. 5, pp. 1078–1094, 2002.

[18] P. A. Bernstein, “Applying Model Management to Classical Meta Data Problems,” *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[19] M. Herrmannsdoerfer, S. Benz, and E. Juergens, “COPE: a language for the coupled evolution of metamodels and models,” *Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management.*, 2008.

[20] C. Atkinson and T. Kühne, “Meta-level independent modelling,” *International Workshop on Model Engineering at the 14th European Conference on Object-Oriented Programming*, vol. 12, p. 16, 2000.

[21] B. Volz, “Werkzeugunterstützung für methodenneutrale Metamodellierung,” Dissertation, Fakultät für Mathematik, Physik und Informatik, Universität Bayreuth, Bayreuth, 2011.

[22] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*. Prentice Hall International; Auflage: 6th edition. Global Edition., 2010, p. 1155.

[23] Microsoft, “When to Use Inheritance,” 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/27db6csx(v=vs.90).aspx. [Accessed: 26-Sep-2012].

[24] G. B. Singh, “Single versus multiple inheritance in object oriented programming,” *ACM SIGPLAN OOPS Messenger*, vol. 6, no. 1, pp. 30–39, Jan. 1995.

[25] B. Zengler, J. Hahn, C. Rupp, M. Jeckle, and S. Queins, *UML 2 glasklar: Praxiswissen für die UML-Modellierung und -Zertifizierung*. München - Wien: Hanser Fachbuchverlag, 2007, p. 559.

[26] S. Jablonski, B. Volz, and S. Dornstauder, “On the Implementation of Tools for Domain Specific Process Modelling,” *International Conference on the Evaluation of Novel Approaches to Software Engineering*, vol. 4, pp. 109–120, 2009.

[27] B. Volz and S. Dornstauder, “Implementing Domain Specific Process Modelling,” *Communications in Computer and Information Science*, vol. 69, pp. 120–132, 2010.

[28] T. Kühne and F. Steimann, “Tiefe charakterisierung,” *Modellierung 2004 : Proceedings zur Tagung*, pp. 109–120, 2004.

[29] C. Atkinson and T. Kühne, “The essence of multilevel metamodeling,” *«UML» 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pp. 19–33, 2001.

[30] C. Atkinson, “Supporting and applying the UML conceptual framework,” *Lecture Notes in Computer Science, UML’98*, pp. 21–36, 1999.

Developing Patterns Step-by-Step

A Pattern Generation Guidance for HCI Researchers

Alina Krischkowsky, Daniela Wurhofer, Nicole Perterer, Manfred Tscheligi
Christian Doppler Laboratory for "Contextual Interfaces"
HCI & Usability Unit, ICT&S Center, University of Salzburg
Salzburg, Austria
{firstname.lastname}@sbg.ac.at

Abstract—**Despite the broad application and usefulness of patterns in many application areas, there is still a lack of information on how patterns are generated. In this paper, we introduce a step-by-step guidance for generating patterns in the domain of human-computer interaction (HCI). With our guidance, we support researchers in structuring and presenting gathered empirical knowledge for special contexts (automotive, home, mobile). By means of the pattern generation guidance, we support researchers without previous expertise in pattern generation to make their insights available for other HCI researchers. Furthermore, our approach enhances the pattern generation process towards more traceable and comparable patterns.**

Keywords-Pattern Development; Guidance; CUX Patterns

I. INTRODUCTION AND MOTIVATION

Patterns have turned out to be a valuable tool for structuring and capturing knowledge in many application areas. For example, patterns are used in architecture, software engineering, interface design, pedagogics or ubiquitous computing (e.g., [1], [2], [3], [4], [5], [6]). In these contexts, patterns have been applied to document proven solutions for reoccurring problems in a specific domain. In the field of human-computer interaction (HCI), patterns have been used for documenting results from empirical studies (see e.g., [7][8]). As patterns allow to structure and collect study results in a systematic manner, the gained knowledge can be easily and quickly provided to other researchers and stakeholders.

Despite the broad application and apparent usefulness of patterns in general, there is still a lack of information on how patterns are generated. In fact, pattern generation seems to be more a matter of experience than of a structured process. In the pattern community, there is little literature available that tells more about the genesis of patterns. It still remains unclear how patterns actually come into existence and how patterns should be generated [7]. This makes it especially difficult for novices, who have no previous experience in developing patterns. In the area of HCI particularly, it turned out that patterns are a valuable tool to systematically structure and collect knowledge from empirical studies. There is a need for supporting researchers in developing patterns. Research in this area - i.e., how to come from empirical findings to patterns - is rare. There are some first attempts dealing with the generation of patterns; however, we did not find systematic descriptions of the generation process. Thus, the process of pattern generation can be considered as implicit knowledge – knowledge that is based on one's expertise or experience and often hard to articulate. This is not only difficult for researchers who are unfamiliar with pattern generation but also poses the problem of traceability and comparability. To the best of our knowledge, a systematic guidance for developing patterns based on empirical study results does not exist to date.

This prevalent deficiency encouraged us to develop a step-by-step online guidance for pattern generation in the area of HCI. In particular, we intended to support User Experience (UX) researchers in converting their gathered knowledge from empirical studies into patterns. The structural foundation for the intended patterns is the so-called Contextual User Experience (CUX) patterns format [9]. CUX patterns provide solutions on how to improve a user's experience when interacting with an interface in a specific application area. They are characterized by explicitly combining contextual aspects and UX.

The objective of this paper is to introduce a step-by-step UX pattern generation guidance. After motivating the need for systematic pattern generation guidance in Section I, we give an overview on patterns in HCI as well as on existing pattern development approaches in Section II. Based on a critical examination of existing pattern development approaches, we then present in Section III, our attempt to guide researchers in the pattern generation process. In Section IV, we provide insights on how we employed the suggested pattern guidance in a first pre-test in order to gather suggestions for further improvements and iterations. Based on related work done in this area, our proposed step-by-step guidance as well as the insights gathered within our first employment, we then, in Section V reflect and discuss our actions taken and provide an outlook for future work.

II. RELATED WORK

A. The Role of Patterns in HCI

In HCI, patterns have gained a lot of attention over the last years. Especially in interface or interaction design, there are numerous pattern collections (e.g., [10], [11], [4]). The concept of patterns in this area is known under different names; e.g., 'interaction (design) patterns', 'user interface (UI) patterns', 'usability patterns', 'web design patterns', 'workflow patterns' or, more general, 'HCI patterns'. Basically, these patterns provide solutions to commonly occurring usability problems in interaction and interface design. As the comprehensive use of patterns shows, patterns have been proven as a valuable tool for designing usable systems.

Apart from dealing with common user interface or interaction problems, patterns have been also used to document knowledge based on empirical studies. Martin et al. [7] developed patterns for cooperative interaction in order to organize, present, and represent material from ethnographic studies. In their work, patterns primarily served as a vehicle for presenting the major findings of previous studies and as communicative devices. In contrast to interface or interaction design patterns, this approach does not deal with solution-orientated patterns but rather with descriptive patterns in the tradition of Erickson [12]. UX research represents another specific domain of HCI, where patterns have been deployed to collect and structure knowledge based on empirical findings [8]. In the following section, we will introduce the idea of UX patterns in more detail as this represents the basis for the patterns generated with our pattern generation tool.

B. UX and Patterns

One major aim of HCI research is to create a positive experience while interacting with an interface [13], [14]. Research in this area is often referred to as "UX research". According to Alben [15], UX comprises all aspects of how people use an interactive product. This means, all the aspects of how people use an interactive product: the way it feels in their hands, how well they understand how it works, how they feel about it while they are using it, how well it serves their purposes, and how well it fits into the entire context in which they are using it [15]. Patterns have already been applied in the area of UX in order to structure and preserve knowledge. Blackwell & Fincher [16] suggest to adopt the idea of patterns and UX in the form of Patterns of User Experience (PUX). Such patterns should help HCI professionals to understand what kind of experiences people have with information structures.

Obrist et al. [8] applied UX patterns to document knowledge on UX in the domain of audio-visual networked applications (e.g., Facebook or YouTube). By means of UX patterns, they intended to capture the essence of a successful solution to a recurring UX related problem or demand of audio-visual networked applications. They developed a set of 30 UX patterns, summarizing the most important insights based on qualitative and quantitative studies. Thus, empirically grounded guidance on how to design for a better UX in audio-visual networked applications is provided. An extension of the UX patterns, are the so-called Contextual User Experience (CUX) patterns [9]. This approach relates contextual issues to UX and provides a pre-defined pattern structure to do so. Accordingly, patterns are "used to describe knowledge on how to influence the user's experience in a positive way by taking context parameters during the interaction with a system into account."

C. Approaches on Pattern Generation

As already stated before, there is not much literature on how to generate patterns. The process of looking for patterns is often considered as pattern mining [10]. However, pattern collections or languages are often introduced without explicitly stating how the patterns emerged. One of the few outlining their experiences and difficulties in developing patterns were Martin et al. [7], who deployed patterns for describing insights from ethnographic studies. They started pattern creation by looking, for instance, of repeated phenomena in ethnographic studies (re-examination of previous studies). Thereby, they included a reference to their context of production and seeking in their pattern descriptions. For them, the main purpose of patterns was to present major findings of previous studies and as communicative devices. For their creation of patterns, they began with looking for specific examples in a particular domain and then tried to expand the observed phenomena to other domains (similar but different examples).

In his work, Vlissides [17] describes seven habits for successful pattern writers. According to the author, reflection is the most important activity in pattern writing; this should be done by thinking about the developed applications and the problems and (if existing) successful solutions. This will provide the raw material of patterns. Additionally, similar applications or domains with similar problems can also give support for problems and solutions and, therefore, for the development of a pattern.

According to the author [17], the next step will be to choose a suitable and consistent structure for the patterns to be developed. Another important point in the development process of pattern is concreteness (compared to abstractness), meaning that concreteness improves the comprehensibility for people. It is also crucial to always keep the intents of patterns in mind, as well as the relationships between the patterns, so that the details of the patterns do not prevail. Moreover, effective presentation of patterns, including typesetting and writing style, is substantial for the quality of patterns. It is also important to mention that continuous iteration is essential, as patterns are never completed and always can be improved. Re-writing patterns, is therefore, a "normal" and necessary process. Finally, the collection and incorporation of feedback is another important step in the development of patterns. This includes the fact that patterns should be understandable to people, who had never been concerned with the problem before.

In order to develop patterns, Christopher Alexander defined the following questions to be answered within the process of mining [1].

1) "What, exactly, is this something? We must define some physical feature of the place, which seems worth abstracting.
2) What, exactly, is this something helping to make the place alive? Next, we must define the problem, or the field of forces, which this pattern brings into balance.
3) And when, or where, exactly, will this pattern work? Finally, we must define the range of contexts where this system of forces exists and where this pattern of physical relationship will indeed actually bring it into balance."

Alexander already pointed out the difficulty of generating patterns: "One very important question in writing patterns is, of how someone can recognize a pattern when coming across one? A simple but precise answer to this question is, that someone cannot always know."

According to Appleton [18], the best way to learn how to recognize and document useful patterns is by learning from others how they have done it well. It might be a good idea to read several books and articles that describe patterns and then try to see the necessary pattern elements and desirable qualities of a pattern. It has to be highlighted, that it is important to be introspective about everything to read. However, this is again about implicit knowledge and does not make the process of generating patterns explicit. There exist different criteria, which should be met by patterns in order to be considered as "good" patterns [18]. Further, there are defined processes a pattern should undergo [19]: (1) pattern mining, (2) pattern writing, (3) shepherding, (4) writers workshop, (5) author review, (6) pattern repository, (7) anonymous peer review, and (8) pattern book publication. However, there are no specific descriptions of each process in detail, and it is still not explicitly described how a first version of a pattern is developed.

D. *Pattern Generation as Implicit Knowledge*

According to May and Taylor [20], knowledge cannot always be handled directly. Knowledge emerges from a combination of expertise, perception, personal skill, and history, as well as constructive memory [21]. Indeed, some gathered information might be rather implicit and needs to be transferred into explicit knowledge. Thus, alternatives to capture and manage information in a way that supports making knowledge explicit and transferable are necessary. In order to capture and manage information to make knowledge more explicit, they suggest the use of patterns. Based on this, we see that the process of pattern generation can be considered as tacit or implicit knowledge – knowledge that is largely based on one's experience and hold by experts in patterns and pattern development [20].

It is quite common, that experts are unable to explain their methods or rationalize their actions. So far, the process of pattern generation is hardly explained in detail or described explicitly. In order to allow also non-experts to generate patterns, we aim to convert the implicit knowledge on pattern generation into an explicit one by applying our step-by-step pattern generation guidance. In this paper, we present a step-by-step pattern generation guidance whereby more details on our guidance are outlined below.

III. A Step-by-Step Pattern Generation Guidance

Within our research activities, the need for pattern guidance occurred within two national projects. These two projects focus on interface research. One project especially takes into account UX in the automotive context, whereas the other project deals with advanced interfaces in the home and mobile context. In both projects we aim to preserve knowledge gained on UX and contextual aspects based on empirical studies. Therefore, we used the CUX patterns approach [9], which has already proved its value for collecting and structuring knowledge on UX [8]. In comparison to other pattern structures, the CUX patterns approach seemed as most appropriate as it explicitly considers the relation of UX and contextual aspects. As this is an objective of our research, we chose the CUX pattern structure as a tool for preserving our knowledge.

Confronted with the fact that researchers involved in these projects were domain experts but mainly novices with regard to pattern generation, we systematically scanned literature in order to find advice for non-pattern experts on how to develop patterns. As already pointed out in our related work part, the main problem we identified was that the process of pattern generation represents implicit/tacit knowledge (i.e. expert knowledge). In order to make this knowledge also usable for non-experts, it has to be made explicit. According to our knowledge, this has not been done so far in a systematic manner. Thus, our step-by-step guidance on pattern generation represents a first step towards making the process of pattern generation explicit, allowing non-experts also to generate patterns and making the pattern generation process itself more traceable.

In the following section, we outline our developed pattern generation guidance in detail, reflecting on each step individually. Our major goal was to develop a systematic process that supports researchers to create patterns out of empirical study results. In order to ensure that the researchers have the possibility to iteratively as well as remotely, succeed with the pattern generation guidance, we set up an online survey (see for tool specific details [25]). Further, the use of this online survey tool supports storing data in a database, resulting in a pattern at the end.

After conducting intense desktop research, we developed an initial suggestion of a structured pattern generation guidance to support HCI researchers to create their own CUX pattern out of empirical study results. Our guidance is divided into five steps, all described in detail below.

A. *Step I: Introduction on Patterns*

Within this first step, the concept of CUX patterns [9] is introduced to the targeted HCI researchers (novices as well as experts). We split this first step into the following four sub-topics.

Outline of Major Goal (1): The major goal of our guidance tool was to support a structured pattern generation process in order to preserve and pass on knowledge from empirical study findings in the form of a pattern. This goal was first outlined in the guidance. Literally, it was defined as "Collecting and sharing UX and context related knowledge (based on empirical results either gathered within your own study or from literature) in a structured way by using a pattern form!" After outlining our major goal, we have included a visualization of an exemplary pattern in the guidance based on [8] with an UX focus on involvement and motivation (see in detail [26]). This should help the researchers to get a better impression about what CUX patterns are and how they are structured.

Characteristics of Patterns (2): After explaining our major goal and presenting an exemplary CUX pattern, the guidance provides an overview on the most important characteristics of patterns. Based on Vlissides [17], we defined the following eight aspects to be essential when creating a pattern especially for HCI researchers that are not experienced in developing patterns:

What you need to know about patterns!

① They capture expertise and knowledge to make it accessible to experts as well as non-experts.
② Their names collectively form a vocabulary that helps developers to communicate better.
③ They help people understand a system more quickly when it is documented with the patterns it uses.
④ Patterns represent a structured way to represent and communicate knowledge.
⑤ Using the same vocabulary avoids misunderstandings and ambiguities.
⑥ Patterns are abstract enough to make generalizations but as well detailed enough to provide practical solutions or suggestions.
⑦ Patterns are easy to understand (in a unified and human-readable format).
⑧ Patterns are short enough so that the knowledge can be accessed quickly.

Patterns within the Specific Project (3): Next, our guidance describes the purpose of CUX patterns and intended stakeholders within the targeted project. Furthermore, the definition of CUX patterns is provided to the researchers (see Section II.B).

Additional Information on Patterns (4): To provide further and more detailed information about patterns, we added some links that deal with software patterns, pattern languages (see [23]) as well as general information on patterns such as selected collections and publications (see [24]).

B. *Step II: Reflect and Select Your Key Finding(s)*

After giving the researchers a brief overview and input regarding patterns, the next step of the guidance focuses on the reflection and selection of relevant UX related results from empirical studies conducted by the researchers. This is one of the key steps within our process, since the process of reflection is the most important activity in pattern creation according to Vlissides [17]. We provide three text boxes within the survey, asking the researchers to select and summarize three findings. We have decided to provide three text boxes for the key findings in order to ensure that at least one of the key findings is appropriate for a pattern. These findings should be gathered within their studies and should represent insights on UX. Each key finding should be entered in one box.

In order to support the researchers in recognizing appropriate results to create a pattern with, we remind them within that part of the guidance that the main goal of generating the patterns is, to collect and share UX related results that have been gathered within their study in a structured way. After the researchers have entered three UX related key findings, we ask them in a next step to reflect on their chosen findings. Therefore, we ask the researchers to analyze their key findings according to the following aspects. These aspects ensure that they will be able to create a pattern and meet the predefined structure of our suggested CUX pattern based on their key findings:

Analyze according to the following checklist!

① My key finding addresses a/some specific UX factor(s).
② I can give a detailed and further description of my result(s).
③ I can describe the context from which my chosen key finding is extracted/gathered from
④ I can create design suggestions from these results.
⑤ I can underpin or visualize my design suggestions with examples.

After this checklist, the previously entered key findings are visualized again to ensure that the researchers can directly check their entered results and reflect on them according to the pointed out aspects outlined above. If the researchers were not able to identify any UX related key findings that satisfy those needs, we ask them to have a closer look at their results again in order to identify a potential UX related result there. By including this reflection cycle in the guidance, we want to ensure that the researchers proceed with an appropriate result to be able to create a pattern.

C. *Step III: Develop Your Pattern*

TABLE I. STRUCTURAL OVERVIEW OF OUR PATTERN

Instructions on Each Pattern Section		
#	***Section Name***	***Instruction on Each Section***
1	Name	*The name of the pattern should shortly describe the suggestions for design by the pattern (2-3 words would be best).*
2	UX Factor	*List the UX factor(s) addressed within your chosen key finding (potential UX factors listed in this section can be e.g. workload, trust, fun/enjoyment, stress...). Please underpin your chosen UX factor(s) with a definition.*
3	Key Finding	*As short as possible - the best would be to describe your key finding (either from an empirical study or findings that are reported in literature) in one sentence.*
4	Forces	*Should be a detailed description and further explanation of the result.*
5	Context	*Describe the detailed context in which your chosen key finding is extracted/gathered from.*
6	Suggestions for Design	*1) Can range from rather general suggestions to very concrete suggestions for a specific application area. 2) The design suggestions should be based on existing knowledge (e.g., state of the art solutions, empirical studies, guidelines, ...). 3) More than one suggestion are no problem but even better than only one. 4) There can also be a very general suggestions and more specific "sub-suggestions".*
7	Example	*Concrete examples underpinned by pictures, standard values etc. Examples should not provide suggestions (this is done in the suggestion part) but rather underpin and visualize the suggestion presented above.*
8	Keywords	*Describe main topics addressed by the pattern in order to enable structured search.*
9	Sources	*Origin of the pattern (e.g. literature, other pattern, studies or results)*

After the reflection cycle in Step 2, the researchers should be ready to actually create their own CUX pattern. Therefore, the pattern guidance again reminds them that their generated patterns should 1) capture expertise and knowledge, 2) be abstract enough to make generalizations, 3) but as well detailed enough to provide practical suggestions and 4) be easy to understand in a short and concrete manner. In order to support the researchers to meet these goals, we show them a predefined pattern structure visualized as a table. This provides an overview on the sections to be filled in. Further, this should encourage the researchers to keep our suggested structure. Our patterns are structured according to the nine sections (see section name) shown in TABLE I. The researchers are then asked to fill in the sections sequentially according to the given instructions below each section. In TABLE I. the instructions according to each section are outlined in more detail. After proceeding through each of these sections, the researchers have developed a first version of their CUX pattern based on their empirical results.

D. *Step IV: Final Check*

In order to ensure that the researchers have successfully conducted the process of pattern writing and met our predefined format of CUX patterns, we ask the researchers in a fourth step to have a final look at their pattern according to the following points:

Have a final check!

① Do a spell check by reading the pattern from the beginning to the end.
② Check if all sections are filled in appropriately.
③ Check if you have written everything in an easy and understandable way.
④ If you want to insert e.g. pictures, links in the "examples" or "sources" section, check if you have attached them.
⑤ Check if you are as concrete and short as possible.

To support the researchers in checking their generated pattern, we visualize the generated pattern below this checklist to make it easier for the researchers to assess if the generated pattern fulfills all the criteria listed above.

E. *Step V: Feedback*

In a last step, the guidance asks the domain specific researchers to provide feedback on the pattern generation process. Thus, we get insights on how to improve the guidance as a basis for further iteration. Therefore, we developed a short questionnaire (9 items) focusing on helpfulness, effort, difficulties, and concrete problems when using the pattern generation guidance.

IV. EMPLOYMENT OF THE PATTERN GENERATION GUIDANCE

In order to evaluate the guidance in terms of helpfulness, effort, task difficulty, and other issues occurring when applying our guidance, we have conducted a first pre-test with one HCI researcher who has had no previous experience in generating patterns. This initial evaluation cycle allowed us to get insights on the applicability and weaknesses of the guidance in practice. Based on these insights, we iterated our guidance especially for researchers with no previous experience in generating patterns. The pre-test was conducted in December 2012 and the researcher needed two hours to create his/her pattern out of gathered empirical results; we had expected that the generation process would generally take much longer.

TABLE II. represents the major issues evaluated during the pre-test, which have been clustered in four different problem categories. Apart from these more significant issues, the HCI researcher has also reported about minor issues, such as spelling mistakes and design issues of the survey. These minor issues are excluded from the reported problem categories below, since these issues are not relevant to the aim of the guidance and are easy to correct and do not need to be outlined in much more detail.

TABLE II. REPRESENTATION OF IDENTIFIED PROBLEMS

Overview of Identified Problems		
Problem Category	***Identified Problems***	***Reference to Guidance***
Sequence	(1) Sequence of sub-steps could be structured more clearly and intuitive	(1) Step 2
	(2) The chosen key-finding from Step 2 should appear in Step 3 again	(2) Step 3
	(3) Sequence of the sections (in the pattern structure) is not intuitive enough, since this is not the way how people create a pattern in their mind	(3) Step 3
Wording	(4) For Step 2 and 3 the wording "pattern" within the guidance is misleading since this would imply that the HCI Researcher already has to have the outcome in his/her mind	(4) Step 2 and 3
Repetitions/ unneeded information	(5) Detailed definition of Patterns is unnecessary, since the guidance should guide you how to create a pattern, therefore it´s not necessary to know a definition of what patterns are in our case	(5) Step 1/C
	(6) Graphical visualization of general pattern structure is shown again, which is unneeded information at that point	(6) Step 3
Text complexity	(7) The provided information in Step 3 (especially the reminder) is formulated too long and complex	(7) Step 3
	(8) The provided input in the introduction section is too long and not to the point	(8) Step 1

Overall, three problems were evaluated that relate to the sequence of different steps and sub-steps within the guidance. The pre-study participant reported that the sequence for the different sub-steps within Step 2 and 3, need to be iterated, in terms of making the sequences more intuitive and clear for the researchers. This means that for example, the reported problem number (3) 'the sequence of sections in the pattern structure' should be switched since the current sequence is not supporting the researchers, how they intuitively would generate a pattern in their mind. Therefore, we would suggest to change the sequence, in an iterated version of the guidance, as followed: 1) Pre-step, where the chosen key finding (from step 2) by the researchers is visualized again, 2) Forces, 3) Context, 4) Suggestions for Design, 5) Example, 6) Key Finding, 7) UX Factor, 8) Keywords, 9) Sources, 10) Examples. In order to check, if the sequence change of the sections makes it easier and more intuitive for the researchers to generate their pattern, we aim to test this changed order of sequences in another pre-test.

Besides this suggestion how to improve the guidance in terms of sequence changes in the pattern structure when generating the pattern, other areas for improvement could have been identified. Within Step 1 (introduction on patterns) especially, some parts of the guidance contain of unneeded/unnecessary information that is formulated rather complex at some parts. Our pre-study participant reported that some sections/parts (e.g. detailed definition of what CUX patterns are) do not have to be part of the guidance, since the guidance itself should direct the researchers in a way, that the generated pattern complies with our view on what CUX patterns for a structural representation of empirical study results are. Therefore, we aim to reduce such unneeded information in terms of deleting these sections and, therefore, reduce the information flow and complexity of the guidance. As another step to reduce the information overflow, we aim to formulate the different instructions/information shorter and especially formulating these parts more active in terms of "Researcher, do this… do that…" in order to provide short and concrete instructions for the researchers. This might reduce the potential of misunderstanding some parts.

Summarizing our first use of the guidance, we can state that when generating a pattern out of empirical study results, it is important to address an intuitive sequence of the different sections and steps, as well as to be concrete, short and to the point with the instructions provided in the guidance for the researchers.

V. CONCLUSION AND FUTURE WORK

In this paper, we introduced a step-by-step pattern generation guidance to support non-pattern experts in the generation of patterns and to support a traceable pattern generation process. Thus, knowledge gained within empirical studies is captured in the form of CUX patterns. We claim that our pattern generation process supports explicit knowledge regarding pattern development, and thus makes it easier to share and access knowledge with other HCI researchers. By applying our approach, we preserve and structure UX and context related knowledge gained within research projects and thus make knowledge accessible for researchers. Further, the researchers have to reflect on the quality of their empirical results which effects also the quality of the generated pattern. However, the presented approach also has some shortcomings. For example, the initially suggested sequence of the pattern generation was not intuitive, as turned out in the employment of the guidance. This issue will be addressed in an iterated version of the guidance. Another weakness of the presented approach is that patterns sometimes might not be the right format to represent empirical results. However, we believe that in most cases, patterns are able to summarize insights on contextual user experience.

We are aware that there is still space for improvement of our approach. For example, we would suggest that researchers could take different sources for their pattern. For instance, a researcher could take one key finding from his/her study, and the rest from reported literature. Using various resources (e.g., a published paper from the field, other domain-specific patterns, norms or guidelines) helps researchers to reflect about their relevant key finding, to combine it with relevant aspects and thereby, increases the quality of the generated pattern. According to Appleton [18], patterns should tell us a story, which captures the experiences they are trying to convey. In this context, we found that the right sequence of our pattern guidance is important. Especially Step 5, which asked to describe the context, should be represented earlier within the generation process. Further, we will support the researcher by listing six contextual categories to be selected: user context, system context, social context, temporal context, physical context, and the category “others”. By presenting concrete contextual categories, we assist the researchers to assign the key finding to the specific context. With such detailed information about the context, we get a deeper understanding of the relevant context.

In general, we consider the development of the pattern generation guidance as an iterative process, which demands continuous evaluation. In a first step, we plan to iterate the pattern sequence according to the drawbacks reported in the first employment of the guidance. In particular, we will change the different sections and check if this order is more intuitive for the researchers. Another issue for future work is the extension of the guidance towards a validation of the created patterns [22]. This will be easy to realize as the patterns are already digitalized and can be provided to others for validation. Further, we will conduct an expert workshop on the suggested process in order to identify further improvement potentials. After iterating the guidance, we will employ the pattern generation guidance in the field by providing the guidance to HCI researchers, with differing experience in generating patterns, in order to collect patterns for the automotive, home, and mobile context.

ACKNOWLEDGMENT

The financial support by the Federal Ministry of Economy, Family and Youth and the National Foundation for Research, Technology and Development is gratefully acknowledged (Christian Doppler Laboratory for Contextual Interfaces). This research was also supported by the Austrian project AIR Advanced Interface Research funded by the Austrian Research Promotion Agency (FFG), the ZIT Center for Innovation and Technology and the province of Salzburg under contract number 825345.

REFERENCES

[1] C. Alexander, “The Timeless Way of Building,” New York: Oxford University Press, 1979.

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

[3] C. Crumlish, and E. Malone, “Designing Social Interfaces,” O’Reilly, 2009.

[4] J. Tidwell, “Designing Interfaces : Patterns for Effective Interaction Design,” O’Reilly Media, Inc., 2005.

[5] P. Kotzé, K. Renaud, and J. V. Biljon, “Don’t do this – pitfalls in using anti-patterns in teaching human-computer interaction principles,” Comput. Educ., Volume 50, Issue 3, 2008, pp. 979–1008.

[6] R. Reiners, I. Astrova, and A. Zimmermann, “Introducing new Pattern Language Concepts and an Extended Pattern Structure for Ubiquitous Computing Application Design Support,” Third International Conferences on Pervasive Patterns and Applications, 2011, pp. 61-66.

[7] D. Martin, T. Rodden, M. Rouncefield, I.Sommerville, and S. Viller, “Finding Patterns in the Fieldwork,” Proceedings of the Seventh European Conference on Computer-Supported Cooperative Work, Bonn, Germany, 2001, pp. 39-58.

[8] M. Obrist, D. Wurhofer, E.Beck, A. Karahasanovic, and M. Tscheligi, “User Experience (UX) Patterns for Audio-Visual Networked Applications: Inspirations for Design,” Proceedings of the NordiCHI, Reykjavik, Iceland, 2010, pp. 343-352.

[9] M. Obrist, D. Wurhofer, E.Beck, and M. Tscheligi, “CUX Patterns Approach: Towards Contextual User Experience Patterns,” Proceedings of the Second International Conferences on Pervasive Patterns and Applications, Lisbon, Portugal, 2010, pp. 60-65.

[10] A. Dearden and J. Finlay, “Pattern Languages in HCI: A Critical Review,” HCI, Volume 21, 2006, pp. 49-102.

[11] J. Borchers, “A pattern approach to interaction design,” Chichester et al.: John Wiley & Sons, 2001.

[12] T. Erickson, “Lingua Francas for design: sacred places and pattern languages,“ Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques, Brooklyn, New York, 2000, pp. 357-368.

[13] M. Hassenzahl and N. Tractinsky “User experience - a research agenda,” Behaviour & Information Technology, Volume 25, Issue 2, 2006, pp. 91-97.

[14] M. A. Blythe, K. Overbeeke, A. F. Monk and P.C. Wright, “Funology: From Usability to Enjoyment,” Kluwer Academic Publishers, 2004.

[15] L. Alben, „Quality of experience: defining the criteria for effective interaction design,“ Interactions, Volume 3, Issue 3, 1996, pp. 11-15.

[16] A. F. Blackwell and S. Fincher, “Pux: patterns of user experience,” Interactions, Volume 17, Issue 2, 2010, pp. 27–31.

[17] J. Vlissides “Pattern Hatching: Design Patterns Applied,” IBM Thomas J. Watson Research Center, Addison-Wesley Professional, 1998.

[18] B. Appleton, “Patterns and Software: Essential Concepts and Terminology,” Object Magazine Online, Volume 3, Issue 2, 1997, pp. 20-25.

[19] S. Köhne, “Didaktischer Ansatz für das Blended Learning: Konzeption und Anwendung von Educational Patterns,” Dissertation, Universität Hohenheim, 2005.

[20] D. May and P. Taylor, “Knowledge Management with Patterns. Developing techniques to improve the process of converting information to knowledge,” Communications of the ACM - A game experience in every application, Volume 46, Issue 7, 2003, pp. 94-99.

[21] I. Nonaka, H. and Takeuchi “The Knowledge-Creating Company. How Japanese Companies Create the Dynamics of Innovation,“ Oxford University Press, 1995, pp. 3-19.

[22] D. Wurhofer, M. Obrist, E. Beck, and M. Tscheligi, “A Quality Criteria Framework for Pattern Validation,” International Journal On Advances in Software, Volume 3, Issue 1-2, 2010, pp. 252-264.

[23] The Hillside Group. Accessed March 2013. Retrieved from: http://hillside.net/index.php/patterns

[24] HCI Patterns. Accessed March 2013. Retrieved from: http://www.hcipatterns.org/

[25] LimeSurvey tool. Accessed March 2013. Retrieved from: http://www.limesurvey.org/

[26] Current Version of CUX Pattern Survey. Accessed March 2013. http://survey.uni-salzburg.at/index.php?sid=96811&newtest=Y&lang=en

An Analysis Model for Generative User Interface Patterns

Stefan Wendler, Detlef Streitferdt
Software Systems / Process Informatics Department
Ilmenau University of Technology
Ilmenau, Germany
{stefan.wendler, detlef.streitferdt}@tu-ilmenau.de

Abstract — **Graphical user interfaces (GUIs) are a crucial sub-system of current business information systems. They provide access for users to application kernel services in correspondence to business processes. As the processes and services change dynamically in our days, there is a strong need to adapt GUIs quickly to the changes. To enable both efficiency and usability during the adaptation, ongoing research has suggested to resort to model-based development processes, which employ patterns and their instantiation for specific GUI contexts. Those patterns are based on human computer interaction patterns and need to be formalized for their automated processing by generator tools. However, current research is still at the edge to express the concepts for such generative user interface patterns. The state of the art is not able to cover crucial factors of those patterns and misses a standardized format. Continuing our previous work on requirements for user interface patterns and their aspects, the aim of this paper is the development of an analysis model, which is able to express those needs in more detail using a semi-formal notation. With this step, a detailed description of generative user interface patterns is achieved, which can be the basis for the verification of current approaches of model- and pattern-based GUI development or even a deeper analysis.**

Keywords — user interface patterns; model-based user interface development; HCI patterns; graphical user interface.

I. Introduction

A. Motivation

Domain. Business information systems of our days are being maintained to upkeep or raise their effectiveness in supporting users carrying out operative tasks, which are demanded by the business processes of the respective company. Being a layer of a given business information system, the graphical user interface (GUI) is part of a value creation chain, as it enables the user to access functional, data and application flow related components of sub-systems located lower in hierarchy. Accordingly, the GUI allows the user to select and initiate functional behavior that processes data relevant to active tasks. As result, value is being created, which is meaningful to the sequence of the business process within the value creation chain. Due to systems are constantly matched closer to the set of tasks of the business processes and thus users are facing an increase in task scope and complexity, the need for well designed and adaptive GUIs has emerged.

GUI requirements. In this context, a user interface primarily is required to fulfill both the criteria of functionality and usability. On the one hand, a GUI has to reflect the current process definition and thus offer access to the respective activities in order to provide effective support for the user. On the other hand, for this support to be efficient, the non-functional requirement of usability, which embraces the suitability for the task and learning, as well as a high degree of self descriptiveness [1], plays an important role for testing and the acceptance for productive runs.

GUI adaptability. As business processes tend to change over time, the functional requirements based on them, such as use cases or task models, may change considerably, too. With those changes taking place, new requirements, having a significant impact on the GUI artifacts, are being introduced. Consequently, this part of the system has to conform to a high demand on adaptability besides the first release-specific requirements. Especially standard software systems, which offer a configurable core of functions to support business models, like applied in E-Commerce, see a distinctive demand for adaptive user interfaces [1]. Accordingly, a user interface of a business information system has to be based on a software architecture or development process, which facilitates the transition to new visual designs, dialogs, interaction designs and flows without causing significant costs in manpower and time.

Current limitations. Nowadays, the above mentioned requirements still cannot be accomplished fully by automation and generative development processes. On the one hand, available GUI-Generators can only cover certain stereotype parts of the user interface and may not lead to the desired quality in usability [1][3]. On the other hand, model-based development processes, which are able to generate more sophisticated user interfaces, also cannot support all variations on interaction and visual designs the changing business processes may demand for [4]. Finally, concepts that combine increased reuse and automation in user interface development and adaptation are being sought of.

User Interface Patterns. Together with other researchers [1][3][10][11][12][22], we believe that certain aspects of the GUI can be modeled independently in order to be composed and instantiated to their varying application contexts. As evolution and individualism in GUI implementations generally induce high efforts, an approach has to be followed, which enables a higher degree of reuse and hence allows for more common basic parts to be shared along components. For reuse, the basic layout of a dialog, its positioning of child elements and navigation flow as well as reoccurring user interface controls (UI-Controls) and their data type processing are to be mentioned as candidates for automated generation. In this context, the occurring variability needs to be expressed by new artifacts in the development process chain. The need for a systematic description of reusable GUI artifacts arose and initially has found its expression in human computer interaction (HCI) [5][6][7] or, more recently, in user interface patterns (UIPs) [8][9]. In this regard, UIPs describe the common aspects of a GUI system in an abstract way and the developers concretize them with the required parameter information suited for the context of their instantiation.

UIP conception. The existing work about UIPs applied in model-based development processes [10][11][12] has laid down conceptual basics and milestones towards experimental proofing. However, no dedicated pattern definition for user interface development [14] has emerged yet and so, the motivation of the PEICS 2010 workshop still stands [15].

Factor model. To progress towards a more detailed and complete UIP conception, we deeply elaborated requirements with impacts to architecture, formalization and configuration of UIPs in [4]. A process, which enables the instantiation of UIPs and their compositions to form a GUI of high usability and adaptability, altogether, needs such a clear basis of requirements. However, the factors we have modeled, reside on a descriptive level that is not favorable to be directly translated to notations or formats for generative UIPs.

B. Objectives

The impacts of our factor model in [4] have led us to the strategy, to specify an analysis model for the UIP aspects and their various impacts. This model serves as a medium to close the gap between descriptive requirements of the factor model and formal notations. With the analysis model, we are detailing the requirements even more and progress towards a semi-formal notation for their description. The model is intended to capture all essential aspects, properties and required parameters for context-specific application of UIPs. With this contribution, a first version of the analysis model is presented.

In this regard, we focus on the UIP representation and not its mapping or deployment process, since other researchers have advanced in that area, but still lack a proper UIP representation. This representation is elaborated here along with related work, criteria, examples and finally an analysis model. The following questions shall be answered by our model:

- What information is needed to describe a UIP as a generative pattern applicable as a GUI architecture design unit?
- What elements a formal language has to feature in order to permit the full specification of such UIPs?

C. Structure of the Paper

The following section provides an overview of the pattern type to be covered in this work. Additionally, we summarize the outcomes of our previous work on the examination of model-based development processes and requirements related to UIPs. In Section III, the problem statement is formulated. This is followed by our approach in Section IV. The elaboration of the analysis model is presented in Section V. The results of our work are reflected in Section VI, before we conclude and suggest future work in Section VII.

II. Related Work

A. Human Computer Interaction Patterns and User Interface Pattern Definition

To open the discussion of reusable GUI entities, aspects of patterns related to GUI development are now introduced. We approach the term “user interface pattern” (UIP), which will drive the further elaboration of related work. For this purpose, we ask what the origins for definitions of UIPs in the context of UI generation are.

HCI pattern ambitions. The early stages of patterns for user interfaces were determined by the goal to describe reoccurring problems and feasible solutions for GUI design offering high usability. Borchers [7] stated that human computer interaction (HCI) experts had a hard time communicating their feats in ensuring a good design of a systems GUI to software engineers. Thus the idea was born to express good usability via patterns as this was already a good practice for software architecture design. In this regard, Van Welie et al. [16] argued that patterns are more useful than guidelines for GUI design. In addition, they suggested the term pattern for user interface design along with criteria how to assess the impact on usability of each pattern.

Research into HCI patterns went on and culminated into pattern languages such as the one created by Tidwell [17]. Prior to this development, Mahemof and Johnston [5] outlined a hierarchy of patterns, what already implicated that there are complex relationships inside HCI pattern languages.

No unified pattern notation. Some years later, Hennipman et al. [18] claimed that available HCI pattern approaches could be improved as there are still obstacles for their efficient usage. Their analysis of relevant sources reveals major issues such as the missing guidelines how to formulate new HCI patterns, integrate them in tools and how to apply them. The request for a standard pattern specification template already was formulated by [16] and [7]. In this regard, Borchers mentions early sources adopting the pattern notion by Christopher Alexander. Thus, Fincher finally introduced PLML [19] in [20]. However, the issue of a missing standardized pattern format still persists [15], which eventually is detailed by Engel et al. [21]. Therein, they analyze the shortcomings of current HCI pattern catalogs and the intended standard notation of PLML.

UIP definition. Vanderdonckt and Simarro [22] separate two main representations of patterns based on the intended usage. Descriptive patterns serve a problem description and solution specification purpose. In contrast, generative patterns feature a machine readable format as they are to be processed by tools and in particular GUI generators.

B. Formal Languages for GUI Specification

Now, we ask if there are languages available that permit the formal specification of GUIs or even UIPs.

In our previous work [1][8], we already went into the possibilities to express UIPs with the means of mature GUI specification languages UIML [23] and UsiXML [25]. As these languages are focused on platform-independent full-fledged GUI specification and intended to be machine processed, some of their elements may be candidates to be included in a sophisticated UIP definition model. Both languages feature common elements to define the visual layout, interactive behavior, and content of a certain GUI part. For pattern-specific application UIML and UsiXML differ in their capabilities: UIML incorporates elements for template definition and a peer section, which decouples structures or UI-Controls within the layout from their technical counterparts. In contrast, UsiXML is based on a more complex approach, which defines a metamodel consisting of a model hierarchy and methodology [26]. The abstract and concrete user interface model may be of relevance for our objective.

C. *Influence Factor Model for User Interface Patterns*

Continuing on previous work, we progressed towards an elaborate influence factor model for UIPs, which is depicted in Figure 1. Motivated by missing standards and competing UIP notations inside modeling frameworks, this model was intended to establish an independent requirements view on the formalization and instantiation of generative UIPs: We took our examples and architecture experiments [1], as well as criteria, aspects and variability concerns [8], and refined them. The requirements stand close to the profile of current approaches in research. For details, [4] can be consulted.

The UIP definition to be sought after has to introduce a pattern conception, which is backed by a limited set of types, roles, relationships and collaborations among GUI related specifications and components. Because of the complex nature of both GUI architectures and specifications, a restriction and specialization of the entities to be involved in the development environments for pattern-based GUIs have to be set. Along with this restraint, the GUI specific kind of pattern still needs to be abstract in order to enable vast customization and instantiation to differing contexts. The major share of the patterns vigor has to be sourced from the similarity in structural (*view aspect*) and behavioral (*interaction* and *control aspect*) definition of new GUI entities.

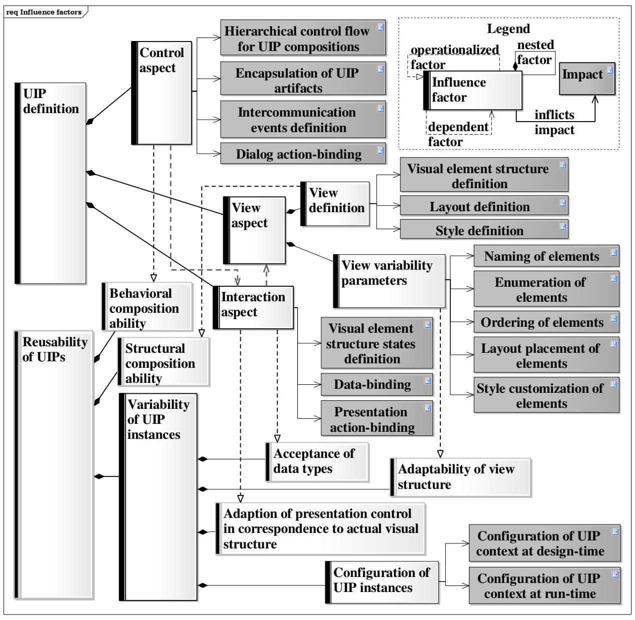

req Influence factors
UIP definition
Control aspect
Hierarchical control flow for UIP compositions
Encapsulation of UIP artifacts
Intercommunication events definition
Dialog action-binding
Legend
operationalized factor
nested factor
Influence factor
Impact
dependent factor
inflicts impact
View aspect
View definition
Visual element structure definition
Layout definition
Style definition
View variability parameters
Naming of elements
Enumeration of elements
Ordering of elements
Layout placement of elements
Style customization of elements
Behavioral composition ability
Interaction aspect
Visual element structure states definition
Data-binding
Presentation action-binding
Reusability of UIPs
Structural composition ability
Variability of UIP instances
Acceptance of data types
Adaptability of view structure
Adaption of presentation control in correspondence to actual visual structure
Configuration of UIP context at design-time
Configuration of UIP instances
Configuration of UIP context at run-time

Figure 1. Influence factor model for generative UIPs described in [4]

In other words, the pattern definition introduces certain quality aspects in GUI design, which can be altered quantitatively, when they are respectively complemented with necessary structure, layout and style details (*view variability parameters*) as well as combined with each other (*behavioral* and *structural composition abilities*). This commonality ensures that no longer specialized solutions or manually refined structures, which cannot be covered by mere UIP instantiation, are applied in the same GUI system architecture.

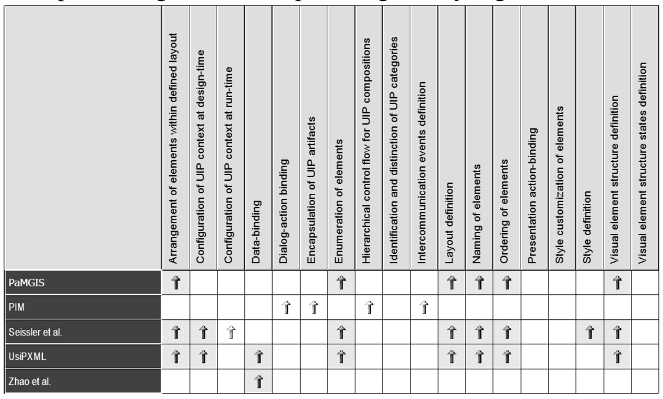
D. *Model-Based Development Processes involving User Interface Patterns*

The enhancement of model-based development by generative UIPs already found strong reception. In reference [4], we presented an overview and assessment of the approaches of Zhao et al. [1], PIM [27], UsiPXML [10], PaMGIS [11] and Seissler et al. [12]. For a summary, Table I TABLE Icompares the above described approaches.

TABLE I. COMPARISON OF APPROACHES FOR MODEL-BASED DEVELOPMENT EMPLOYING UIPs

	Approach			
	Zhao et al.	***UsiPXML***	***PaMGIS***	***Seissler et al.***
Pattern types	Task patterns based on [28], set of window and dialog navigation types	Task, dialog, layout and presentation	Task and presentation patterns, fine grained hierarchy based on	Task, dialog and presentation patterns
UIP formal-ization notation	Unknown	Enhanced UsiXML	Unknown, XML based, <automation> tag and DTD	Embedded UIML supplemented by parameter and XSLT enhancements
UIP config-uration	At design	At design	At design	At design and run-time
Process output	Target code	UsiXML, M6C	Target code	Augmented UIML to be interpreted

Not all of the factors' impacts were supported or inspired by the approaches. A summary of realized (arrow in a box) or inspired (single arrow) impacts is given by Figure 2.



	Arrangement of elements within defined layout	Configuration of UIP context at design-time	Configuration of UIP context at run-time	Data-binding	Dialog-action binding	Encapsulation of UIP artifacts	Enumeration of elements	Hierarchical control flow for UIP compositions	Identification and distinction of UIP categories	Intercommunication events definition	Layout definition	Naming of elements	Ordering of elements	Presentation action-binding	Style customization of elements	Style definition	Visual element structure definition	Visual element structure states definition
PaMGIS	⬆						⬆				⬆	⬆	⬆				⬆	
PIM					⇧	⇧		⇧		⇧								
Seissler et al.	⬆	⬆	⇧				⬆				⬆	⬆	⬆			⬆	⬆	
UsiPXML	⬆	⬆		⬆			⬆				⬆	⬆	⬆				⬆	
Zhao et al.				⬆														

Figure 2. Impacts covered by examined approaches

Since our valuation revealed that there were many open issues associated with the different approaches, we only considered the full and no partly or probable realization of an impact. Notably is that the *view aspect* was realized by the most recent approaches. In contrast, the *interaction aspect* was only considered for *Data-binding*. Moreover, the *control aspect* was not realized by any approach, but inspired by PIM. Lastly, the *Configuration of UIP instances* was restricted to design-time only, but already inspired by Seissler et al. in reference [13].

III. Problem Statement

A. UIP Definition

Descriptive UIPs. From our observations concerning descriptive UIPs, we learned that they are well-understood as specification elements and supported by the HCI community. Nevertheless, the research into descriptive HCI patterns has not yet converged towards a standardization for the structure and organization of UIPs [15][21].

Generative UIPs. Generative UIPs may be classified as software patterns and as those they need a formal notation, and thus, are seldom encountered.

From our point of view, the past work on HCI patterns is concentrated on the descriptive form. As there is no unified approach in specification and usage of descriptive HCI patterns, they can hardly be used to source and abstract common elements of a generative representation. First and foremost, descriptive UIP sources may be a useful resource to assemble dialogs that may act as representative examples for a certain system or domain. On that basis, requirements or criteria for UIP formalization can be inductively obtained. Partly, we revert to this approach and sketch some example UIP instances in Section IV.B.

As a consequence, there is a large gap concerning the detailed definition of generative UIPs. Thus, a format for UIPs has to be found that is at least able to express most impacts of view and interaction aspect. Filling the gap with their own UIP concepts and notations, the model-based approaches of Section II.D are converging concerning the view aspect, but failed to convey all UIP impacts.

B. Formal GUI Languages and model-based Development

Enhancements. As there is still no dedicated language for UIP formalization, developers have to revert to existing GUI specification languages like UIML or UsiXML, which will be referred as XML languages in the following. As a result, two factions among the model-based approaches arose, one using UsiXML and the other applying UIML. Both languages need enhancements to express UIP related variability. Accordingly, the approaches incorporated their own parameter and configuration concepts. In sum, they all failed to publish enhancements that empower the specification languages regarding the interaction and control aspects. Currently, the notations are restricted to the view aspect mostly.

Generation of XML specifications. The XML languages have been developed to offer a platform-independent specification of GUI systems. In this context, they have been based on a metamodel that is somewhat similar to common universal object-oriented programming languages, which cannot handle aspects or traits and thus are incapable of expressing patterns in their abstract form. The XML languages clearly fail in the fulfillment of the reusability, variability and composition ability criteria [8].

However, applying the XML languages for their original purpose, apart from pattern definition, may play out their strengths. Accordingly, developers could use them for concrete GUI definition and final rendering to the desired platform. To integrate UIPs in this procedure, a generation of XML language code could be a possible solution to overcome the inabilities as proposed in [1]. This idea was already followed either by generation of UsiXML [10] or the interpretation of UIML [12]. The XML code would hold the already instantiated UIPs or the required information for rendering. The benefit would be the possibility to use existing tools for the XML languages. In addition, a more important merit would exist in obtaining a concrete user interface level (CUI) specification [26], and thus, the ability to be independent from platform specifics.

In any case, a new language or extensions for the XML languages are to be sought after. Whether UIPs are being defined concretely in XML or the latter is generated, the XML languages will be a fundamental part of this solution. Consequently, the new language must facilitate the expression of UIP instances in rich XML language specifications. For that purpose, a unified UIP-model has to be established, which truly holds all information for the definition of generative UIPs and parameters for their transformation to UIP instances or instance compositions forming a concrete GUI model.

IV. Our Approach

A. Strategy

As mentioned in the objectives, the impacts in reference [4] resulted in the strategy to develop an analysis model, which is aimed at further detailing the UIP aspects. We develop a structural model that is biased towards an implementation of a dedicated UIP language.

Motivation of an analysis model. Some requirements such as *interaction* and *control aspects* are cross-cutting concerns and are really hard to achieve for pattern formalization. Thus, more planning and rationale is required before we can consider the development of a dedicated language. We follow the way of traditional modeling of requirements and ease their transformation to design with an analysis model. The model is intended to express the domain terms and concepts with a structure.

With a structural and more detailed model, the tracing of the influence factor impacts to potential solutions is better possible than with the pure influence factor model presented by Figure 1. In the factor model, there exist no separated entities that are modeled with their attributes and relationships to reflect a possible solution approach.

Assessment of recent approaches. Although we pointed out the factor support and issues we could so far discover as result of our assessment of other available approaches in reference [4], we also concluded that more details on examples and the applied notation have to be revealed in order to refine the assessment. By developing an analysis model, we seek to overcome the lack of detail and rationale on the design of notations suitable for UIPs. The notation to be used for modeling is the UML 2.0 class model.

Why do we propose a semi-formal model? For a technical architecture design or a generative process for formal UIPs to be verified, a wide range of requirements emerging from the initial criteria have to be taken into account, which cannot comprehensively modeled on a formal basis. In contrast to other researchers directly pushing towards a formalization of UIPs, we think this intermediate step is necessary and helpful. In our opinion, a semi-formal model is more useful to the developer than a formal model in first place, hence the mental conception about full scale generative UIPs has to be inspired first. The understanding of these complex patterns, their aspects and element relationships is the primary goal that should not be hindered by formal media, which cannot be imagined easily. A semi-formal model enables a better understanding than a grammar, since it may visualize concepts, their structure and relations depending on the chosen notation.

In sum, the model has to satisfy the information needs of the developers first, before they can think of how to employ the available formalization options or even GUI XML languages to express the requirements residing inside the model. Primarily, the model has to capture requirements in way that is easily understandable for human-beings.

Why do we apply the UML 2.0 class model? The UML class model lies in between the descriptive nature of the factor impacts and a formal notation. In this regard, a class model is already inclined towards a formal implementation. This is the case for class models serving as a design model for object oriented programming languages. In analogy, our analysis model may lead to a design for new language elements for the definition of generative UIPs. The language to be sought after also should rely on a structural paradigm, since the GUI implementations form a structure as well.

Moreover, a class model already proved useful for the expression of design patterns. The paradigm employed allows us to model abstract data types, their common attributes as well as their cardinalities and relationships. As the model entities all reside on an abstract level and do not describe already instantiated objects, the class model proves to be suitable for our task. More precisely, the UIP concepts can be modeled from a point of view where the abstraction and instantiation are separated. The class model forces the developer to express his solutions by abstractions that concentrate the commonalities of later instantiated objects. As we seek to express UIPs that feature reusable GUI solution aspects, a class model may provide a proper notation.

With the class model, we will be probing the modeling of required information for UIPs. Currently, developing a particular language or focussing on a certain architecture experiment seems to be too specific. In contrast, we investigate how the information of UIPs and their configuration can be established in general. To sort out possible options, trace factor impacts on more detailed granularity and map them to the final solution, the analysis class model may prove as a valuable asset. Finally, we may draft a coupling between a UIP, its configuration and GUI architecture or at least mandatory prerequisites.

B. User Interface Pattern Examples

By reason that we do not want to claim being able to establish a UIP analysis model applicable for each domain, we stick to business information systems as mentioned in the introduction. More precisely, as stated in Section III.A, we rely on common dialogs for E-Commerce applications as a basis. In fact, we subsequently derive the analysis model by focusing both on the factor model in Figure 1 and the following example dialogs.

Simple search. For an easy example, we start with a dialog that has the “Search Box” [28] pattern instantiated. The simple search illustrated in Figure 3 is mainly composed by a single panel (*ContentPanel*), which defines a GridBagLayout as seen in the upper part of Figure 3. The UI-Controls are fixed and aligned in respective fashion. For variability, only the concrete object data types need to be bound to the combobox and textfield. In fact, this kind of UIP is mainly invariant.

Advanced search. The next example shall be more complicated and thus, demand for every aspect described within the factor model. We decided for an “Advanced Search” [28] pattern, which alters its visuals and interaction options depending on user input.

Our example, depicted in Figure 4, mainly consists of two panels for layout definition as shown on the upper half. The panel *RootPanel* defines a GridBagLayout consisting of three cells (grey borders). Located in the center of this container, the *SearchCriteriaPanel* defines a layout of several rows each containing on cell (solid black borders). Additionally, the latter may grow or shrink in height to accommodate or discard search criteria lines to fit inside the container. Lastly, the *SearchCriterionPanel* (dashed borders) defines a layout appropriate for individual search criterions.

The usage of this dialog is as follows: Firstly, the user selects an object to be searched from the “Type of Object” combobox. Secondly, he chooses an attribute from the combobox inside the *SearchCriteriaPanel*. Accordingly, the UIP dynamically has to instantiate new sub-UIPs, which resemble the single search criteria rows. For each datatype, a pre-defined UIP, which is similar in shape to the *SearchCriterionPanel*, is assumed to be available. In the example, the datatypes String, price, and week are considered. With the buttons on the right hand side, the user may add or drop new search criteria rows and so the view aspect will change.

The variability is limited to the object types and their attributes to be searched with this UIP. Controller related aspects have to be adapted based on the UIP definition.

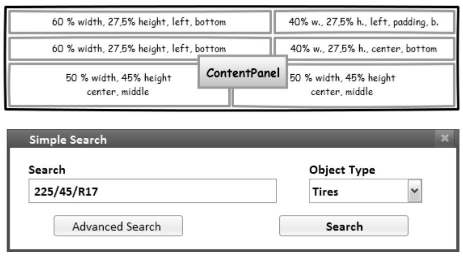

60 % width, 27,5% height, left, bottom
40% w., 27,5% h., left, padding, b.
60 % width, 27,5% height, left, bottom
40% w., 27,5% h., center, bottom
50 % width, 45% height center, middle
ContentPanel
50 % width, 45% height center, middle
Simple Search
Search
225/45/R17
Object Type
Tires
Advanced Search
Search

Figure 3. Simple search UIP example layout and dialog

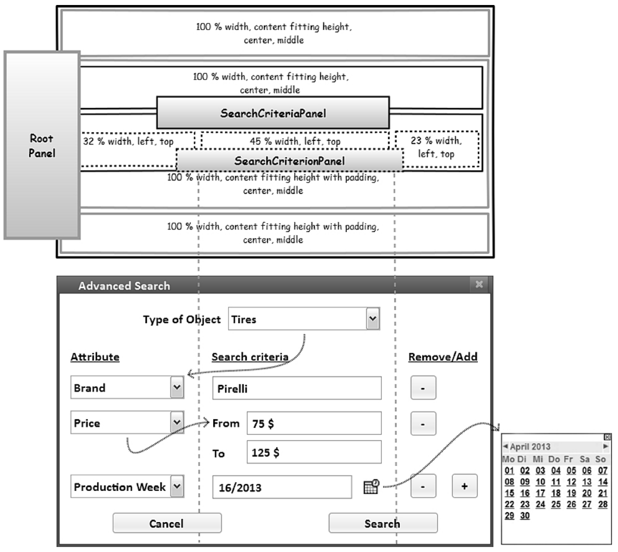


Figure 4. Advanced search UIP example layout and dialog

V. The Analysis Model

In this section, we develop the proposed analysis model. At first, we review each UIP aspect and its associated impacts in order to elaborate the decisions in design of the new model. Afterwards we present the structure of the model and finally apply the model to both examples introduced in Section IV.B. The terms in *italics* refer to respective analysis model elements.

A. *Analysis Model Bias*

On principle, there are two options on how to bias the model. Firstly, the model could be biased towards the software architecture and thus employ proven design patterns in its structures. This option would be rather suitable for generators and the further automated processing of the model, but it would be tedious to translate it back to the UIP requirements for the developers. In addition, the formal XML GUI languages (Section II.B) were not designed to accommodate architectural knowledge.

Secondly, the analysis model may be biased towards requirements and thus acting as a traditional analysis model, which captures and visualizes requirements. This option would be rather easy for the developers to understand, but would be costly to be translated to formal languages and generators. However, the translation to the XML languages is only a theoretical aspect, since generative UIPs cannot be expressed by their facilities as discussed in Section III.B. Eventually, we decided for the latter option.

B. *General Rationale*

Separation of definition and instances. A fundamental decision was the separation of elements or features that may be available in a UIP definition and the several element instances that may appear in a particular UIP instance for a certain context. In other words, we divided the UIP analysis model into two parts. One part holds the definition and reoccurring features (class names in black). The other part allows the description of instance information (class names in white).

UIP configuration. Following this approach, the main class *UserInterfacePattern* takes part in relationships that mostly focus on definition purposes, but also is connected to *UIPConfiguration*, which enables the description of particular UIP instances of the respective kind. The information used for pattern definition purposes will be covered in the following sub-sections. The configuration of UIP instances further branches into *Defaults* and *Parameters*. Both classes resemble containers that hold the *UIControl* instances, which are declared as *UIControlConfigurations*, for a particular UIP instance.

The *Defaults* are intended to omit stereotype configurations of default *UIControl* instances, which commonly appear in most contexts and shall not be defined redundantly. Concerning the example dialogs, the basic or invariant *UIControls* needed for user understanding and interaction like the labels, textfield and combobox of the simple search should be defined as *Defaults*, as there is hardly variability. This way, already established configurations may partly be reused among individual UIP instances. That means a UIP may contain pre-configured elements and parameters to avoid repetition. Later on, this facility will become useful for the dynamic adaptation of a UIP instance at run-time.

Both *UIPConfiguration* and *UIControlConfiguration* are primarily used for the "Configuration at design-time" impact and thus contain the declarations a developer may define in interaction with an "instantiation wizard" [10]. The configuration of *UserInterfacePatterns* and *UIControls* has to be separated, since both offer different sets of attributes, and more important, impact the GUI on different levels of abstraction or scope.

C. *View Aspect Design*

View definition. To begin with "View definition", this factor defines the *UIControls* or *UserInterfacePatterns* to be generally contained in a UIP specification unit as visual components. Both resemble a *ViewStructureElement*, which has a unique *ID* as identifier inside the pattern used by *UIPConfiguration* and *UIControlConfiguration* to reference the respective element. *UIControl* is a classifier for the various visual components or widgets a GUI framework may possess as types.

A UIP is always composed of a *ViewStructureElement* set and thus may build a varying hierarchical structure of those graphical elements. However, *ViewStructure* only holds each *ViewStructureElement* to be available to build instances once. The resulting element structure of a particular UIP instance is not described by *ViewStructure*. Instead, this is the responsibility of the configuration classes. The *ViewStructure* only defines what elements are generally available for the particular UIP. Based on that decision, the *ViewStructureElements* later may be exchanged without altering the already defined configurations.

For each *UIControl* of the resulting *ViewStructure*, style and general layout have to be defined. The style impact is not detailed here, since we have not came to a result in this regard and focused on the other impacts. For the sake of uniform views and maintaining corporate design, style information may be governed globally and locally by each individual *UIPConfiguration*. In addition, there may be constraints for each element, which determine its allowed minimum and maximum occurrences.

Layout rationale. With respect to "Layout definition" impact, we ask if there is a need for dedicated layout-patterns or if the distinction between primitives (*UIControl*) and composites (*UserInterfacePattern*) is adequate.

Referring to UsiPXML [11], layout patterns can be defined separated from presentation patterns. How they are integrated at various stages in the hierarchy, and more important, how they can be handled dynamically at run-time, remains an open issue, as there were no detailed examples for pattern composition and specification code given.

In addition, it is arguable whether a layout is assigned separately to a paralleled UIP composition or if each UIP models layout partly but explicitly. Partly means that UIPs need to define attributes for the number of rows and columns of a grid, their relative width and height, as well as the alignment. A visual impression of the abstract layout definition expressed by UIPs is depicted in the upper parts of Figure 3 and Figure 4. We decided to model this information by UIPs, as for advanced search, the layout needs to be re-configured dynamically with respect to *SearchCriteriaPanel*. This panel may grow and shrink in row numbers.

Layout definition. Inspired by our examples, we treat the layout container as a UIP, and thus, a layout pattern is already merged inside. So, the above mentioned layout definition parameters have to be associated to each *ID* of a UIP-type class, since it is acting as a superior container. Consequently, the advanced search dialog consists of three UIPs designated as containers in Figure 4. Translated to GUI frameworks, this implicates that each UIP will be treated as a panel or even window frame with a certain *LayoutManager* attached. We reason our approach with the fact that every dialog at some stage needs layout containers and these are eventually to be mapped to peers in the GUI framework. The detailed parameters for layout, such as padding, orientation and size policies, may be governed globally.

View variability parameters. To configure parameters for an element of the *ViewStructure*, regardless of what type, the respective *ID* of that element is used as a reference.

The *UIControlConfigurations* assigned to UIPs influence the instantiated unit in a global way. So, for the *view aspect* the general layout of the instances *ViewStructure* is declared by *LayoutManager*, which decides on the actual grid, for example. This way, the layout and orientation of UIP instances may be altered, but have to be declared explicitly for each *UIPConfiguration*.

As the elements defined by a UIP are abstract, the reference to the *ID* acts in analogy to the class concept for object-orientation. In fact, the element occurrence is determined by the number of respective configurations. For the individual element instances, one or many *UIControlConfigurations* can be declared to specify their characteristics. More precisely, as *view aspect* parameters we arranged for *Name*, *Caption*, *and Order* inside a layout grid cell and *Style* of each element. Some of these parameters are even optional. With *LayoutPosition* the position of the element with respect to the declared *LayoutManager* can be defined.

D. Interaction Aspect Design

In the factor model, the *interaction aspect* was not separated between stereotype definitions and parameters, as this was done for *view aspect*. Finally, the main classes, which model the *interaction aspect*, resemble parameter types. Since the factors apart from the *view aspect* ones mostly resemble cross-cutting concerns, the resulting *interaction* and *control impacts* refer to the static and variable declaration of *view impact* elements as a basis. In detail, the *interaction* related *UIControlConfiguration* parameters comprise of *DataType*, *PresentationEvent* and *EventContext* as an additional child of the latter.

Coupling points. For a UIP definition to be integrated in a GUI architecture, there is the need to arrange for coupling points. These points allow the integration of automated generated code and manually defined UIP information. Potentially, these can be comprised of the following:

- Standard events (*control* - "intercommunication events definition", "dialog action-binding")
- Input and output data (*interaction* - "data binding")

The latter point may resemble GUI architecture models discovered in common MVC architectures. The mentioned coupling points are either evaluated (events) or processed by the dialog kernel or logic part of the dialog. It is not necessary for that component to know where data changes and events have originated from. So, these suggested coupling points may be a good starting point. Accordingly, events (*PresentationEvents* and *OutputActions*) and the "GUI Data Model" have been included in the analysis model.

Data-binding. The binding of a *UIControl* to certain data is accomplished by a *UIControlConfiguration* parameter. So, the *DataType* binds the elements to certain data structures. As *DomainDataTypes* may significantly differ from the types used by the GUI framework, the class *GUIProjection* is rather associated as the configured *DataType*. For the *DataType*, it can be configured if the data is to be displayed only (input) or if the user may conduct changes (output), which are finally applied to the GUI *Model* part. The *DataType* parameter also may be associated to *EventContext*, which configures the data to be submitted by a *PresentationEvent* of the respective element.

Besides the distinction between input and output, *Models* have to be provided as coupling points for both cases to obtain data for display. The application kernel has to provide a respective query to obtain *Entity* data and the GUI architecture has implement a certain *Model* to enable the presentation of the query with appropriate data types for *UIControls*, e.g., data conversion to strings or string lists. In this regard, aspects like the timing, refresh rate, lazy loading are no concern of the UIP definition and have to be implemented by the data sources or queries. The *Model* has to rely on the data source and is not responsible of those technical aspects. In contrast, the *Model* needs to provide the navigation inside data structures and the structuring of data for presentation purposes that may be altered from application and data layer designs in order to offer a suitable projection for human processing.

Currently, we are unsure how UIPs specific *Model* requirements are to be formalized. However, this information is essential for the coupling. In addition, it will provide useful for the checking of the validity of configuration and *view* variability of the UIP instance. Concerning the advanced search, there must be a *Model* available to provide object types and their attributes as well as another *Model* to accommodate the chosen search criteria as the dialog result.

Events rationale. For *PresentationEvents*, we enumerated some typical events implemented in GUI frameworks. To progress towards a unified solution for generative UIPs, we think that a standardization of events, *PresentationEvent* as well as *OutputAction*, and similar types is necessary. The integrative and strict type definitions of the GUI specification language UsiXML on CUI level [26] may be a valuable resource for that approach. Otherwise, both specification and tool processing would demand for niche solutions that are hardly manageable with respect to versions and dependencies. We wonder how UsiPXML [10] or the UIML UIP definition by Seissler et al. [12] are defined as a language to be integrated in tool environments, which are to handle the generic concept of their variables and assignments effectively. We have to wait for them to publish detailed language definitions and code examples.

Presentation action-binding. To bind an element to a certain *PresentationEvent* type, the desired event has to be included in the appropriate *UIControlConfiguration*. This event may be declared for various purposes concerning visual structure states as described below.

Visual element structure states definition. The first *interaction aspect* impact needs to be further detailed. Depending on the actual structure of the UIP, states that occur within the scope of the contained *UIControls* and states, which alter the view of embedded UIPs have to be covered. To trigger changes in state for both cases, only *UIControls* can be specified as sender of respective events.

***UIControl* states.** For changes in state, we consider the activation or deactivation as well as hiding and unhiding of single *UIControls* or sets of them. Those abstract events are to be translated to technical representations and their detailed implementation. For instance, a checkbox in a sub-form may deactivate the delivery address (if it is equivalent to billing address) or in another case, a collapsible panel may be collapsed. In our model, the *ViewStateAction* is defined as an abstract feature for a UIP. By the UIP specification, the possible actions are defined and associated to affected *UIControlConfigurations* and thus *UIControl* instances. Finally, for these actions triggering *PresentationEvents* can be associated.

Embedded UIP states. Since the possible states for composite UIPs cannot be enumerated or state machines finitely defined inside pattern specifications, we employ information, which describes the results of the state change, and thus, enables a generator to build appropriate state machines or comparative implementations.

The *ViewStructureAction* is designed to handle the change of visual states for UIPs. For the trigger, a respective *UIControlConfiguration* is needed, which is aimed at a certain *ID* to allocate the *UIControl* and the type of *PresentationEvent*. We considered the addition, replacement, or removal of UIP instances. This behavior is closely related to the <restructure> tag of UIML [24] and may be refined based on its semantics. However, for UIML these facilities can only be applied with already instantiated UIPs.

DynamicStructures are used for the addition, removal or replacement of *UserInterfacePattern* instances. They are selected on the basis of defined *Keys*, which enumerate certain *DataTypes* or *EventContext* data to assign pre-configured *UIPConfigurations* to the triggered *ViewStructureAction*. A *UIPConfiguration* may be used by more than on *Key*, which models a certain context situation. Concerning the advanced Search example, the *Model* holding the object and attributes lists must return values that match the specified keys. Each time a combobox is changed, the presentation event handling routine must query the *Model* for the selected objects attribute and its kind or type of representation. The query result will be embedded in the *EventContext*, which is matched to a *Key* value. So, the UIP and its *DynamicStructures* are based on a canonical representation of *DomainDataTypes*.

Moreover, the *ViewStructureActions* rely on pre-configured elements, which may only allow for variability concerning the *DataType*. They either rely on a self-reference (removal, replace) or additionally are associated to available elements of the *ViewStructure* (add, replace) via *DynamicStructures*.

However, this mechanism only makes sense for *UserInterfacePatterns*, which are specified by *Defaults* and always represented by default *IDs* present inside the *ViewStructure* of a UIP definition. In this way, the *DynamicStructures* will only affect default or invariant *UserInterfacePatterns* inside the given *ViewStructure*, hence it is not desirable to replace entire sets of UIP instances defined on behalf of the developer for a specific context. Thus, manually defined UIPs portions have to be separated from *DynamicStructures*.

Based on the considerations for *DynamicStructures*, we decided to associate *DataType* with *GUIProjection* rather than with *DomainDataType*. A reference to *DomainDataTypes* would have meant to define a *Key* and appropriate *UIPConfigurations* for each *DomainDataType.* Each change of types would have cascaded to each UIP relying on *DynamicStructures.* We believe that *GUIProjections* may be more stable than *DomainDataTypes* and even be shared among *DomainDataTypes.*

E. *Control Aspect Design*

Dialog action-binding. So far, we have not progressed to feasible results for most *control aspects*. Only the binding of *UIControls* to application actions has been included. Via the global *OutputAction* parameter declaration of a UIP, one can define what events of that kind are raised by the *UIControlConfigurations*. These can be bound to a certain *UIControl* only by a link with the *PresentationEvent*.

F. *Structure View on the Analysis Model*

The resulting analysis model is illustrated by Figure 5. The classes shaded in medium grey are related to the "view definition" factor. Configuration related classes are shaded in dark grey and feature a white caption. Most *interaction aspect* impacts are supported by the classes shaded in white.

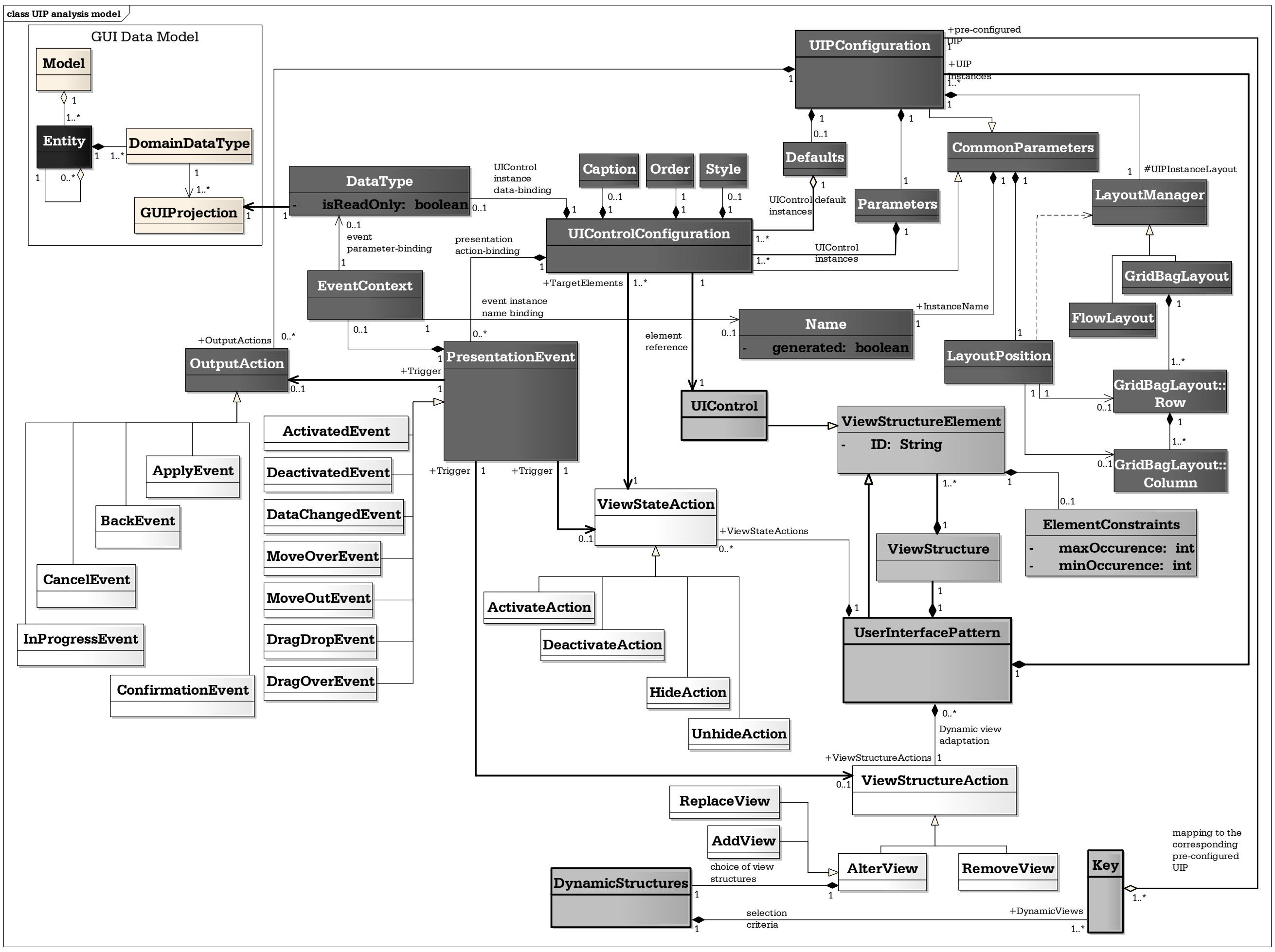


Figure 5. User interface pattern analysis model

VI. RESULTS AND DISCUSSION

Achievements. With the elaboration of our analysis model, we detailed most factor impacts of our previous work on requirements for generative UIP representations [14][4]. Accordingly, we proposed fine-grained structures, which are in closer proximity to real applicable pattern notations than pure requirements can be.

Judgment. The current state of the analysis model is quite imperfect. However, with this initial iteration we achieved a better understanding of the information needed to express UIPs and their instances. A more vivid impression on requirements, which we have modeled explicitly and are implicitly supported by current approaches employing UIPs for model-based development [4], has been gathered. Furthermore, the model already may be used to verify the capabilities of notations for generative UIPs.

The potential notation, generator tool-chain and especially the generated architecture, which may be derived in the future from the analysis model, most likely will be somewhat complex, but since they are solely intended for automated processing without manual interference, this is a trade-off for a step further to implement generative UIPs.

Again, we would like to invite other researchers to contribute either critical judgments or improvements for the presented analysis model or its requirements basis.

Unsolved control impacts. Currently, our model only supports *ViewStructures*, which consist of UIPs always being in close cooperation. Nested UIPs are not yet intended to be reused outside the specification or their super-ordinate UIP. Being aware of this barrier, we may need to define facilities such as pattern interfaces, as this was proposed by both UsiPXML [10] and Seissler et al. [12]. In this regard, the *OutputAction* may be refined to accommodate the events required for UIP inter-communication. Eventually, the *UIPConfiguration* may be supplemented by certain input types. In the end, the first three *control aspect* impacts remain unsolved for now.

Open issues. We are aware that our model needs further elaboration and especially verification. Further issues to be solved persist in the classification and delimitation of UIP specification units. The relationships among UIPs discussed by Engel, Herdin and Märtin [21] may be considered, too.

VII. CONCLUSION AND FUTURE WORK

By resuming our previous work on requirements towards a definition for generative UIPs, we drafted an analysis model for UIPs. Together with our factor model, it may be taken into consideration for the verification of other approaches mentioned and not mentioned here. With the progression towards an improved version of our analysis model, a more general applicable model-based UIP development process may be established in the future.

Future work. For future work, we see a refining and correcting iteration for the analysis model with regard to simplicity and completeness according to all impacts. In detail, we have to assess the mandatory and optional parameters on the basis of our listed examples. Furthermore, we will concentrate on the unsolved control aspect issues.

REFERENCES

[1] X. Zhao, Y. Zou, J. Hawkins, and B. Madapusi, “A Business-Process-Driven Approach for Generating E-commerce User Interfaces,” Proc. 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07), 2007, Springer LNCS 4735, pp. 256-270.

[2] S. Wendler, D. Ammon, T. Kikova, and I. Philippow, “Development of Graphical User Interfaces based on User Interface Patterns,” Proc. 4th International Conferences on Pervasive Patterns and Applications (PATTERNS 12), July 2012, Xpert Publishing Services, pp. 57-66.

[3] G. Meixner, “Past, Present, and Future of Model-Based User Interface Development,” in i-com, 10(3), November 2011, pp. 2-11.

[4] S. Wendler, D. Ammon, I. Philippow, and D. Streitferdt “A Factor Model capturing requirements for generative User Interface Patterns,” Proc. 5th International Conferences on Pervasive Patterns and Applications (PATTERNS 13), May 27 - June 1 2013, Xpert Publishing Services, in press.

[5] M. J. Mahemoff, L. J. Johnston, “Pattern Languages for Usability: An Investigation of Alternative Approaches,” Proc. 3rd Asian Pacific Computer and Human Interaction (APCHI 98), 1998, IEEE Computer Society, pp. 25-31.

[6] A. Dearden and J. Finlay, “Pattern Languages in HCI; A critical Review,” Human-Computer Interaction, 21(1), pp. 49-102.

[7] J. Borchers, “A Pattern Approach to Interaction Design,” Proc. Conference on Designing Interactive Systems (DIS 00), August 17-19 2000, ACM Press, pp. 369-378.

[8] D. Ammon, S. Wendler, T. Kikova, and I. Philippow, “Specification of Formalized Software Patterns for the Development of User Interfaces,” Proc. 7th International Conference on Software Engineering Advances (ICSEA 12), Nov. 2012, Xpert Publishing Services, pp. 296-303.

[9] A. Wolff, P. Forbrig, A. Dittmar, and D. Reichart, “Tool Support for an Evolutionary Design Process using Patterns,” Proc. Workshop: Multi-channel Adaptive Context-sensitive Systems (MAC 06), May 2006, pp. 71-80.

[10] F. Radeke and P. Forbrig, “Patterns in Task-based Modeling of User Interfaces,” Proc. 6th International Workshop on Task Models and Diagrams for Users Interface Design (TAMODIA 07), Nov. 2007, Springer LNCS 4849, pp. 184-197.

[11] J. Engel and C. Märtin, “PaMGIS: A Framework for Pattern-Based Modeling and Generation of Interactive Systems,” Proc. 13th International Conference on Human-Computer Interaction (HCII 09), July 2009, Springer LNCS 5610, pp. 826-835.

[12] M. Seissler, K. Breiner, and G. Meixner, “Towards Pattern-Driven Engineering of Run-Time Adaptive User Interfaces for Smart Production Environments,” Proc. 14th International Conference on Human-Computer Interaction (HCII 11), July 2011, Springer LNCS 6761, pp. 299-308.

[13] K. Breiner, G. Meixner, D. Rombach, M. Seissler, and D. Zühlke, “Efficient Generation of Ambient Intelligent User Interfaces,” Proc. 15th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (KES 11), Sept. 2011, Springer LNCS 6884, pp. 136-145.

[14] S. Wendler, I. Philippow, “Requirements for a Definition of generative User Interface Patterns,” Proc. 15th International Conference on Human-Computer Interaction (HCII 13), July 2013, in press.

[15] K. Breiner, M. Seissler, G. Meixner, P. Forbrig, A. Seffah, and K. Klöckner, “PEICS: Towards HCI Patterns into Engineering of Interactive Systems,” Proc. 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems (PEICS 10), June 2010, ACM, pp. 1-3.

[16] M. van Welie, G. C. van der Veer, A. Eliëns, “Patterns as Tools for User Interface Design,” C. Farenc, and J. Vanderdonckt (Eds.), “Tools for Working ith Guidelines,” 2000, Springer, London, pp. 313-324.

[17] J. Tidwell, “Designing Interfaces. Patterns for Effective Interaction Design,” 2005, O’Reilly.

[18] E. Hennipman, E. Oppelaar, and G. Veer, “Pattern Languages as Tool for Discount Usability Engineering,” Proc. 15th International Workshop Interactive Systems. Design, Specification, and Verification (DSV-IS 08), 16-18 July 2008, Springer LNCS 5136, pp. 108-120.

[19] S. Fincher, “PLML: Pattern Language Markup Language,” http://www.cs.kent.ac.uk/people/staff/saf/patterns/plml.html 24.03.2013

[20] S. Fincher, J. Finlay, S. Greene, L. Jones, P. Matchen, J. Thomas, and P. J. Molina, “Perspectives on HCI Patterns: Concepts and Tools (Introducing PLML),” Extended Abstracts of the 2003 Conference on Human Factors in Computing Systems (CHI 2003), ACM, 2003, pp. 1044-1045.

[21] J. Engel, C. Herdin, and C. Märtin, “Exploiting HCI Pattern Collections for User Interface Generation,” Proc. 4th International Conferences on Pervasive Patterns and Applications (PATTERNS 12), July 2012, Xpert Publishing Services, pp. 36-44.

[22] J. Vanderdonckt and F. M. Simarro, “Generative pattern-based Design of User Interfaces,” Proc. 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems (PEICS 10), June 2010, ACM, pp. 12-19.

[23] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, “UIML: An Appliance-Independent XML User Interface Language,” Computer Networks, 31(11-16), Proceedings of WWW8, 17 May 1999, pp. 1695-1708.

[24] UIML 4.0 specification, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml 12.04.2013.

[25] J. Vanderdonckt, Q. Limbourg, B. Michotte, L. Bouillon, D. Trevisan, and M. Florins, “UsiXML: a User Interface Description Language for Specifying multimodal User Interfaces,” Proc. W3C Workshop on Multimodal Interaction (WMI 04), 19-20 July 2004.

[26] J. Vanderdonckt, “A MDA-Compliant Environment for Developing User Interfaces of Information Systems,” O. Pastor, J. F. e Cunha (Eds.): Proc. 17th International Conference on Advanced Information Systems Engineering (CAiSE 2005), 2005, Springer LNCS 3520, pp. 16-31.

[27] F. Radeke, P. Forbrig, A. Seffah, and D. Sinnig, “PIM Tool: Support for Pattern-driven and Model-based UI development,” Proc. 5th International Workshop on Task Models and Diagrams for Users Interface Design (TAMODIA 06), Oct. 2006, Springer LNCS 4385, pp. 82-96.

[28] M. van Welie, “A pattern library for interaction design,” http://www.welie.com 27.04.2013.

[29] C. Märtin and A. Roski, “Structurally Supported Design of HCI Pattern Languages,” Proc. 12th International Conference on Human-Computer Interaction (HCII 07), July 2007, Springer LNCS 4550, pp. 1159-1167.