Plugin for Visual Paradigm for Generating and Intelligent Optimization of a Component Diagram Based on Attributes in the Class Diagram

Carsten Mueller[1], Anton Pohl[2*]

[1] University of Economics, Informatics Department, Prague, Czech Republic
[2] Baden-Wuerttemberg Cooperative State University, Mosbach, Germany

* Corresponding author. Email: pohl.anton89@gmail.com
Manuscript submitted January December 30, 2014; accepted February 16, 2015.
doi: 10.17706/jsw.10.3. 355-365

Abstract: In today's software development process (SDP) it is necessary to do much preparatory work to achieve a good quality of the software. Besides the requirements engineering the design process is very important. The designer can describe all aspects of the software to be built. Excellent designing skills and the question "What makes a 'good' software?" are essential to create a competent "cookbook". The plugin presented in this paper shall support designer using Visual Paradigm (VP) as a modeling-tool during their modeling process. Based on given attributes in the Class Diagram the plugin can generate and optimize intelligently a Component Diagram.

Key words: Software modeling, plugin for visual paradigm, software design, software component.

1. Introduction

We live in an information age where data processing becomes more important and difficult. To manage the growing amount of information the complexity of today's software systems increases steadily. Every human that takes pictures with a camera, every sensor that for example gauges the physical condition of an electronic part, and the communication over the internet produces data. The EMC Digital Universe study estimates the amount of data growing by 40% per year [1].

A complex system often requires a complex software to use all capabilities of the system. A waste of resources could be, e.g. to run a software that only processes data sequentially on a system that could process the data parallel. To develop such a compound software the process of software engineering has to be more rigorous to reduce risk and improve the quality [2].

There are many different paradigms and models of the software engineering process e.g. the waterfall model or the incremental model, the modular or the component-based software engineering. Every model has its advantages and disadvantages. Depending on the system and the use of the solution it is important to weigh the pros and contras of each model and programming paradigm.

The only crucial aspect of the SDP independent to the system or the use of the solution is the preparatory work. Every software development process should consider at least a requirement engineering phase and a software design phase to achieve a good quality of the software. Project manager should arrange enough time for these phases. The more preparatory work is done rigorously, the less failures should appear in the later phases of the

SDP. Less failures during the upcoming phases of the process reduce the expenditure that would be necessary to correct them.

The output of the design phase is a fundamental document used as basis for the team that implements the software [3]. The plugin presented in this paper shall especially support designer of a component-based software project.

2. Problem

Regarding the fundamental task of a designer it is easy to say that it is just to find the best solution of a problem and to describe how the solution has to be organized [4]. The designer produces a "cookbook" for the software problem. The design task by itself is even more complex than it might sound. There is a lot of information the designer has to consider to find the best solution for a software problem. Stakeholder have different interests on the project and may change their mind to any of the points they mentioned. The technical specification of the future system of the software might prevent the implementation team from implementing because some artifacts are not realizable with the given hardware.

Like mentioned before the complexity of today's software systems increases constantly. To use all its capabilities the software running on these systems often has to be intricate as well. This is what makes the task of a designer even more difficult. Complex software may lead to sophisticated design models.

Growing design complexity may cause loss of time and money during the project. The software has to be able to process the growing amount of data quicker. A simple search in a data base may take a long time. Besides the hardware the software model is an essential element. Good software models help the developers to create the software that is demanded. Automating the design process or parts of it requires to surmount several difficulties. The environment should supply a simple interface that provides many functionalities at the same time. A design model has to be easy to understand and accomplish every requirement to deliver an excellent blueprint. An additional problem is the optimal disposition of the diagram elements to promote comprehension of the diagram.

3. Software Component

There are several existing definitions of what a software component is. Reference [5] formed one possible definition that was formulated at the 1996 European Conference on Object-Oriented Programming as a result of the Workshop on Component-Oriented Programming:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

This definition mentions important characteristics. Treating a software component as a unit of a composition means, that is a part of a set of collaborating components. The software is composed of many fragments without being a monolithic application. If one of the pieces is outdated or defect, only that particular piece has to be replaced.

Contractually specified interfaces determine which services are provided by the software component. A component has to be able to be integrated into an application. Therefore, a software component has to contribute one or more interfaces. These interfaces are contractually specified and are not allowed to be changed. Changing one of these interfaces would lead to a change of every system that is using this component. This is why a

software component has to be deployed independently, the internal functions or routines should not change its interface or force other components to be changed.

Some software components use other software parts, e.g. different libraries, data bases, and other components. An explicit context dependency specifies what the component needs to be able to work correctly.

Software components aim for being reused as a subject to composition by third parties. People can compose their own applications by integrating different components into their systems.

4. Visual Paradigm and UML

Visual Paradigm is a modeling tool supporting UML 2, SysML, BPMN, and other design and management tasks. VP's products are used by known companies like Adobe, Apple Inc., Intel Corporation, and Oracle [6]. Users can create different diagrams e.g. class diagrams, component diagrams, and use case diagrams. It also provides an interface for developers to integrate their own features [7].

The basic structure of a plugin consists of an XML configuration-file that contains several information for example plugin id, required libraries, and custom actions [7]. One important part of the plugin is the action controller which performs all the action of the implemented functions. It can be represented through different buttons or menu items.

The programming language used for the implementation is Java. Through an instant generator .NET languages are supported as well. The API supports different IDEs e.g. Eclipse, IntelliJ, and Visual Studio [8]. Developers can import additionally several recommended libraries to run VP in debug mode. Reference [9] shows a tutorial for eclipse users to perform debugging on the plugin.

The API of VP is open source so users of the Community Edition can develop features in a non-commercial use as well. Reference [7] shows a detailed tutorial how a plugin can be implemented and deployed.

A component diagram assists a designer to model the architecture of software components and their dependencies within an object-oriented software system [10]. Typically used model elements in a component diagram are:

1) Component
2) Port
3) Interface
4) Connectors
5) Class

Chapter III of this paper describes and defines a software component. A component in UML is defined in the following way, as in [11]:

"A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these provided and required interfaces […]. One component may therefore be substituted by another only if the two are type conformant. Larger pieces of a system's functionality may be assembled by reusing components as parts in an encompassing component or assembly of components, and wiring them together.

A component is modeled throughout the development life cycle and successively refined into deployment and run-time. A component may be manifest by one or more artifacts, and in turn, that artifact may be deployed to its

execution environment. A deployment specification may define values that parameterize the component's execution."

This definition basically seizes the same characteristics of a software component but in a more detailed way, and additionally picks up some aspects of the development life cycle. A component representing a modular part of a system matches the characteristic of a software component being a unit of composition. Encapsulating its contents describes the component's property to be deployed independently. Changes within a component's internal structure do not affect other external parts. A replaceable manifestation within its environment imply a component being reused as a unit of composition within its environment or by third parties.

The second paragraph describes that a component's contractually specified interfaces and its explicit context dependencies in detail. The last part of this paragraph addresses the characteristic of modularity. A component can be implemented as a single module into an application or as a composition of collaborating components.

The first part of the last paragraph considers a component is modeled throughout the development life cycle, which means the task of modeling a component is only finished when the component is developed and ready to be reused on different systems. The last part of the last paragraph addresses the characteristics of modularity and independent deployment. This leads to two types of components in UML the basic components and the packaging components.

4.1. Basic Components

As described in [11], a basic component is a subtype of a class. It consists of attributes and operations and is able to have different connectors which are specified in a later part of this paper. As a subtype of a class it may have an internal structure. It has provided and required interfaces which may be realized directly and designated by a usage dependency or be provided and required by a public port.

4.2. Packaging Components

A packaging component defines its grouping and namespace aspects, which means all involved or related model elements are owned or imported explicitly [11].

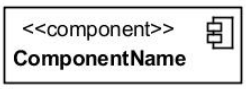

Fig. 1. Notation of a component.

The UML-notation of a component is displayed in Fig. 1. It is pictured as a rectangle with the keyword «component». The icon of the UML 1.4 notation of a component can be placed optionally in the upper right-hand corner. The icon is a rectangle with two smaller rectangles one upon the other overlapping the left border. In VP the notation of a component is equal to the notation shown in Fig. 1.

As mentioned earlier one type of a component's interface can be a port. The UML-definition of a port is, as in [11]:

"A port is a property of a classifier that specifies a distinct interaction point between the […] classifier and its environment or between the classifier and its internal parts. Ports are connected to properties of the classifier by connectors through which requests can be made to invoke the behavioral features of a classifier. A port may specify the services a classifier provides […] to its environment as well as the services that a classifier expects […]

of its environment."

Consequently, a port represents the interface between a component's internal structure and its external interfaces. To separate a provided and a required interface the ports have different connectors. The UML-notation is displayed in Fig. 2.

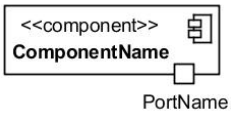

Fig. 2. Component with port

A port is pictured as a small square overlapping a border of a component. The position of the port can be defined as desired. It can be placed everywhere around the border of the component, but it should be placed next to its connected internal part.

The other type of a component's interface is a simple interface. An interface in UML specifies a contract which must be fulfilled by a classifier that realizes that interface [11]. An interface is described as a declaration that is not instantiable and has to be implemented by an instance of an instantiable classifier [11]. The UML-notation of an interface has two different appearances.

The first appearance is similar to a component, as shown in Fig. 3.

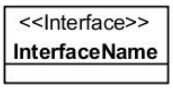

Fig. 3. Interface as classifier rectangle

It is displayed as a classifier rectangle with the keyword «interface». This notation is used by VP for example in a class diagram. The second appearance is the so-called ball-notation as shown in Fig. 4.

Fig. 4. Interface in ball-notation.

It is displayed as a circle or a ball with a label. Depending on its type, its appearance can vary. A provided interface has a different appearance than a required interface. The difference will be defined by the type and context of the used connector. This notation of an interface is used in VP e.g. in component diagrams. The UML-definition of a connector in a component context is, as in [11]:

"The connector concept is extended in the Components package to include contracts and notation.

A *delegation* connector is a connector that links the external contract of a component […] to the realization of that behavior. It represents the forwarding of events […]: a signal that arrives at a port that has a delegation connector to one or more parts or ports on parts will be passed on to those targets for handling.

An *assembly* connector is a connector between two or more parts or ports on parts that defines that one or more parts provide the services that other parts use."

This definition elucidates two significant types of connectors. The delegation connector passes a signal or massage between the connected elements potentially over multiple levels that are usually not contributed in all systems [11]. In UML a delegation connector is pictured as a solid line between two elements as displayed in Fig. 5.

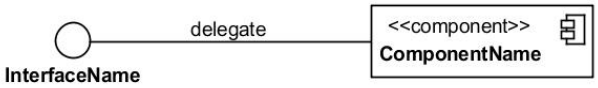

Fig. 5. Delegation connector of a realization

A provided interface connected through a delegation connector has the appearance of a lollipop. This notation implies that the component realizes its interface directly. The appearance of the interface changes to a socket when it is a required interface. This notation is defined as usage dependency, as pictured in Fig. 6.

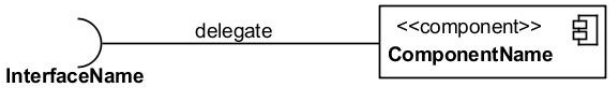

Fig. 6. Delegation connector of a usage dependency

An assembly connector connects multiple parts and passes services from providing to requiring elements. This enables components to be replaced by others that offer at least the same composition of services [11].

A simple assembly connector has the notation as displayed in Fig. 7.

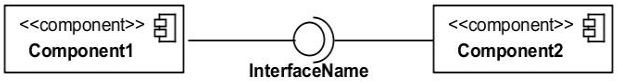

Fig. 7. Assembly connector between two components

This connector is pictured as a lollipop connecting to a socket. In this case Component1 offers services which are used by Component2. The assembly connector can be connected to ports of a component or directly to the component.

A class in UML is defined in the following way, as in [11]:

"A class describes a set of objects that share the same specifications of features, constraints, and semantics."

The features of a class are attributes and operations which may be inherited from interfaces through a realization as well. The UML-notation of a class is pictured in Fig. 8.

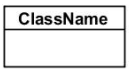

Fig. 8. Notation of a class

A class is represented through a classifier rectangle and usually is divided into three segments. The upper segment contains the name of the class. The middle segment comprises a list of attributes whereas the bottom segment holds a list of operations [11]. The presented notational system of the Object Management Group (OMG) is completely supported by Visual Paradigm.

5. Solution

The plugin requires a modeled class diagram. Based on the attributes of the diagram's elements a component diagram will be generated. The basic steps performed by the plugin are:

1) Scan the given class diagram and gather all required information
2) Simplify the stored data
3) Create a new component diagram

4) Create diagram elements and if required their internal structure

5.1. Class Diagram Model

The evaluation will be conducted by the example class diagram shown in Fig. 9. Every class has a universally unique identifier (uuid) and a single public operation. The classes ClassA and ClassB have a realization relationship to their interfaces. Between the classes ClassA and ClassB exists a 1: N-relationship. Between the classes ClassB and ClassC, and ClassC and ClassD exist 1:1-relationships. In Visual Paradigm the affiliation of a class to a component is stored in the tagged values. These can be displayed optionally within a diagram and is located in the upper segment of the classifier rectangle.

In this example ClassA and ClassB belong to Component1, and ClassC and ClassD belong to Component2.

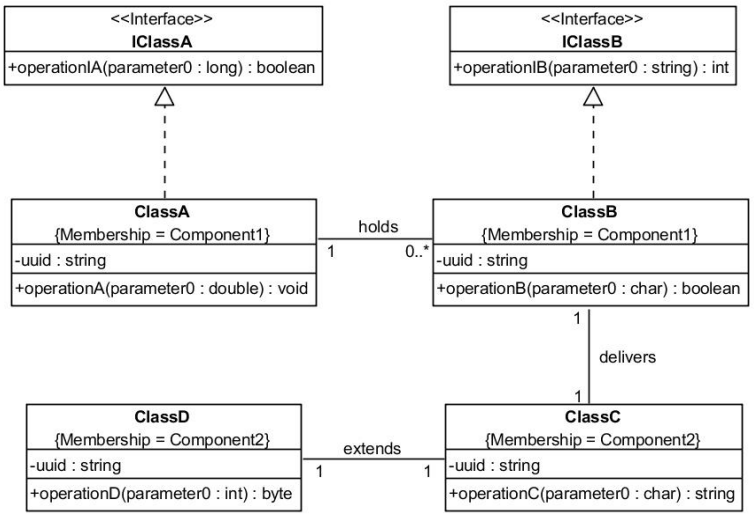

Fig. 9. Example of a class diagram.

5.2. Gather Information and Simplification

One of the most important interfaces of the API is the diagram manager. Through this instance of the application developers can communicate with the diagrams or its elements. A diagram manager is accessed by the following operation:

1) ApplicationManager.instance().getDiagramManager();
Source: VP API-call for instantiating a diagram manager

A diagram is an object of the type IDiagramUIModel within the diagram manager and is accessed in the following way:

1) Diagrammanager.getActiveDiagram();
Source: VP API-call to return the active diagram

Every element of a diagram is stored in an array of the type IDiagramElement and can be retrieved by the following operation:

1) Diagram.toDiagramElementArray();
Source: VP API-call to retrieve all diagram elements

Since the diagram elements are resided into an array, the next step is to iterate over them with a loop. To read information of an element the model element has to be extracted from the diagram element and is of the type IModelElement. This object contains several data and can be reached by calling the following procedure:

1) Diagramelement.getModelElement();
Source: VP API-call to retrieve an element's model

The most important attributes of a diagram element are the tagged values which are saved in a container and need to be extracted to evaluate the membership of the element, the operations in case of a single class belonging to a component, and its relationships to ensure the internal structure of a component to be complete and all inherited operations can be stored as well. Since operations usually can be found in a class element of a class diagram, a class element is required. To get a class element it is sufficient to cast a model element as IClass. Relationships are saved inside of an iterator which are retrieved by a while loop.

The following methods are required to access these attributes:

1) modelElement.getTaggedValues().getTaggedValueByIndex(int index);
2) classElement.getOperationByIndex(int index);
3) classElement.toRelationshipIterator();

Source: VP API-calls to retrieve tagged values, operations and a relationship iterator

To get access to inherited operations the type of an element's relationship needs to be checked. To ensure that the element at the end of a relationship node is an interface it is possible to verify the stereotype of the element.

During the information gathering process the plugin checks the relationships between the class diagram elements. Classes with a 1:1-relationship that belong to the same component will be consolidated and simplified. Their public operations will be merged. Two situations may appear:

1) The combined classes have one or more 1:N-relationships to classes that belong to the same component
2) The combined classes have one or more relationships of any multiplicity to classes that belong to different components

The first situation is resolved by a new component and an interface which includes the merged public operations in its specification. This structure will be integrated into the internal structure of the superior component. Within the internal structure the classes of 1:N-relationships will be connected to the new component-interface construction.

The second situation is simply resolved by a new component-interface structure that contains all public operations of its affiliated classes.

5.3. Create Component Diagram

The next step is to create an empty component diagram to generate the required elements. Simultaneously it can be implemented as an object to change its characteristics, for example the name, and the background color. The component diagram is created concretely by calling the following operation:

1) Diagrammanager.createDiagram(String diagramType);
Source: VP API-call to create a diagram

The parameter is a string that contains the diagram type. To create a component diagram "ComponentDiagram" has to be passed. As a result of this operation a new entry is created in the diagram navigator as shown in Fig. 10.

Fig. 10. Component diagram entry in the diagram navigator

A new component diagram was created but not yet opened. This should be done at the end of the creation

process. Otherwise, it could occur that some elements are not displayed as they are supposed to be.

The function to open the diagram and see the results of the diagram generation executed as follows:

1) diagramManager.openDiagram(IDiagramUIModel diagram);

Source: VP API-call to open a diagram

The instance of the component diagram is passed as parameter to the function. The opened diagram will be automatically displayed in the foreground.

5.4. Create Diagram Elements

Based on the collected data elements now can be created. Depending on the context different sources are required. For the superior elements it is sufficient to create instances of a model element and a diagram element. During the modeling process a variation of the diagram element, the IShapeUIModel provides more important modeling features for example the "fitSize" that automatically scales the size of a diagram element to its minimum. It can be accessed by a cast while creating a diagram element. Before doing so it is recommended to create its model element. This can be done in the following way:

1) IModelElementFactory.instance().createModelType();
2) (IShapeUIModel) diagramManager.createDiagramElement(
3) IDiagramUIModel diagram, IModelElement modelElement);

Source: VP API-call for creating a model element and a diagram element of the type IShapeUIModel

For the model element every type has its own method. To create a class "createClass()" is called, to create a component "createComponent()" is called. As parameters for the diagram element the destination diagram is required and the model element. Alternatively the shape type can be passed as string. In this case no properties can be defined prior. The element obtains a default name.

Created diagram elements are connected by calling the following operation:

1) diagramManager.createConnector(
2) IDiagramUIModel diagram,
3) IModelElement connectorModel,
4) IDiagramElement fromDiagramElement,
5) IDiagramElement toDiagramElement,
6) Point[] connectorPoints);

Source: VP API-call for creating a connector between two diagram elements

This operation requires five parameters. The first parameter is the destination diagram where the connector will be created. The second parameter is the model element of the connector. Alternatively the shape type can be passed as string. The third and the fourth parameter are the diagram elements from which to which the connector shall be created. The last parameter requires an array of coordinates where the end nodes of the connector should be placed. To use the default location the value "null" can be passed.

The elements of the internal structure of a component may face the situation of combined classes having relationships to other classes that belong to the same component. These classes need to be created as a linked copy from the class diagram into the component.

To handle this problem two objects are required. The first object is the model element from the class diagram. The second object is a newly created diagram element in the component diagram of the same type. With these two objects a link can be generated.

1) diagramElement.setMetaModelElement(IModelElement metaModelElement);

Source: VP API-call to link diagram elements between different diagrams

The internal structure of a component can be created by adding child elements to it. For this process the diagram element of the component and the shape element of the child are required. The integration of a child is achieved by the following function:

1) diagramElement.addChild(IShapeUIModel childShapeModel);

Source: VP API-call to add a child element to a diagram element

Now all crucial methods are known to generate the component diagram and the result may look like displayed in Fig. 11.

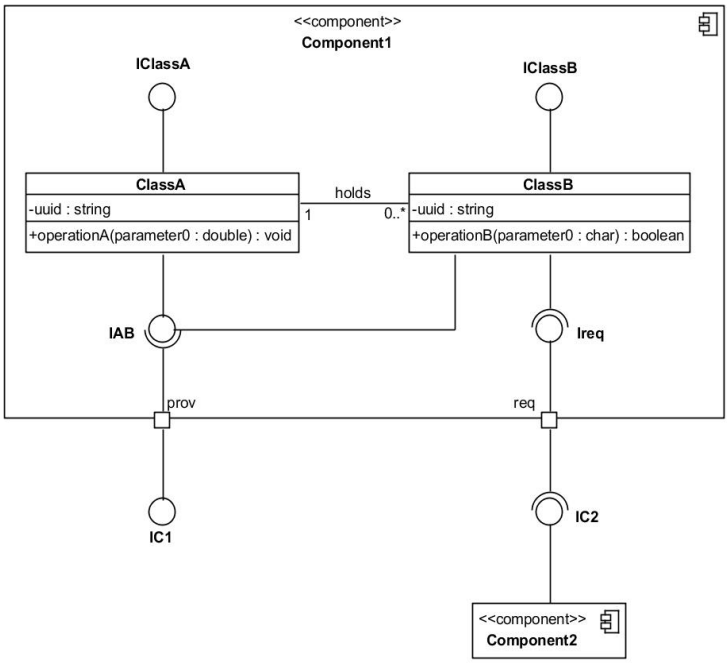

Fig. 11. Completely generated component diagram.

The last step performed by the plugin is to add a component's member to its references by calling the following operation:

1) diagramElement.setReferencedByModelAddresses(String[] referencedModelAddresses);

Source: VP API-call to add references to an element

The parameter requires a string array of addresses of model elements that refer to the component which can be retrieved by the following method:

1) modelelement.getAddress();

Source: VP API-call to retrieve a model element's address

The result can be viewed by clicking on the component and the curved arrow in the lower right-hand corner, as displayed in Fig. 12.

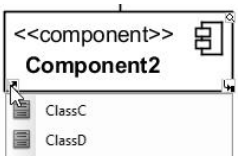

Fig. 12. References of Component2

Component2 has references to its members ClassC and ClassD, as in Fig. 9. Their public operations are stored in the specification of the component's interface IC2.

6. Conclusion

The creation of a software design is a protracted process. Automating this process or parts of it is connected with several difficulties. The proposed plugin offers a solution that facilitates the software modeling process of a component-based software architecture and is based on attributes of a class diagram. The output is a simplified component diagram.

References

[1] EMC corporation and IDC. (2014). *EMC Digital Universe study with research and analysis by IDC*. Retrieved 2014, from http://www.emc.com/leadership/digital-universe/index.htm#2014

[2] Jacobson, I. (2000). *The Road to the Unified Software Development Process*. Cambridge, U.K.: Cambridge University Press.

[3] Birrell, N. D., & Ould, M. A. (1986). *A Practical Handbook for Software Development*. Cambridge, U.K.: Cambridge University Press, 1986.

[4] Budgen, D. (2003). *Software Design*. Pearson Education.

[5] Szyperski, C. (2002). *Component Software, Beyond Object-Oriented Programming*(2nd ed.). Pearson Education.

[6] Visual paradigm. *Selected Users List* (2013). Retrieved 2013, from http://www.visual-paradigm.com/aboutus/userlist.jsp

[7] Visual paradigm. (2014). *User's Guide, Part I. Developing Plugin*. Retrieved 2014, from http://www.visual-paradigm.com/support/documents/pluginuserguide.jsp

[8] Visual paradigm. *FAQ, IDE Integration*. (2014). Retrieved 2014, from http://www.visual-paradigm.com/support/faq.jsp

[9] Visual paradigm. (2009). *Debug your plug-in with eclipse*. Retrieved 2009, from http://www.visual-paradigm.com/tutorials/debugplugin.jsp

[10] Visual paradigm, *VP Gallery, Component diagra*m. (2015). Retrieved 2009, from http://www.visual-paradigm.com/VPGallery/diagrams/Component.html

[11] Object management group. (2011). *OMG Unified Modeling Language (OMG UML), Superstructure*. Retrieved 2011, from Available: http://www.omg.org/spec/UML/2.4.1/Superstructure/

Carsten Mueller holds a M.Sc. degree from the University of Liechtenstein in Vaduz and a Ph.D. degree from the University of Economics (Department Informatics and Statistics) in Prague. He works since 15+ years as an independent consultant and trainer in the areas of IT process optimization, Object-oriented Modelling and IT Governance. He is currently the technical head of Processes and IT Management at a large airport in Germany and senior lecturer for software engineering, UML and algorithms at the Baden-Wuerttemberg Cooperative State University.

Anton Pohl works for a leading global supplier of technology and service in Germany. He studies Applied Informatics at the Baden-Wuerttemberg Cooperative State University and is a member of a research group.