Common Metamodel of Component Diagram and Feature Diagram in Generative Programming

Novak, Matija; Magdalenić, Ivan; Radošević, Danijel

Source / Izvornik: **Journal of Computer Science, 2016, 12, 517 - 526**

Journal article, Published version
Rad u časopisu, Objavljena verzija rada (izdavačev PDF)

https://doi.org/10.3844/jcssp.2016.517.526

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:990492

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

Journal of Computer Sciences

Original Research Paper

Common Metamodel of Component Diagram and Feature Diagram in Generative Programming

Matija Novak, Ivan Magdalenić and Danijel Radošević

Faculty of Organization and Informatics, University of Zagreb, Varaždin, Croatia

Article history
Received: 05-07-2016
Revised: 09-11-2016
Accepted: 14-12-2016

Corresponding Author:
Matija Novak
Faculty of Organization and Informatics, University of Zagreb, Varaždin, Croatia
Email: matnovak@foi.hr

Abstract: Component-based software engineering and generative programming are common approaches in software engineering. Each approach has some benefits and domain of usage. Component-based development is used to build autonomous components that can be further combined in different ways, while generative programming is more suitable when building systems that have different variants. Before a variable component based system can be build, it needs to be modeled. In this article, a new common metamodel that aims to enable modeling a system which combines both component-based development and generative programming is introduced. The introduced metamodel proposed in this paper combines the component diagram that is used to model systems in component-based development and the feature diagram that is employed in modeling systems in generative programming. The combined metamodel enables modeling of variable systems using components.

Keywords: Metamodel, Component-Based Development, Component Diagram, Generative Programming, Feature Diagram

Introduction

Component-Based Software Engineering (CBSE), is one of the most common approaches in software engineering today. The basis for this approach is Component Based Development (CBD) (Crnkovic *et al*., 2006).

CBD enables software development that consists of autonomous and loosely coupled components. It is common in Service-Oriented Architectures (SOA), in which a system is divided into different services which communicate together via interfaces. A UML component diagram is used to define the structure of a component based system.

This approach is not very practical when different system variants exist, which is often the case in generative programming. This type of programming enables the development of systems that exhibit very similar characteristics. A new system is thus only a new variant of the same system that is specialized for a particular purpose. While Feature Oriented Analysis (FODA) is used to design such systems, a Feature Diagram (FD) is employed to define the features of the system. Feature models are widely used for variability and commonality management in software product lines (Benavides *et al*., 2010).

So what if we want to make a system that is both component based and variable. A component diagram is not adequate for that purpose because it is not suitable for modeling of variants. Moreover, it is aimed to show the system structure and does not define what the features of the system are. Conversely, the downside of a feature diagram is oriented to show the system features rather than system structure. To overcome this gap we have combined these two models using their metamodels. A good description of metadata models development can be found in (Hay, 2006).

In this article we will describe two models: UML component diagram, later referred to as "component diagram" and feature diagram. Our aim is to introduce a common metamodel based on the metamodels of these two diagrams that would enable us to model a family of systems which use components.

For example, this common metamodel will be used to model the workflow generator for Extract, Transform and Load (ETL) processes. "Extract, Transform and Load (ETL) is a process that makes it possible to extract data from operational data sources, to transform data in the way needed for data warehousing purposes and to load data into a Data Warehouse (DW)". (Novak and Rabuzin, 2014) When

Science Publications

© 2016 Matija Novak, Ivan Magdalenić and Danijel Radošević. This open access article is distributed under a Creative Commons Attribution (CC-BY) 3.0 license.

building a data warehouse, about 70% of time and resources (or 80%, according to Inmon (2002)) is used for the ETL purposes. So to speed up this process we want to build an ETL workflow generator that would be able to generate different ETL workflows for current ETL systems. Before constructing this system, we first need to model it. Because this ETL workflow generator should be modeled as a variant component based system, we need to combine both of the aforementioned modeling techniques to fully represent it.

An important consideration that needs to be mentioned here is that this system is intended to be message based, with all communication conducted via messages. As a result, most of its components constitute integration patterns which are described in (Hohpe and Woolf, 2012). The description of message patterns and communication through messages in the ETL workflow generator is out of the scope of this article and will not be discussed. As mentioned before, the focus of our article is on variant component systems.

The approach used in this paper differs from other approaches reported in literature mainly in the model structure and the representation of the common metamodel as well as its implementation. In the SCT generator model (Radošević and Magdalenić, 2011), for example, the metamodel uses model elements Specification, Configuration and Templates that can be graphically represented by a specification diagram and a configuration diagram. Also, SCT is implemented as a source-code generator that can generate program code on demand (Magdalenić *et al*., 2013) or build program files. The majority of approaches is based on metaprograms. Metaprograms are defined as generic, incomplete, adaptable programs (Jarzabek *et al*., 2006). Some approaches are based on frames defined as XML frames, such as XVCL (Jarzabek *et al*., 2003), while others are oriented at some specific features of the problem domain that require using particular kinds of metamodels, like ontologies, as described in (Magdalenić *et al*., 2009).

On the other hand, the approach presented in our paper has some similarities with metaclass based approaches described in Grigorenko *et al*. (2005; Tolvanen and Rossi, 2003; De Lara and Vangheluwe, 2004). This model is also based on UML and have some similarities in testing phase with (Xu *et al*., 2008).

This article is structured as follows. In section 2 we describe the methodology used. In section 3 we describe the component diagram and in section 4 the metamodel of the component diagram. Sections 5 and 6 contain the description of the feature diagram and its metamodel. The common metamodel combining the metamodels of the component diagram and the feature diagram is introduced in section 7, along with a description of applications of this new metamodel. Conclusions and future work are given in section 8.

Used Methodology

As already stated in the introduction section a good description of metadata models development can be found in (Hay, 2006). But there are other works using the same approach like (Androcec and Dobrovic, 2012). But let us briefly describe the used approach and for more details read (Hay, 2006).

To develop a new model a specific problem is needed that cannot be solved with models that we already have. In our case this was the model for a modeling family of systems which use components. Once we know the problem we need to find two or more models that can partially solve the problem.

Next, what we need to describe all elements of every model that we want to combine. In our case these are “future diagram” and “component diagram”. What we do next is an “Appearance table” that enables us to get a form of “appearances in real world” to a “metadata model” then “metadata metadata model” and so on. We are going up this chain so long until we get to a point where two of our models have the same representation in our case this was “metadata metadata model”.

Once we find that point, we go one level down and make an entity-relationship model of this level in our case this was “metadata model”. Once we have the entity-relationship model the hardest part is to find a point where these two models can be merged. There should be at least one point (like entity) which represents basically the same thing and that can be merged. In our case we had two connecting points, but this is described in section 7.

Once these connection points are found we merge the two entity-relationship models with all the existing attributes. If there are duplicate attributes we remove them. Last step is to specify how this new model should be used. Since all this started with a specific problem an example through all these steps is appreciated.

Now in the next chapter we will start with the description of the component diagram and continue the development of a new model based on the described approach.

Component Diagram

A component diagram shows components, provided and required interfaces, ports and relationships between them. This type of diagrams is used in Component-Based Development (CBD) to describe systems with Service-Oriented Architecture (SOA) (Fakhroutdinov, 2014).

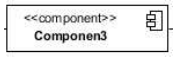
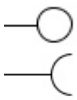
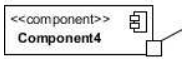
A component diagram provides architects with a neutral format for modeling solutions. Its purpose is to show the structure and connections between components. In other words, a component diagram shows the high-level system architecture (Bell, 2004).

The creation of a component diagram can be described through the following steps:

- Identify and define system components and stereotypes
- Define component ports
- Define component interfaces
- Group components into more complex components (vertical composition)
- Define connection between components
- Describe component restrictions

To be able to create a component diagram it is necessary to know the elements that the component diagram is made of. Table 1 displays the elements and their descriptions. A generic example of the component diagram in Fig. 1 shows a simple system which has four components, where component 3 requires component 1 and provides one interface for usage. Component 1 consists of two components: Component 2 and component 4, which are connected. Component 4 delegates its port to the parent component 1. Component 1 then offers this port as a provided interface which is, as already mentioned, used by component 3.

Table 1. Elements of component diagram

Representation (All images are based on (Bell, 2004))	Element	Description
	Component	An autonomous unit containing a particular part of the system logic and providing interfaces for use.
	Interface	The interface can either be provided or required. A provided interface is a formal contract which offers components to some client. A required interface tells what other components a particular component depends on. Although each component is an autonomous unit, it can depend on other components.
	Relationship	The relationship is a connection between two components. The lollipop denotes the provided interface and the socket denotes the required interface.
	Port	Port is the entrance to the internal structure of the component. One port can have one or zero interfaces. A port can be delegated when the internal port of the component is delegated to the external component.
Looks like a component. The only difference that instead of the key word <<component>> the name of the stereotype is included.	Stereotype	The following types of stereotypes exist: Subsystem, process, service, specification, realization and implementation.

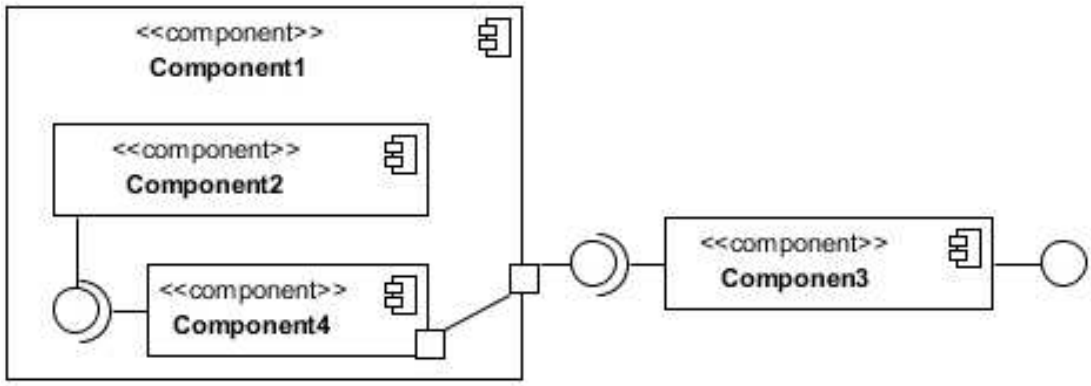


Fig. 1. Example of a component diagram with delegated interfaces

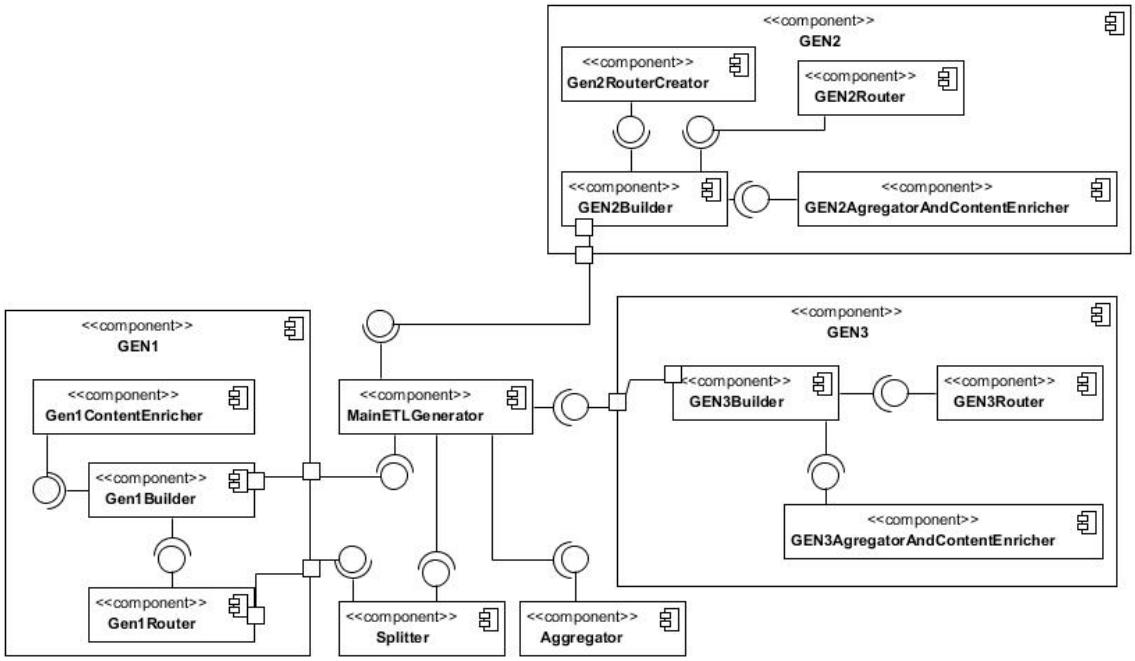


Fig. 2. Example of component diagram of ETL workflow generator

Figure 2 gives the component diagram of the ETL workflow generator, which consists of three main components (GEN1, GEN2 and GEN3). The idea of the ETL workflow generator, as explained in the introduction, is to generate ETL workflows. The purpose of the "GEN1" component is to generate workflow parts that correspond to transformations on one attribute from one data source. The "GEN2" component is supposed to generate workflow parts that correspond to transformations that use two or more attributes from one source. The "GEN3" component generates workflow parts that correspond to transformations that use two or more attributes from different sources.

The "MainETLGenerator" component represents the main logic which uses these three main components via their interfaces and delegates jobs to these components. "MainETLGenerator" also uses the "Splitter" and "Aggregator" components. Since the whole system is intended to be message based, "Splitter" and "Aggregator" are basically integration patterns. The labels "Router", "ContentEnricher" and "Aggregator" also refer to different integration patterns.

Similar to the "MainETLGenerator" component, the "GEN1" component is used by the "Splitter" component. "GEN1" consists of three components: "Gen1ContentEnricher", "Gen1Builder" and "Gen1Router". The components "Gen1Builder" and "Gen1Router" delegate their ports to the parent component "GEN1". Furthermore, the "Gen1Builder" component uses the "Gen1Router" component and the "Gen1ContentEnricher" component. The "GEN2" component consists of four components: "Gen2RouterCreator", "Gen2Router", "Gen2AgregatorAndContentEnricher" and "GEN2Builder".

"Gen2Builder" uses the other three components inside the "GEN2" component and delegates its port to the parent component. The "GEN3" component consists of three components: "Gen3Builder", "GEN3AgregatorAndContentEnricher" and "GEN3Router". "GEN3builder" uses the other two components from the "GEN3" component and delegates its port to the parent component. Based on the component diagram in Fig. 2, we propose a metamodel of the component diagram described in Section 4.

Metamodel of Component Diagram

To be able to combine the component diagram with the feature diagram, we first need to create their metamodels. For the purpose of creating the metamodels, we used the "Appearance table" as described in (Hay, 2006). Table 2 offers the "Appearance table" for the component diagram. Four main parts that every component diagram consists of are: "Component", "Relationship", "Port" and "Interface".

Using the ERA model in Table 2, we developed a metamodel of the component diagram (Fig. 3). Let us explain the ERA model. The relationship between the entities "Component" and "Port" is clear. One port must belong to one component and one component can have zero or multiple ports. The relationship between

"Component" and "Interface" is similar. One component can theoretically have zero or multiple interfaces. One interface must belong to one component.

Although we used the "Relationship" as an entity in Table 2, in the ERA model it can be represented as a unary relation on the entity "Interface" and a unary relation on the entity "Component". This is because two components are always connected through some interface. If one component has a required interface for the other component, then this required interface can be connected only by using the provided interface of the other component. One required interface can be connected to only one provided interface, but one provided interface can be connected to multiple required interfaces. In other words, one provided interface can be used multiple times by different components. Since a component can be placed inside another component (in the so-called vertical composition), we need another unary relation on the entity "Component". Every component can be a parent of zero or multiple components and one component can have zero or one parent.

Table 2 Appearance table for component diagram

Element of metadata (metadata model) "meta-metadata"	OBJECT „Entity type" „Attribute"			
Metadata model „metadata"	ENTITY TYPE „Component" ATTRIBUTE „ID, Stereotype, Name, Component part"	ENTITY TYPE „Relationship" ATTRIBUTE „Relationship type"	ENTITY TYPE „Port" ATTRIBUTE „Name, Port type, belongs, offers "	ENTITY TYPE „Interface" ATTRIBUTE „Name, Interface type, Provided, Required",
Appearance data in the real world "Appearance in the real world"	COMPONENT „ Gen1Router, Gen1 Builder, GEN1"	RELATIONSIP „Gen1Router-Gen1Builder, Gen1Content Enricher-Gen1Builder, GEN1- Gen1Builder" RELATION TYPE „ vertical composition, component relation"	PORT „GEN1-Gen1Builder" PORT TYPE „delegated" BELONGS TO COMPONENT „GEN1" OFFERS COMPONENT „Gen1Builder"	INTERFACE „Gen1Router provides an interface, Gen1Builder requires interface" INTERFACE TYPE „requires, provides"
Appearance in the real world	Gen1Router, Gen1Builder, GEN1	Gen1Builder requires Gen1Router, Gen1Builder requires Gen1Content Enricher, GEN1 consist of component Gen1Builder	Gen1Builder delegates port to GEN1	Gen1Router provides an interface, Gen1Builder requires an interface

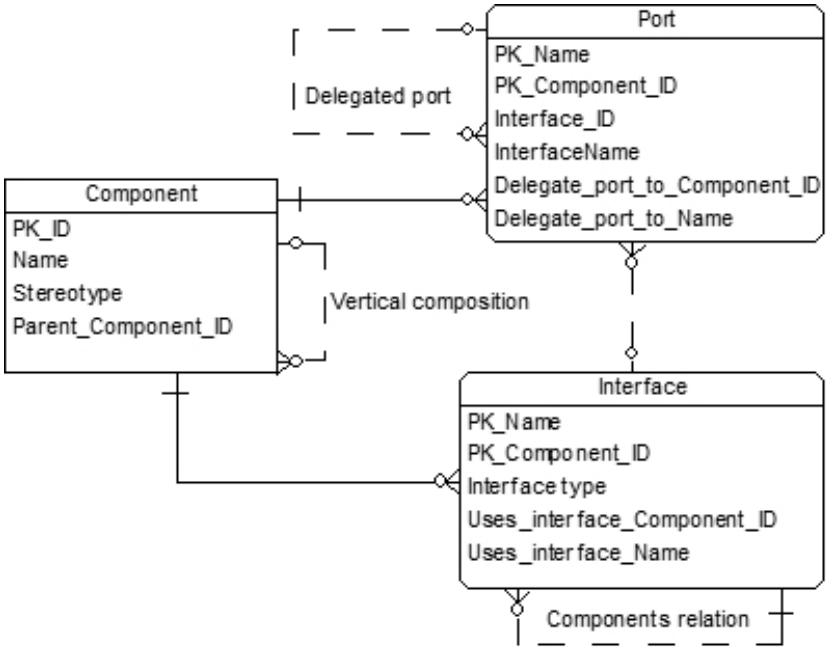


Fig. 3. ERA model-metamodel of component diagram

In the ERA model there is a third unary relation – the "Delegated port". This unary relation is needed when one component delegates its internal port to a parent component. A port might not be delegated, or can be delegated only once. However, one port to which delegation is made can be used multiple times by different delegated ports of child components. It is also evident that in the ERA model "port" is connected to "Interface". A port might not have an interface (in the case of delegated port).

Conversely, an interface might not have a port because it can be directly connected to a component. On the other hand, one interface can represent multiple ports.

Feature Diagram

A feature diagram shows hierarchical feature decomposition including if some feature is mandatory or not, alternative or optional (Czarnecki and Eisenecker, 2000). This diagram is used in domain analysis to show variability and common features in some domain.

The feature diagram creation process can be described using the following steps:

- Define the concept that needs to be modeled
- Define the main features of the concept
- Define the relation type between the concept and the main features and between the features on the same level
- Define the features of features
- Define the relation type between every feature and its parent feature and between features on the same level
- Repeat steps 4 and 5 until you get to the last feature that contains no more child features

An example of a feature diagram is shown in Fig. 4, while its elements are described in Table 3. Figure 4 offers a simple feature diagram which has one "concept" feature that is modeled. The "concept" consists of two main features, one of which is optional and one mandatory. The "optional feature" further consists of two features, at least one of which is mandatory. The "mandatory feature" also consists of two subfeatures, but this time they are mutually exclusive alternatives. While the "optional feature" can have both features at the same time, the "mandatory feature" can have only one "alternative feature" at the same time.

Figure 5 represents an example of a feature diagram which defines the ETL workflow generator that consists of five main features: "GEN1", "GEN2", "GEN3", "Splitter" and "Aggregator". "GEN2" and "GEN3" are optional and "GEN1" is mandatory. This is because every variant will have at least one data source so "GEN1" will always be used, while "GEN2" and "GEN3" will be used if transformations on multiple attributes or data sources are used. "ETLGenerator" can either have the "Splitter" or the "Aggregator" feature, or both. "GEN1", "GEN2" and "GEN3" contain child features, some of which are mandatory and some optional. The"GEN1" feature must contain "Router" and "Builder", while "ContentEnricher" is optional. Similarly, "GEN2" must have "Builder" and "RouterCreator", but "Router" is optional and itself contains the optional subfeature "AggregatorAncContentEnricher". "GEN3" only contains the mandatory feature "Builder", while "Router" is optional. "Router" in "GEN3" also has the optional sub-feature "AggregatorAndContentEnricher". Based on Fig. 5 below, the metamodel of the feature diagram presented in the following section will be made.

Metamodel of Feature Diagram

To be able to combine the component diagram with the feature diagram, we first need to create their respective metamodels. To create the metamodel of the feature diagram we also used the "appearance table" (Table 5), as in the case of the metamodel previously described in Section 4. As may be seen from Table 5, the feature diagram consists of four entities at the meta-level. The "Relation" entity represents the parent-child relation between features. The "Connection" entity represents the relation between features on the same level. So the "Relation" element from the feature diagram is split into two entities, "Relation" and "Connection". This is necessary because one feature can at the same time be connected to another feature on the same level and have a parent and/or have children.

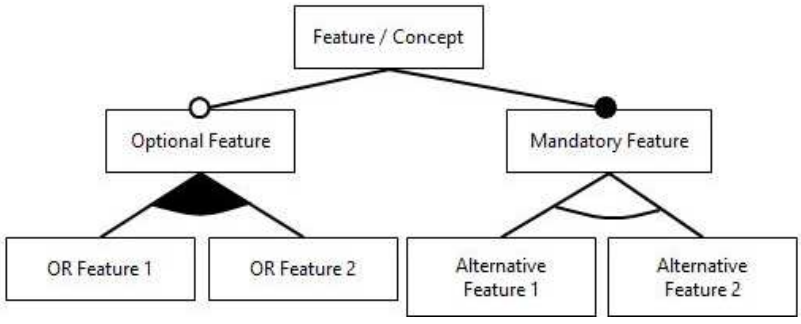


Fig. 4. Feature diagram-simple example

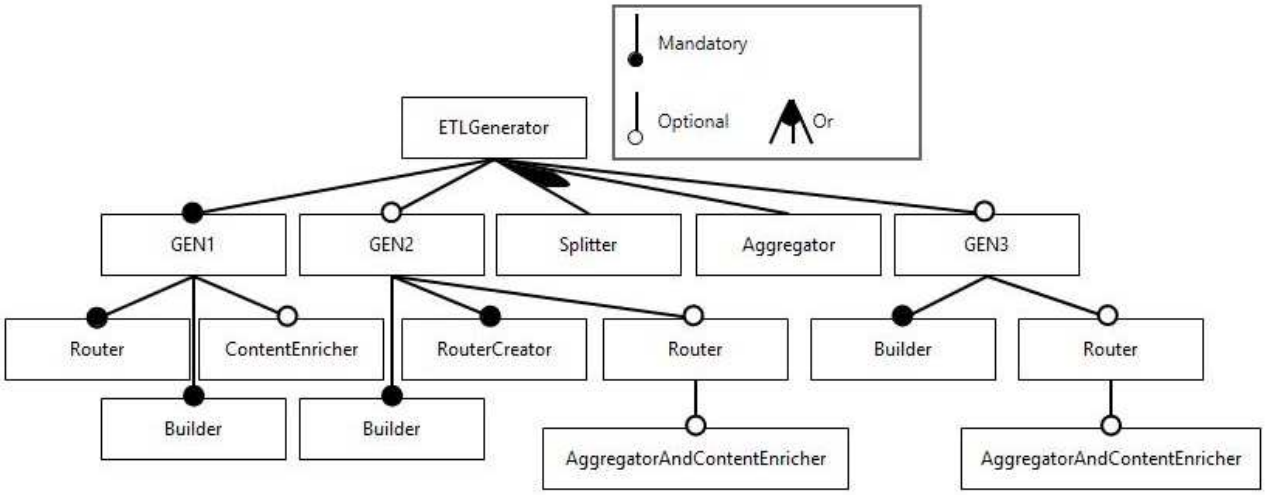


Fig. 5. Example of feature diagram of ETL workflow generator

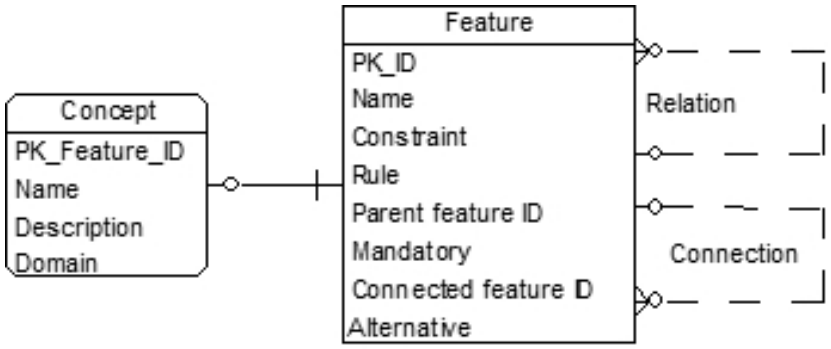


Fig. 6. ERA model-metamodel of feature diagram

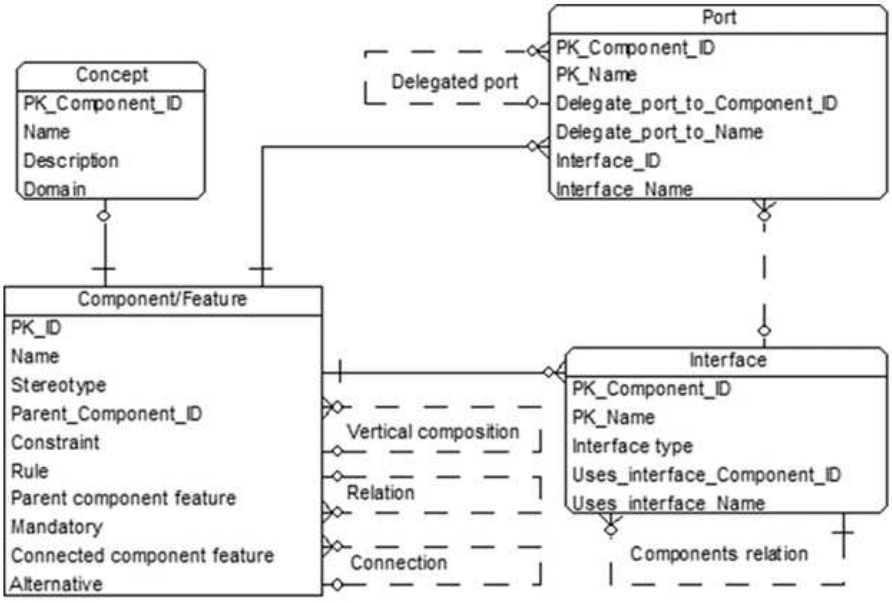


Fig. 7. ERA model-common metamodel of metamodel of component diagram and feature diagram

Table 3. Description of feature diagram elements

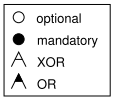
Label/Representation	Element	Description (based on (Apel and Christian, 2009))
„Feature / Concept“	Concept	Represents the concept from some domain that is modeled.
„Optional Feature”, “Mandatory Feature”, etc.	Feature	Represents a part of the concept. A feature can be: First level feature, which means it is the main part of the concept; or it can be a feature of a feature of a modeled concept.
	Relation	Relation represents the parts of which a subject consists. The relation can be mandatory or optional, indicating whether a particular feature must or does not have to be a part of a subject or a concept. The relation can also be “XOR alternative”, which means that some features are mutually exclusive alternatives to each other. For example, a computer can either have an Intel i3 processor or an Intel i5 processor, but not both of them. Furthermore, the relation can be “OR”, which means that, for example, a computer can have a USB 3.0 and/or a USB 2.0 port.

Table 4. Appearance table for feature diagram

Element of metadata (metadata model) “meta-metadata”	OBJECT „Entity type“„Attribute“			
Metadata model „metadata“	ENTITY TYPE „Concept“ ATTRIBUTE „Name, description, domain“	ENTITY TYPE „Feature“ ATTRIBUTE „ID, Name, Constraint, Rule“	ENTITY TYPE „Relation“ ATTRIBUTE „Relation type, Parent feature, Child Feature“	ENTITY TYPE „Connection“ ATTRIBUTE „Feature 1“ „ Feature 2“ „Type of connection“
Appearance data in the real world “Appearance in the real world”	CONCEPT „ ETL WF generator “	OSOBINA „GEN1, GEN3, GEN2, Splitter“	RELATION „GEN1-Router“ FEATURE „Router“ „Builder“	CONNECTION „Splitter-Aggregator “ FEATURE „Splitter“ „Aggregator“ “ETLGenerator”
Appearance in the real world	ETL WF generator	GEN1, GEN3, GEN2, Splitter	GEN1 has mandatory feature Router.	ETLGenerator can have Splitter or Generator or both

Based on Table 4, we created the ERA model (i.e., metamodel of the feature diagram) that is shown in Fig. 6. As in the component diagram, the “Relation” entity in the ERA model is represented as a unary relation on the entity “Feature”. “Feature” can have one parent and one parent can have zero or multiple children. The same applies to “Connection” that is also represented as a unary relation. One feature can be in connection with zero or multiple features of the same level. If feature 1 is in connection with feature 2 and feature 2 is in connection with feature 3, then feature 1 is implicitly in connection with feature 3. Owing to this, a unary relation is sufficient. It is not necessary to make a new table, as defined by the representation rules in ERA modeling for the M:N relationship. A mandatory field in the entity “Feature” indicates if the relation is mandatory or optional. The alternative field in the entity “Feature” indicates whether the connection is “OR” or “XOR”.

“Concept” is basically the same as “Feature”, the only difference being that it has no parent and has no features on the same level. Since “Concept” has some extra attributes, the relation between “Concept” and “Feature” is 1:1. In other words, a feature can be a concept, but one concept can only be represented by one feature.

Common Metamodel

As already mentioned in the introduction, we attempted to combine the component diagram and the feature diagram with the aim of modeling system families by using components. The component diagram only shows the structure of the system and not its features. The feature diagram shows the features of the system, but does not necessarily show the complete structure and all the connections within the system. From the feature diagram we can conclude which parts are variable and which are not. The component diagram makes it possible to discern the overall structure of the new system as well as connections within it. However, from these two diagrams it is still not possible to tell how many times a particular feature can be generated. In the case of the ETL workflow generator, this depends on configuration messages that the system receives.

Now that both metamodels have been created, the two diagrams can be connected. The common point between these two models is the “Concept”. In the feature diagram the “Concept” is the main feature (i.e., the feature that has no parent), while in the component diagram the “Concept” encompasses the whole diagram. In other words, the “Concept” corresponds to what is

modeled by means of the component diagram. In a certain system the "Concept" can be a whole new component with new interfaces to be used by other systems or components. Other connections are the features in feature diagram. Features correspond to components in the component diagram and, conversely, components are equal to features in the feature diagram.

Every feature in the feature diagram will have one corresponding component in the component diagram. Since every component does not have to implement an important feature, we can have more components than features. In the common model we can say that every feature is implemented by one or more components.

One should consider that the vertical composition in the component diagram is not the same as the parent-child relation in the feature diagram. For example, in the feature diagram the feature "GEN2" has a child feature "Router" that has a child "Content enricher". In the component diagram the "GEN2" component has two inner components, "Router" and "Content enricher", that are on the same level.

The common metamodel is represented in Fig. 7. To create the common metamodel (Fig. 7) we copied the metamodel of the component diagram (Fig. 3) and enhanced it with the missing elements from the metamodel of the feature diagram (Fig. 6). In the common metamodel, the "Component" entity (copied from the component diagram metamodel) is equal to the "Feature" entity from feature diagram metamodel. So the "Component" entity in the common metamodel is renamed to "Component/Feature".

All the attributes from the entity "Feature" from the feature diagram metamodel are added to the entity "Component/Feature" except for the "Name" and "PK_ID" attributes. "PK_ID" is not added because we need only one primary key. "Name" is not added because every feature corresponds to one component and the feature can therefore be named after the component. The "Concept" entity is simply taken in its original form from the feature diagram metamodel and added to the common metamodel. The "Concept" entity is connected to the "Component/Feature" entity through the same relation that connected it to the "Feature" entity in the feature diagram metamodel. The meaning of each attribute stays the same as it was in the original metamodel that it was taken from.

The steps for using this new common metamodel are as follows:

- Define the concept that needs to be modeled
- Define the main features of the concept
- Define the relation type between the concept and the main features and between the features on the same level
- Define the features of features
- Define the relation type between every feature and its parent feature and between features on the same level
- Repeat steps 4 and 5 until you get to the last feature that contains no more child features (7) All features now become components and define any missing components that are not represented by features
- Define stereotypes
- Define component ports
- Define component interfaces
- Group components into more complex components (vertical composition)
- Define connection between components
- Describe component restrictions

Conclusion and Future Work

In this work we presented two modeling diagrams which are used in software development. Each of them is associated with a different approach so they are not normally used together. In this work we combined these two modeling methods to solve the problem of modeling system families that are built by jointly using components and generative programming.

By abstracting the component diagram and the feature diagram to the metamodel level we successfully combined these two different models. The major benefit of the presented approach is that we showed that these two models perfectly complement each other. We can therefore say that it is possible to model systems that use generative programming and also include components.

As an example of such an approach we modeled the ETL workflow generator that is intended to be used for generating ETL workflows for traditional systems.

In our future work, we plan to build the ETL workflow generator prototype based on the presented metamodel using components and generative programming. Components like "GEN2" and "GEN3" will have features like "Router" or "Content Enricher" which we aim to generate during the execution of the program based on the configuration. At the same time, "Router" and/or "Content Enricher" will be components from which the system is built.

So our focus in the future will be on the automation for generating ETL workflow. The idea is to use the common metamodel to build a system which will "model" ETL process based on semantics and suggest the needed transformations and mappings for automatic generation of ETL workflows. Also, as stated in the introduction section this system will be message based. In our future work we plan also to focus on the description of message patterns and communication through messages that will be used in ETL workflow generator. For the implementation we plan to use Apache Camel.

Author's Contributions

Matija Novak: Proposed the main idea of this study and wrote the first version of the manuscript. Also he was mainly responsible for the component diagram elements and ETL.

Ivan Magdalenić: Connected the two diagrams and complemented the connection to generative programming.

Danijel Radošević: Wrote the introduction to related work. He was mainly responsible for the feature diagram elements.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and no ethical issues involved.

References

Androcec, D. and Z. Dobrovic, 2012. Creating hybrid software engineering methods by means of metamodels. Proceedings of the 34th International Conference on Information Technology Interfaces, Jun. 25-28, IEEE Xplore Press, pp: 481-486. DOI: 10.2498/iti.2012.0430

Apel, S. and K. Christian, 2009. An overview of feature-oriented software development. J. Object Technol., 8: 49-84. DOI: 10.5381/jot.2009.8.5.c5

Bell, D., 2004. UML basics: The component diagram.

Benavides, D., S. Segura and A. Ruiz-Cortes, 2010. Automated analysis of feature models 20 years later: A literature review. Inform. Syst., 35: 615-636. DOI: 10.1016/j.is.2010.01.001

Crnkovic, I., M. Chaudron and S. Larsson, 2006. Component-based development process and component lifecycle. Proceedings of the International Conference on Software Engineering Advances, Oct. 29-Nov. 3, IEEE Xplore Press, pp: 44-44. DOI: 10.1109/ICSEA.2006.261300

Czarnecki, K. and W.U. Eisenecker, 2000. Generative programming: Methods, Tools and Applications. 1st Edn., Addison Wesley, Boston, ISBN-10: 0201309777, pp: 832.

Fakhroutdinov, K., 2014. UML component diagrams.

Grigorenko, P., A. Saabas and E. Tyugu, 2005. Visual tool for generative programming. Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Sept. 05-09, ACM Press, New York, pp: 249-252. DOI: 10.1145/1081706.1081747

Hay, D.C., 2006. Data Model Patterns: A Metadata Map. 1st Edn., Elsevier Morgan Kaufmann, ISBN-10: 0120887983, pp: 406.

Hohpe, G. and B. Woolf, 2012. Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions. 1st Edn., Addison-Wesley, Boston, ISBN-10: 0133065103, pp: 735.

Inmon, W.H.H., 2002. Building the Data Warehouse. 3rd Edn., John Wiley and Sons, New York, ISBN-10: 0471270482, pp: 432.

Jarzabek, S., P. Bassett, H. Zhang and W. Zhang, 2003. XVCL: XML-based variant configuration language. Proceedings of the 25th International Conference on Software Engineering, May 3-10, IEEE Xplore Press, pp: 810-811. DOI: 10.1109/ICSE.2003.1201298

Jarzabek, S., H. Zhang, S. Ru, V.T. Lam and Z. Sun, 2006. Analysis of meta-programs: An example. Int. J. Software Eng. Knowledge Eng., 16: 77-101. DOI: 10.1142/S0218194006002689

De Lara, J. and H. Vangheluwe, 2004. Defining visual notations and their manipulation through meta-modelling and graph transformation. J. Visual Lang. Comput., 15: 309-330. DOI: 10.1016/j.jvlc.2004.01.005

Magdalenić, I., D. Radošević and T. Orehovački, 2013. Autogenerator: Generation and execution of programming code on demand. Expert Syst. Applic., 40: 2845-2857. DOI: 10.1016/j.eswa.2012.12.003

Magdalenić, I., D. Radošević, anf Z. Skočir, 2009. Dynamic generation of web services for data retrieval using ontology. Informatika, 20: 397-416.

Novak, M. and K. Rabuzin, 2014. Prototype of a Web ETL tool. Int. J. Adv. Comput. Sci. Applic., 5: 97-103. DOI: 10.14569/IJACSA.2014.050614

Radošević, D. and I. Magdalenić, 2011. Source code generator based on dynamic frames. J. Inform. Organiz. Sci., 35: 73-91.

Tolvanen, J.P. and M. Rossi, 2003. Metaedit+: Defining and using domain-specific modeling languages and code generators. Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, Oct. 26-30, Anaheim, CA, USA., pp: 92-93. DOI: 10.1145/949344.949365

Xu, D., W. Xu and W.E. Wong, 2008. Testing aspect-oriented programs with UML design models. Int. J. Software Eng. Knowl. Eng., 18: 413-437. DOI: 10.1142/S0218194008003672