



Internal Report 07-04

augustus 2007

Universiteit Leiden

Opleiding Informatica

“Generative Programming through Inheritance”

Bas Kaptijn

Bachelorscriptie

Leiden Institute of Advanced Computer Science (LIACS)

Leiden University

Niels Bohrweg 1

2333 CA Leiden

The Netherlands

Generative Programming through Inheritance

in a web application development environment

Bas Kaptijn
Cope / LIACS, Leiden University

28th August 2007

Abstract

Generative Programming is a software engineering methodology focused on the automated development of systems families rather than manually developing single systems from scratch.

In this paper, the very basics of Generative Programming are explained. A method called “Generative Programming through Inheritance” is introduced.

“Generative Programming through Inheritance” is about using component frameworks as domain specific languages and using an inheritance relation between components to make the generation from one domain specific language to the other possible.

All of this is illustrated by a web application development environment supporting the “Generative Programming through Inheritance” method. This is currently being developed as a next major version of a successful existing product. This product is capable of generating web survey applications without much effort from developers.

Table of Contents

Introduction

Part 1: Generative Programming

- 1.1 What is Generative Programming?
- 1.2 The Generative Domain Model (GDM)
- 1.3 Generative Programming Steps
 - 1.3.1 Domain Scoping
 - 1.3.2 Feature modeling
 - 1.3.3 GDM Development
- 1.4 Another example: Quaestio

Part 2: Generative Programming through Inheritance (GPI)

- 2.1 Components and Parameters
- 2.2 Atomized components
- 2.3 Feature modeling restricted references as parameter value
- 2.4 Component Frameworks as Domain Specific Language
- 2.5 The inheritance relation “Is a” as a generator step
- 2.6 Bottom-up Generation
- 2.7 Encapsulation and Polymorphism
- 2.8 GPI compared to other technologies

Part 3: Example Web application Development Environment supporting GPI

- 3.1 Example: Quaestio recreated using GPI
- 3.2 Quaestio 6

Conclusion

Outlook

References

Introduction

At Internet Solutions companies such as Cope [9] where the author works, a lot of work is needlessly done more than once; wheels are often reinvented. It goes without saying that it is better to focus at the outset on reuse as much as possible. At Cope the shift in focus on reuse was accomplished by developing a very simple Domain Specific Language (DSL) [3] wrapped in the product Quaestio [6]. Quaestio includes a development tool for easy “configuring” web applications and in particular surveys. This Program has been quite successful for years now. This DSL is in fact a framework of parameterizable components. Solutions can be “configured” by placing components in a tree structure and defining values for their parameters. This results in a “language” that is easy to learn and use. It is even possible to add scripts to the solutions made with Quaestio that add or override behavior and extend the DSL. Currently this is done increasingly more frequently. Solutions for different domains are delivered including domains that the original DSL was not designed for.

The Domain Specific Language developed in Quaestio now addresses a too narrow domain space that is hard to change. This is a known disadvantage with DSLs [3].

After some research we concluded that we need to be able to engineer reusable component frameworks in an evolving way [4]. These frameworks would form the desired DSLs upon which solutions can be built faster.

By using component frameworks, which can be defined in terms of other component frameworks through a form of inheritance, we combine inheritance with component-based software engineering resulting in a Generative Programming approach [2] we called “Generative Programming through Inheritance” (GPI).

This paper explains what GPI is and how it can be used. First the main concept of Generative Programming itself is explained in Part 1. Then in Part 2, Generative Programming through inheritance is introduced in detail and in Part 3 we describe a development environment that supports GPI and which is currently in development at Cope.

Part 1: Generative Programming

In order to describe what GPI is we first introduce the concept of Generative Programming. In the current Part we give a very basic summary of the notion of Generative Programming. In conclusion we describe the example which inspired us to introduce the idea of GPI. Except this example, the bulk of the material in this part is taken from the main source on this subject: *Generative Programming - Methods, Techniques and Applications* by Czarnecki and Eisenecker [2]. Only those concepts of Generative Programming are explained which are needed to understand GPI

First a general description of Generative Programming is given along with Quaestio 5 seen as an example of Generative Programming. Quaestio 5 is an earlier version of the program that in Part 3 of this paper will be given as an example of a web application development environment that supports GPI. Next the notion of a “Generative Domain Model” which lies at the heart of Generative Programming is explained. Subsequently we describe several concepts needed in the development of such a Generative Domain Model.

At the time of this writing, the standard reference on GP is the already mentioned monograph by Czarnecki and Eisenecker entitled: *Generative Programming, methods, tools and applications* [2]. In addition you can also consult the following websites: <http://www.generative-programming.org> and <http://gp.uwaterloo.ca>.

1.1 What is Generative Programming?

Generative Programming is a software methodology that focuses on the development of a family of systems. Instead of creating a single system from scratch, a so-called “Generative Domain Model” is developed from which particular systems in the family can be generated.

A Generative Domain Model consists of three parts:

1. A means of specifying a family member,
2. Implementation components from which family members can be assembled, and
3. Configuration knowledge that maps the specification of a family member to a finished member.

The terminology used to specify family members is referred to as “problem space” and the implementation components with their possible combinations form the “solution space”. The configuration knowledge maps family member specifications in the problem space to complete family members assembled out of the implementation components in the solution space.

This model is analogous to ordering a car. Nowadays no two cars leaving a car factory are exactly the same; there is a plethora of configurations according to which cars can be ordered. In such a car ordering and production system you can find the three main components of a Generative Domain Model.

1. There is a system for ordering cars in which people can specify what specific features the car should have.
2. There are (implementation) components from which cars in the car factory are assembled
3. The car factory is engineered to hold the configuration knowledge of how to assemble a car to a given order.

Note that the people who use the car ordering system don't have exact knowledge about how the car is assembled at the level of the implementation components. Only the engineers that programmed the configuration knowledge into the car factory have that knowledge.

1.2 The Generative Domain Model (GDM)

The diagram below shows the elements a Generative Domain Model.

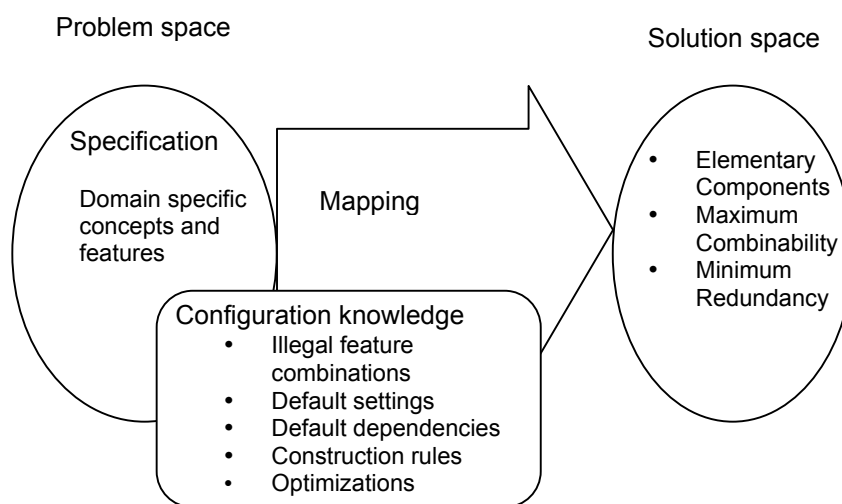


Figure 1, Generative Domain Model

The problem space is thus formed out of domain specific concepts and features forming a domain specific language with which all single systems in the targeted family can be described. When specifying single systems is accomplished by using an application, it will probably use the configuration knowledge to prevent illegal feature combinations and probably will show the default settings and dependencies.

The solution space consists out of elementary implementation components. These can be combined in any way and implement functionality without much redundancy.

The mapping between the two spaces is implemented as a generator that can transform the specification using the concepts in the problem space in an end product assembled from the components in the solution space. This generator normally uses the same configuration knowledge to check the specification and constructs an optimized end product as described in the specification. This generator does not have to be very complex if the concepts in the problem space can be easily mapped to the elementary components.

By way of an example it is clear that a C++ compiler can also be viewed as a mapping of a Generative Domain Model for the family of all computer programs. In this model the problem space is formed out of all possible descriptions of programs in the C++ language and the solution space is formed out of all these programs assembled from assembly codes that are the elementary implementation “components”. The C++ language is not very domain specific however. Thus only trained programmers can take advantage of this generator.

In Generative Programming, a much more specific language is targeted using the concepts of the customer for whom family members are built, even to the extent that it becomes possible to generate similar family members without much effort by the people who need to engineer the configuration knowledge.

Another interesting thing to point out in the C++ analogy is that the mapping from descriptions of family members to family members does not need to be a single step. C++ code is preprocessed, compiled to intermediate code and finally to machine code [8]. Each compiler step can be modeled as a Generative Domain Model and chained together so that the composition they can form one too; mapping from the highest level to the lowest level Domain Specific Language being C++ language and the machine code respectively. It is this insight among others that will lead to the idea of Generative Programming through Inheritance.

1.3 Generative Programming Steps

To use the Generative Programming approach, one needs to develop and implement the Generative Domain Model for the domain that is formed by the family of systems that is targeted.

Development of a Generative Domain Model usually involves the following steps:

- Domain Engineering:
 - Domain scoping
 - Feature modeling
- Design Generative Domain Model
- Implement Generative Domain Model
- Create single systems with the Generative Domain Model

We will now look more closely at each of the above steps.

1.3.1 Domain Scoping

The first and very important step in Domain Engineering is the determination of the scope of a domain. That is, what the boundaries of the targeted family of systems are. A domain can be defined as an area of knowledge

- Scoped to maximize the satisfaction of the requirements of its stakeholders
- Includes a set of concepts and terminology understood by practitioners in that area
- Includes the knowledge of how to build software systems (or parts of software systems) in that area.

Which domain of knowledge to scope is influenced by several factors like:

- Stability / Maturity of the candidate area to become part of the domain
- The available resources for performing Domain Engineering
- The potential to reuse the results of Domain Engineering within or outside the organization

To ensure successful business a domain should be selected that strikes a healthy balance among these factors.

It is important to note that knowledge domains can be infinite and inconsistent. Modeling it to the full detail is doomed to fail. For this, begin with small domains and make sure refining it and defining alternatives is easy. Domain engineering is something that has to be maintained also since knowledge can change over time.

In the end a set of concepts and knowledge around them that together define the scoped domain should have been elicited. All kinds of activities can be used to do this but this is not further explained in this paper except Feature Modeling that is described next.

1.3.2 Feature Modeling

An important step in domain engineering when using Generative Programming is Feature modeling. Using so-called feature models, commonalities and variability's of features of concepts in the scoped domain can be expressed through restrictions like concept A has either feature B or feature C. This can relate different concepts with their features in a more formal way. Feature modeling is a creative process in which new combinations of concepts and features are elicited.

A concept is something else than an OO-class. An OO class has predefined semantics whilst a concept doesn't need to have one. We can think of concepts as "reference points" in the brain for classifying phenomena. A concept is a class of phenomena. We can talk about them without having to list all properties, which is in most cases impossible anyway: try for instance to give an exhaustive definition of the concept "table" that uniquely identifies the concept.

By modeling these concepts it is possible to express commonalities and variability's not only in classes, objects but also in use cases, functions, procedures and so on. This

is why UML is currently not really suited for this purpose because not all concepts are objects that have identity and have a state. Concepts are inherently surjective: their information content depends not only on the person using it, but also on time, context and other factors. UML could be extended or one could use UML in a particular way to do feature modeling however.

One could use abstract classes along with some way of defining optional relations and or-relations between relations for this. Generative Programming as introduced by Czarnecki and Eisenecker uses a slightly adapted and extended version of the FODA feature diagram for this however like in the picture below.

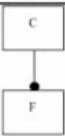
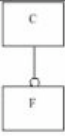
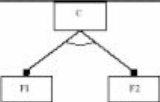
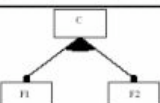
Type	Notation
Mandatory	
Optional	
Alternative	
Or	

Figure 2, Concept relations in feature models

1.3.3 GDM Development

The development of the generative domain model once the problem space is modeled consists out finding the right set of elementary components (in some programming language for instance) such that the concepts and features in the feature models can be easily mapped to the elementary components. Some existing techniques are more suitable for this than others. Some of these are:

- Component Based Software Engineering: descriptions of a solution in the problem space, which is modeled using feature models, are more easily mapped to components that represent the concepts and features in the models.
- Generic Programming: Concepts are often generic, so there will be a need of generic components that can be used in all kinds of contexts.
- Aspect Oriented Programming: some concepts will not refer to a real object but more to aspects of objects. There will be a need to encapsulate the implementation of certain aspects in single components to be used with the components that represent the objects that have these aspects.

If all concepts in the problem space can be mapped to single elementary components, than mapping problem space to solution space is easy. It will often however not be possible to do this without defining fat, complex and non-reusable components. For this, a generator is needed that maps the concepts in the problem space to components of a set of real elementary components which together implement functionality that is minimally redundant and can be combined with each other in as much ways as possible.

When defining a generator, some form of meta-programming is needed, translating the solution in problem space to a solution, possibly consisting out of some programming language defining the solution space. The ultimate goal is to create a generator that compiles a description, non-programmers can understand into some programming language from where it can be compiled to the real solution. Several ways to create such generators as listed by Czarnecki and Eisenecker are:

- Cooperating generators from scratch (costly)
- Use of meta-programming facilities such as C++ template meta-programming
- Extend the programming Environment: Microsoft Intentional Programming

In the next part another new approach to create such generators is introduced called Generative Programming through Inheritance. But first we'll look at another example of Generative Programming that resulted in the idea of Generative Programming through Inheritance.

1.4 Another example: Quaestio

An example of Generative Programming is Quaestio [6]. Quaestio is a software product developed by Cope that is able to give a simple interface called QDesign to non-programmers as well as programmers with which a web survey can be specified. During specification the web survey can be previewed and with one single action, the web survey can be “generated” by deploying the specification to a Quaestio Server. The web survey can be immediately used at that moment.

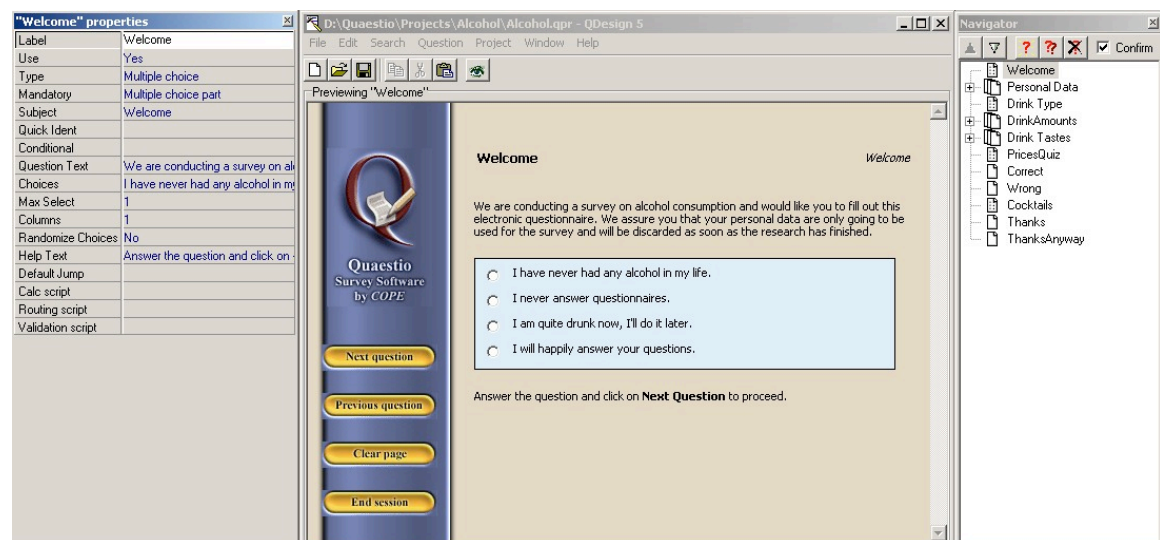


Figure 3, QDesign screenshot

Although Quaestio is not developed with the methodologies described in this paper in mind, it can be seen as an example of Generative Programming. It has QDesign as means to specify family members, namely web surveys as single systems. Quaestio Server consists in essence out of implementation components out of which all the web surveys are assembled and both Quaestio Server as QDesign contain the same configuration knowledge that maps from a specification to the resulting web survey.

In Quaestio, the domain of all web surveys initially and basically was described as follows: A web survey is a list of questions and the questions in the list are presented one by one when the survey is being answered by a respondent. These questions further can have all kinds of properties that can affect layout and working of the questionnaire, which we'll not explain in this example. In QDesign a non-technical stakeholder can specify a web survey by simply creating a list of question-components and specify values of their properties (parameters) if their default values are not good enough. This specification is then sent to Quaestio Server in which the concept "question" in the specification is mapped to a component implementing all functionality associated with that concept in the server which consists out of the code that generates a web form showing a question and a input box in which the respondent can enter his answer as well as a form handler that validates and stores the submitted answer and forwards to the next question.

The problem space here is formed out of the concepts and features that can be used in QDesign to create a specification of web surveys and the solution space are the components web surveys are assembled out of within Quaestio Server. The main two concepts that can be used here are survey and question.

Part 2: Generative Programming through Inheritance (GPI)

In a later version of Quaestio introduced at the end of Part 1 when seeing it as a form of Generative Programming, the problem and solution spaces were greatly extended by allowing programmers to enter PHP [5] scripts as properties of questions that affect the working of the survey, possibly to the point you can't speak of a web survey anymore. That is because PHP is not domain specific. This resulted in that Quaestio is being used for all kinds of data-entry web applications and this and the fact that the Generative Domain Model of Quaestio is implemented in such a way it isn't easily refined or extended leads to the choice of rebuilding Quaestio from scratch.

When thinking about this rebuild we realized that the domain that Quaestio could address, the domain of web surveys, needed to be extended. We also wanted to keep the same easy interface to non-programmers for specifying family members in this whole new extended domain as the interface for specifying web surveys. But also we realized that the domain we wanted Quaestio to address was too big to implement the configuration knowledge for at forehand. It therefore was needed that we could introduce at any time new domain specific concepts and features, the non-programmers can understand and for only the part of the problem space that is needed; see Domain Scoping in Part 1.

These concepts and features then could be used in the same manner as the concepts "survey" and "question" in the old Quaestio. Also these concepts will often refer to similar functionality. This gives possibilities for reuse and to make reuse simple for (not very advanced) programmers, it should be possible to create a mapping from these concepts to some intermediate Domain Specific Language in the same interface and in a similar way of creating surveys.

The result is Quaestio 6 which is presented in part 3 of this paper for which intuitively the concept of Generative Programming through Inheritance was defined to find out only later this was in essence a way of Generative Programming which helped in refining the approach. With Quaestio 6 you can model the problem space through feature modeling and then fill in the implementation reusing functionality at a lower and more technical level that is created in a similar way. This resembles somewhat to an IDE for Executable UML in which you can model domains and then fill in implementation code that links to executable UML in other lower level domains [7].

Quaestio 6 is an implementation of a meta-generative domain model in which any generative domain model for any family of systems being a subset of the family of web applications can be created. Using Generative Programming through Inheritance, new Generative Domain Models can be created from existing ones almost in a same manner as specifying single systems in a particular Generative Domain Model.

In this part Generative Programming through Inheritance is introduced. With Generative Programming through Inheritance, new problem spaces with new concepts can be modeled and generators can be created by specifying a "Is a" relation from these new concepts to "uses" of existing concepts which are part of other lower level problem spaces created earlier. Since for these existing concepts such generators already exist that map them to elementary components, the new concept is then mapped to elementary components. The uses of existing concepts moreover can

contain executable code and thus great flexibility is possible in mapping the features of the new concept to features of existing concepts. This mapping forms a step in the bottom up generation of a specification in the new problem space to the solution space.

To explain Generative Programming through Inheritance (GPI), in the next sections the following concepts are defined and explained in the context of GPI: what parameters and components are, how component frameworks can be used as Domain Specific Languages and how the special “Is a” relation forms a meta-programming step linking one problem space to an existing other one and thus can link problem spaces with a solution space. In a way, this “Is a” relation introduces a form of inheritance in this component based approach and thus properties linked to it like encapsulation and polymorphism are looked at.

2.1 Components and Parameters

Components in GPI are a set of parameters representing concepts that have features.

A parameter thus represents a feature of the concept represented by the component it is part of and has a name that should be part of the jargon used by the most non-technical stakeholder that uses it directly. A parameter is not only a tag but can also contain a value of the following types:

- A string of any length (this can represent numeric values too)
- A binary resource (stored separately from the component)
- A reference to one or more other components
- A script that evaluates the value of the parameter on request.

The values that are stored with the components should be seen as default values for the parameters in each “use” of the component the parameter is part of.

A parameter is not typed; it can always contain a value of any of the types above. Parameters further more can store any kind of meta-information about the parameter. With this meta-information the values that can be stored in the parameter may be

restricted or specified through a special interface made specifically for the component at design time with which the development environment is extended. In execution time these restrictions are not evaluated however except the following meta-properties which are always possible:

- Static: a parameter can be specified to be static in which case the parameter always has the default value as value in runtime. Otherwise, some scripts may put other values in the parameter for the duration of the use of the component it is part of.
- Encapsulated: when a component (the concept it represents) is being made part of a domain specific language, the component is made available for reuse from a central repository. With this action, the parameters that are marked “encapsulated” should be encapsulated away from the user that uses the

component from the repository, that is, the user won't see those parameters and cannot change them.

- Scope: the scope of a parameter may restrict which scripts may access it. The scope can be public, private or protected. When public every script of any component that can reference the component the parameter is part of can access it. When private, only scripts in the component the parameter is part of itself can access it. When a parameter has the protected scope it has private scope but also the scripts in all the components that can be found by following the references in the "Is a" parameter can access it if called in context (explained later on).
- For references to other components as value the parameter may contain reference settings as explained in the next subsection.

There are certain parameters that define the identity of a component and so all components have at least the following parameters:

- A static parameter that uniquely defines the component in the world
- A static parameter that gives a human readable name to the component which is exactly the word used for the concept the component represents and which is part of the jargon used by the most non-technical stakeholder that uses the component directly.
- A static parameter that indicates the domain specific language the concept the component represents is part of.
- A static parameter that holds versioning information for this component

Components further can have the special static "Is a" parameter that refers to a set of parameterized components that actually form it's implementation which is explained later on.

2.2 Atomized components

In Generative Programming through Inheritance, one step is to define new components by specifying the set of parameters and their default values. Another step is to use these defined components in specifications in which the non-static and non-encapsulated parameters may get different values.

In Quaestio 6, which is presented in Part 3, a meta-component called a Quaestio component is defined with which a new component can be defined. These new components can then be atomized. When atomizing a component, the component is stored in a component repository as being part of the domain specific language it is defined to be part of. When using an atomized component from this repository, the component stored is copied but will get it's own identity apart from the one in the repository.

While atomized, its set of parameters cannot be altered and encapsulated parameters should not be made visible to the developer and also static parameters form constants the developer can see.

A developer may however be given the right to de-atomize a component at any time, in which case the component will be handled and shown as a Quaestio Component. Note that the component when de-atomized while it is part of a specification will still function in the same way as it did before until a developer changes the set of parameters or values of static or encapsulated parameters. Like any Quaestio Component, de-atomized components may be atomized again. In this case a newer version of an existing component may be created if the developer did not change the values of the name and domain specific language parameters of the component.

2.3 Feature modeling restricted references as parameter value

Parameters may be references to multiple other components. Since components represent concepts and parameters represent features, it is naturally to be able to restrict the components that can be referred to in parameters as like is done in feature modeling. According to Czarnecki and Eisenecker, Feature models should not include cardinality on relations. This is because these models should not contain such structural information since this gets in the way of the elicitation process. For example: why restrict a model of a car to have a specific amount of wheels while any amount might be possible at a certain time. The amount of wheels should be a variable in generic models of a car.

The mandatory and optional relation in feature models however does add cardinality information to the relations in the model since they represent the cardinalities 1..1 and 0..1! The alternative relation can be seen as a relation from a single component to one of a set of components thus with the cardinality 1..1.

Moreover in feature models the or-relation forms a relation to components from a set of components with the 1..* cardinality. It is intuitive to make optional versions too resulting in the same but with a 0..* cardinality.

In Generative Programming though inheritance a reference can be restricted to reference to only certain other “allowed” components with the 4 possible cardinalities: 0..1, 1..1, 0..* and 1..*. An explicit number of references, if needed, should be modeled as a specific number of parameters with references that have cardinality 1..1.

In GPI one can thus actually model domains using new components as like in feature models and then go on with mapping the new problem space that is modeled to the solution space with which a new generative domain model is defined.

2.4 Component Frameworks as Domain Specific Language

With components that have parameters with which one component may reference the other, graphs of parameterized components can be constructed. The set of components and the referencing rules that are specified in them can be seen as a Domain Specific Language [3], and the graphs created with them form specifications in the problem space the domain specific language forms.

In practice a lot of these domain specific languages can be defined. Some are more suitable for non-technical stakeholders whilst some are very technical. In other words, some are in a higher level than others and at the same time more towards the problem space than towards the solution space.

The key in Generative Programming is that specifications in high level problem spaces should be mapped to specifications in the lowest level solution space [2]. In GPI this can be done by mapping single components from high level specifications to a specification in a lower level through the special “Is a” parameter. Therefore the high level component represents the use of a lower level solution in a sub problem space and which actually forms its implementation. Moreover scripts in other components that can reference to the high level component may access components of its lower level implementation directly in the context of the higher-level component. This can even be done by following multiple “Is a” references too. This is a form of inheritance.

2.5 The inheritance relation “Is a” as a generator step

In literature, it is already mentioned that there exists a special “Is a” relation within component based programming approaches which has something to do with inheritance [1]. The “Is a” relation in GPI between components is such a relation and maps a high level concept in a problem space to a specification in a sub problem space. By first creating domain specific languages for the solution space (elementary components) and then for a sub problem space in which the single components are mapped to solutions in the solution space and repeating this several times, each time at a higher level, one can map specifications in the problem space the most non-technical stakeholder can directly use to solutions in solution space consisting out of the most technical elementary components.

This mapping does not represent a simple replacement operation when the application formed is compiled or executed. The “Is a” parameter maps to “uses” of components, thus to copies of components that can have any values for its non-static and not encapsulated parameters specifically set for this context. These values moreover can be scripts that determine the right value in runtime and these scripts can access the protected parameters of the high level component too. This means that the values set for the parameters in the higher level component can be used in the generation from the higher level component towards the solution referenced by the “Is a” parameter. Furthermore, scripts in components that can reference the high level component by following one or more references, can actually access it as being a lower level component. That is because the high level component may also be referenced by references in parameters that are set to reference some lower level component that exists in the graph consisting of only “Is a” relations which is under the “Is a” reference of the higher level component. The component that is referenced in this way is actually the lower level component, but within the context of the high level component. The scripts in the parameterization of the lower level component may access the public and protected parameters of the components in the path consisting out of only “Is a” references towards the high level component that was actually referenced.

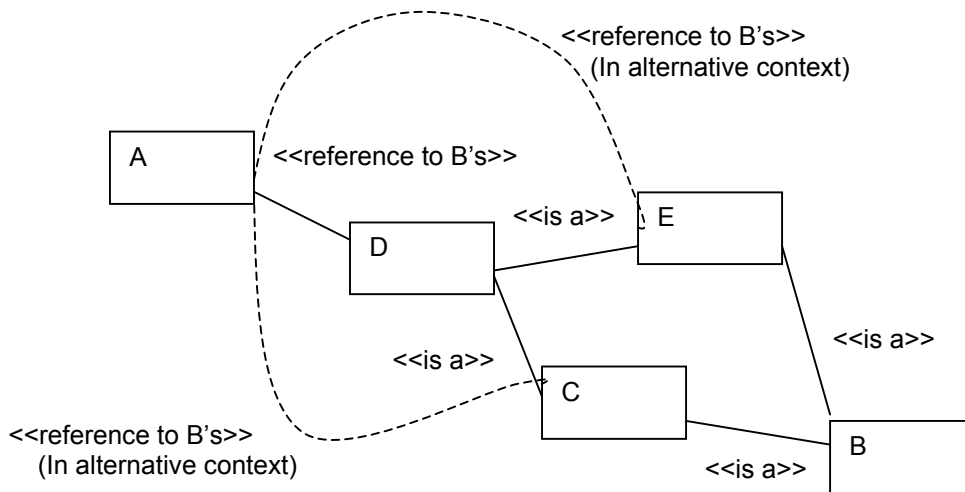


Figure 3.

In other words when looking at figure 3: when a component A can reference in a certain parameter to components B, and a component D references in its “Is a” parameter to a component C which in turn references in its “Is a” parameter to a component B, then component D may be referenced by A. This is polymorphism. When a script follows the reference to D in A, it will automatically get however the component B in the graph under the “Is a” relation of component D, because of the reference settings in A. This inhibits a special breadth first search which results in the rule that certain loops in following the references in only “Is a” parameters are disallowed. The scripts in the parameterization of component B however may access component C and D through a predefined function. In the case that the use of component B here is referenced in the “Is a” parameter of multiple components (not only C, but let’s say for instance component E too), the predefined function returns component C or E depending on how component B is accessed: either A referencing D, C or E. If component B is referenced directly by A, this predefined function returns nothing.

In the case above, component D can be seen as a child component of B. D can augment the working of component B and can be used in any place component B can be referenced. When a component B may be referenced scripts can rely on the fact it will actually get components B when following the reference, also in the case other components are actually referenced, which are specializations of component B.

The “Is a” parameter may reference to any number of any components. If A would reference to two uses of B, when accessing the reference, both B’s are returned as a set. The breadth first search thus doesn’t stop until every B component is found. If in the example above, component D also references in its “Is a” parameter to component E, when finding component B, two paths towards B can be found: one through C and one through E. In this case, the same component is returned twice as a set, but each

result as being in a different context: either the path through C or E. The predefined function thus returns either C or E depending on which version of B is used. A script may also explicitly state the context path to be used when accessing the reference in A. The breadth first search will then start at the deepest component in that path instead of D. In this case, when no B is found, no components will be returned to the script. When no explicit context path is given, a component B will always be found because otherwise D should not have been allowed to be referenced by A.

As stated before, the breadth first search following only “Is a” references should not loop endlessly. Therefore either the search algorithm should stop searching when a component is found that “has been seen before” or loops in the graph consisting of only “Is a” references should just be disallowed. This last solution is even better because this rule is actually what creates the direction of generating a high level specification in problem space to a set of cooperating specifications in solution space.

2.6 Bottom-up generation

Generation in GPI is started and facilitated by a server or interpreter that implements the predefined function and the breadth first search and some other core functionalities. Execution of a solution created with GPI begins by a most elementary concept “Job” which is represented by one of the few components the server can work with. This concept has a feature “action” represented by a parameter, which should contain a script that forms a main function returning the result of the job. This is analogous to a main function in programming languages.

When the server starts this component directly, the predefined function will return nothing because the “Job” component was not started in context of higher-level components. Therefore the script in the action parameter can only access components in this most elementary domain specific language. In GPI a lowest level component such as “Job” normally is always executed in context of the path from a top-level component to the “Job” component.

In Quaestio 6, which includes a specific implementation of the server mentioned above, a solution always has one root level component representing the project in which a set of solutions represented by single components is contained. The server will start executing by accessing each solution as being a “Job” component. Let’s say the references from the server to the solution components are set such that only “Job” components may be referenced. The server will perform it’s breadth first search in this case looking for job components in the graph consisting out of “Is a” parameters under the solution component. All job components found are executed in this context and the parameterizations of the job components will contain scripts that access the higher-level components in the context, which in turn access even higher-level components etcetera. Note that the parameterization of the Job component is done when defining the next level domain specific language and thus it can use next level components directly in its script, which in turn can use the components in the level above etc. This way generation can be seen to take place in a bottom up way.

2.7 Encapsulation and Polymorphism

The components and their parameterizations (parameter configuration with scripts etc.) that are in the “Is a” relation of a child component can and should be encapsulated as being the implementation of the child component. Two uses of the same component cannot have different values in the static and encapsulated parameters. If a parameter is encapsulated and refers to other components, the two uses may refer to the same component uses.

Whole applications may be encapsulated in this way so only higher-level building blocks stay which can be used directly by less technical users. Looking at the analogy with the car factory mentioned in Part 1, with GPI you can have a ordering system in which a customer can configure her/his own car with building blocks that are defined to consist out of a assembly of the most elementary parts directly in the same ordering system by car engineers. This level of detail is encapsulated for the customer. Also the car factory belonging to the ordering system directly knows how to build the car from the description made by customer and car engineer in the ordering system.

In the section about the “Is a” relation above, it is already made clear that polymorphism is an important feature of GPI. This is what enables the use of new components with components that are not created with the new component in mind. This is illustrated in the example given in Part 3.

2.8 GPI compared to other technologies

It is already mentioned that some technologies like component-based software engineering and forms of meta-programming are more suitable for implementing the mapping in the Generative Domain Model. Generative Programming through Inheritance is a component based programming method that reintroduces inheritance, a feature of Object Orientation (OO) to allow Meta Programming.

It is hard too compare inheritance in GPI and in OO. In GPI, when creating a child from a parent component, the parent component must be seen as a template that is configured. This configuration can be made dynamic by using scripts stored with the child component. With these scripts the configuration of the parent component can be made dependent on the configuration of the child component. Moreover, multiple parent components can be configured too work together, and the application formed by these cooperating components can be configured through the child component. Dependent on the reference to the child component, either the child is accessed directly or one of it's possibly dynamically configured parents within the context of the child and all its parents. This means all public features of the parent component are inherited and probably overridden public features when accessing the child as a specialized parent component but they are automatically protected features when accessing the child without knowing its parent. This is not the case in OO in which all of the public properties and methods of the parent are automatically public whether accessing the child as an object of a specialized parent class or directly without knowing its parent type. There may be ways how to do GPI with OO, but it will not be that obvious why to do this. GPI is not only about inheritance, it is also about

generative programming using cooperating components, being bigger units than objects usually are.

In comparison to component-based methods in which component interfaces are glued together with code (often components too) and the implementation of components is engineered specific for these interfaces, in GPI there are only interfaces that are “glued” together with code. Everything under the “Is a” relation forms the implementation of a component and this is normally encapsulated away. In this way, layers of cooperating component frameworks emerge automatically.

A technology that has similarities with GPI is Executable UML[7] which is about domain scoping at different levels and creating cooperating components between these domains directly in the UML model too like GPI which allows to add code to feature models which according to Czarnecki and Eisenecker are more close to how humans think than what is normally modeled in UML.

GPI like Executable UML needs a special IDE since programming is partly done in visualized structures like diagrams and trees and word processors are not suitable for this. In the next Part we’ll look at an IDE for GPI called Quaestio 6 that also allows extending the programming environment; another already mentioned technique suitable for Generative Programming.

Part 3: Example Web application Development Environment supporting GPI

In this part we look at how GPI as explained in Part 2 could be used by giving an example and introduce a web application development environment supporting GPI that is currently in development.

3.1 Example: Quaestio recreated using GPI

In this section, the Quaestio example given in Part 1 is used to illustrate how to use the theory in Part 2. In the diagram below several problem spaces and corresponding component frameworks defined and connected using GPI are showed in a very simplified example of a single system created out of them consisting of a survey containing only one question.

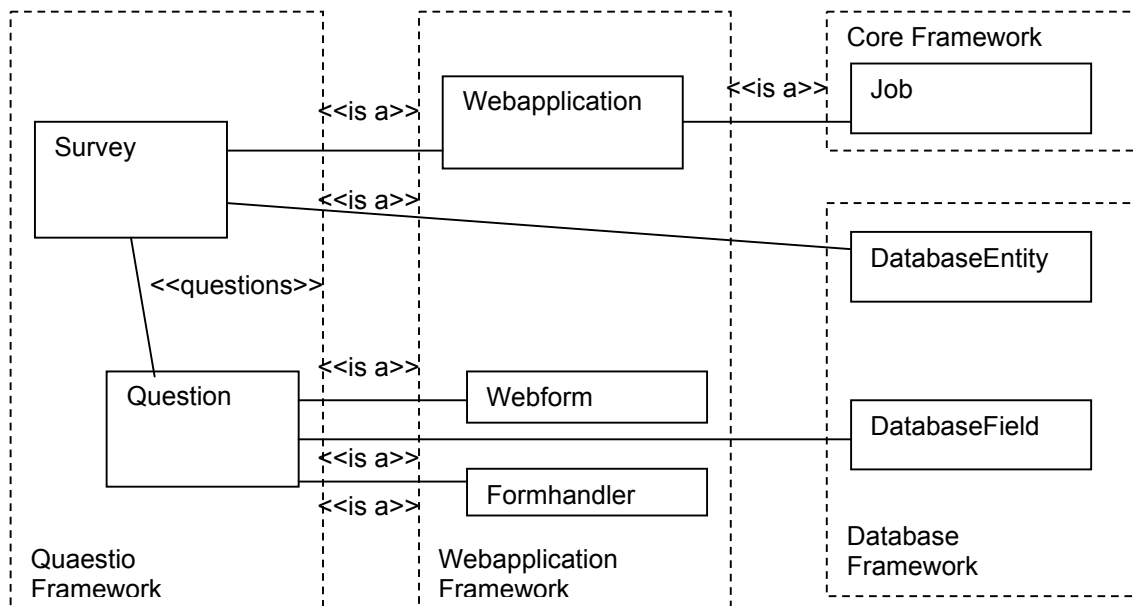


Figure 4

When the application above is started by an end user that wants to answer the questionnaire, the Survey Component is started as being a Job Component and thus a certain parameter of the Job Component from the Core Framework is executed which is parameterized when the Webapplication Component was defined and certainly will retrieve values from the parameters of the Webapplication component which on its turn is parameterized in a way it will use the parameters of the Survey component and so on.

The Webapplication Component (and thus the Survey Component) is the component for accessing a web application and might provide things as session support. The Webapplication Component will be defined such that each time a page is requested through a GET HTTP request a Webform Component is used to get the contents of a page to show and that when a certain POST HTTP request is done, a Formhandler Component is used to handle the submitted data.

At first the Webapplication Component as implementation of the Survey Component will be parameterized with a script that will find the first Question Component referenced in the parameter “questions” of the Survey Component and will call it as a Webform Component. This component will be parameterized such that the settings in the Question Component are translated to a HTML form showing the question and input element as well as a submit button. The HTML Form returned will be set to post again to the server in reference to the Survey Component as a Job (the only component the server interacts with). Again the Webapplication Component will be used but now it will detect posted data and will find and use the Formhandler Component that should handle the submitted data in a same way the Webform Component was found and used.

The specific interpretation of a Formhandler here will have a parameter that should reference to a Database Entity to store all submitted data in. Of course submitted data

may be validated first etc but this is left out of this example. For this the Formhandler Component got a reference to the Survey Component from the Webapplication Component. The Formhandler will use the Survey Component as a Database Entity which will be parameterized specifically for the Survey component to use the Question Components coupled to the Survey Component as DatabaseField Components which will contain settings for the fields the Formhandler can store the submitted data in. After handling the data, the Formhandler will redirect the web browser of the end user again to do a GET http request to the Survey Component.

This time the Webapplication Component will look for a next question. In this case it is not found and the end of the questionnaire is reached and thus a Javascript pop-up box will be put out to thank the user for answering the survey after which the user is redirected to some default page. Now the database will contain an anonymous answering of this single question questionnaire.

The frameworks above can be developed apart from each other in an evolving way. The Webapplication, Core and Database framework are very generic and can be reused in several kinds of applications. Not only for web surveys. Moreover the Quaestio Framework can be used directly in an encapsulated way to less technical end users that want to create web surveys but don't know how to program. They will only see the components survey and question and will only be able to make a list of questions in the "questions" parameter of the Survey Component representing the questions to be answered by respondents.

Further these questionnaires, expressed using Components from the Quaestio Framework may be reused when the whole implementation of the framework is changed for instance to include the possibility for generating the surveys to paper surveys thus to PDF files instead of a web survey.

In the next section a new development environment is introduced in which component frameworks as above can be defined and used using GPI by programmers together with less technical users.

3.2 Quaestio 6

Quaestio 6 is a web application environment that allows a developer to define new domain specific languages consisting of component frameworks with which solutions can be expressed in (sub) problem spaces. It furthermore allows through the use of GPI to map such (sub) problem spaces to lower level spaces and finally to the solution space as shown at the end of Part 2. In this way, a developer is able to create all kinds of generative domain models and generate different single systems from those generative domain models using the same development environment.

By encapsulating components in lower level problem spaces and restricting access to them, less-technical developers can construct solutions expressed in the problem space at the level they understand; solutions constructed out of the simple to use building blocks that are provided for them. For this Quaestio 6 also allows creating special interfaces for components and thus extending the development environment where needed: a second meta-programming technique besides GPI.

Quaestio 6 exists out of a ProjectBuilder and Quaestio Server. ProjectBuilder is the program in which problem spaces, generators and elementary components can be defined and tested. ProjectBuilder can also be used as “ordering system” for non-programmers at the same time. Quaestio Server can execute solutions made with ProjectBuilder on a web server.

The ProjectBuilder main window sketched in Figure 5 shows a component framework repository, a project layout, a pane in which component parameters can be set and a preview window in which solutions can be tested. From ProjectBuilder tested projects are easily deployed to Quaestio Server.

Quaestio 6 is at a whole different level than its predecessors. Quaestio given as an example in Part 1 can be recreated in Quaestio 6 and in theory can be extended to the problem space of all web applications.

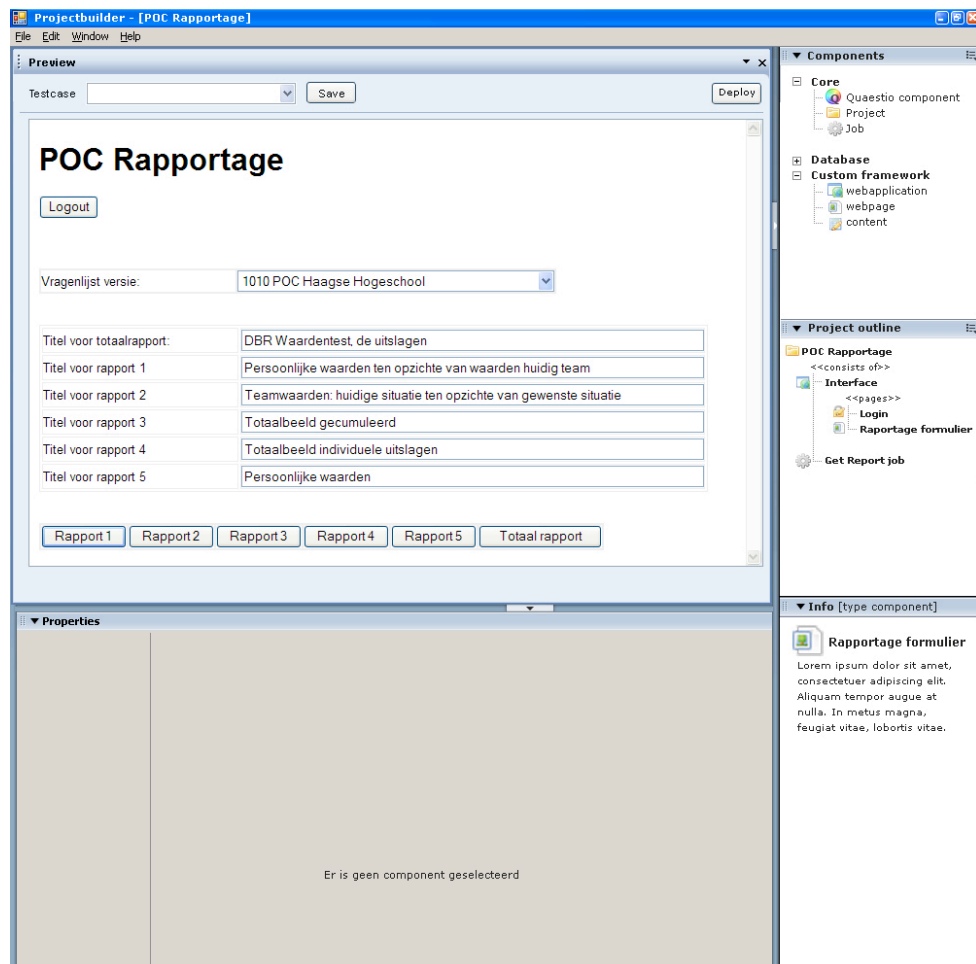


Figure 5

Conclusions

Generative Programming is an emerging software methodology promising great advances in the amount of reuse within software development. By defining a generative domain model, different solutions can be expressed in a simple language targeted for use by non-programmers. GPI is a component-based technique for defining the generators that form the mapping from problem space to solution space in the Generative Domain Model.

GPI uses component frameworks for defining DSLs in which solutions are expressed within the Generative Domain Models. By introducing a special inheritance relation between components, generators can be defined that conceptually map parametrizable instances of concepts in a higher level DSL towards the problem space to parameterized concepts in a lower level DSL towards the solution space. Through the use of encapsulation each level from problem space to solution space can be separated and hidden. This enables the creation of an integrated development environment in which both programmers as well non-programmers can work. Such a development environment is Quaestio 6 currently in development at Cope.

Outlook

There is not a functional prototype of Quaestio 6 at the moment. Only the basics of the server part are almost ready. The added value of GPI to software development can only be evaluated when it has been used in software development projects several times [4]. Because not only the added value of a software methodology needs to be proven in practice, but also because generative programming and therefore GPI targets the development of generative domain models that can be used in multiple projects.

Upon completion of the prototype we will conduct further research and continue with the testing of GPI.

Further research will probably be necessary to make GPI more effective in software development. For instance, it is expected that certain design patterns will be found that are more effective than others. It is also expected that some operations, that scripts are allowed to perform, should not be allowed to prevent problems that make reuse impossible.

It is however already a success if the Generative Domain Models of earlier Quaestio versions can be recreated and used in Quaestio 6. Instructions on how to create this in Quaestio 6 are already available. The goal is to have a working prototype by the end of 2007.

References

- [1] Component Software – beyond Object Oriented Programming 2nd edition.
C. Szyperski. Addison-Wesley / ACM Press, 2002
- [2] Generative Programming - Methods, Techniques and Applications
Krzysztof Czarnecki and Ulrich W. Eisenecker. Addison-Wesley 2000
- [3] Domain-Specific Languages: An Annotated Bibliography. Arie v. Deursen, Paul
Klint, Joost Visser ACM SIGPLAN Notices, 35(6):26-36, June 2000
- [4] Component Oriented Software Development, Oscar Nierstrasz, Simon Gibbs and
Dennis Tsichritzis. September 1992/Vol.35, No, 9 / COMMUNICATIONS OF
THE ACM
- [5] PHP <http://www.php.net>
- [6] Quaestio <http://www.quaestio.com>
- [7] Executable UML, A Foundation for Model-Driven Architecture, Stephen J.
Mellor, Marc J. Balcer, Addison-Wesley 2002
- [8] Compilers: Principles, Techniques, and Tools, A.V. Aho, R. Sethi, en J.D.
Ullman, Addison-Wesley, 1986
- [9] Cope <http://www.cope.nl>