Towards a Model for Specifying and Composing Concerns in Software Product Line Engineering

Volker Kuttruff
FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10-14
76131 Karlsruhe
Germany
kuttruff@fzi.de

Abstract: In order to fulfil sets of similar user requirements within a specific application domain, one typically uses software product line engineering. In this paper, we investigate the nature of implementations of concerns, specific to software product line engineering. Based on these investigations, we present an approach that allows a modular specification and composition of concerns, with the purpose of constructing concrete variants of a software product line. The approach uses concepts from generic and aspect-oriented programming, and adapts them to the requirements imposed by software product line engineering.

1 Introduction

In order to fulfil sets of similar user requirements within a specific application domain, one typically uses software product line engineering (SPLE) [CE00]. The different software programs producible using a software product line (SPL), differ by the presence or absence of specific program features, visible to the customer. Since the explicit specification of each variant of an SPL (implementing a desired combination of features) results in maintenance problems due to redundancies in the implementation, the producer of such an SPL is interested in composing the variants from as few software artifacts as possible. This can be achieved if the implementation of different features (concerns) can be encapsulated in separate modules, and if these concern implementations can be composed in a modular way.

This leads us to the principle of *Separation of Concerns* (SoC) [Dij82], which is one of the fundamental principles in Software Engineering. The term *concern* denotes a particular piece of interest or focus in a program. Since this piece of interest usually depends on the perspective of a stakeholder of a program, the definition of the term concern is heavily overloaded in literature.

In this paper, we will focus on an implementation-centric view. For the case of an object-oriented program, figure 1 shows how concerns are expressed at different levels of abstraction. At the level of the application domain, concerns correspond to the features found during requirements engineering. A concern can either be a functional concern,

defined by a group of functional requirements (e.g. computation of a value), or a non-functional concern, defining interactions between functional concerns (e.g. extendability). On this level, all concerns are modular entities, thus SoC is achieved. In the design phase, some concerns are still modular entities, while other concerns are scattered among design entities. Furthermore, some concerns disappear as concrete entities, e.g. performance or extendability. These concerns are usually realized through the design of the program or the selection of appropriate algorithms and data structures. Finally, at implementation level, a concern can be understood as the implementation of a single facet of the functionality of a program, expressed by a set of program fragments entangled with the program fragments of other concerns.

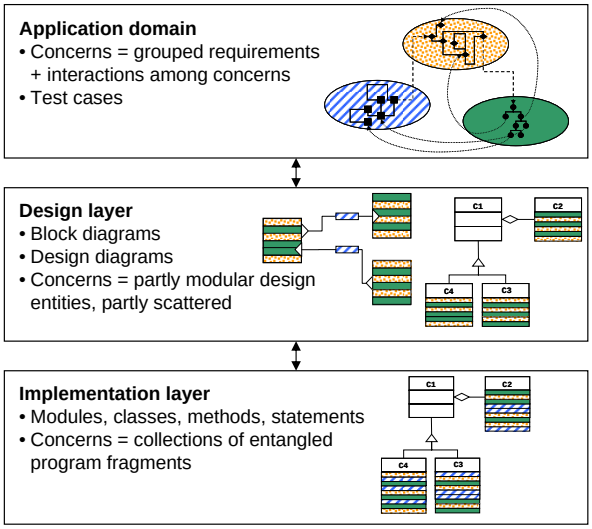

Figure 1: Concerns on different program abstraction layers [Gen04]

As indicated in figure 1, the refinement of a program during design and implementation has two effects: formerly direct concern interactions may become indirect (e.g. passing values using a database, which cannot be observed directly), and concerns tend to be less modular (scattered and tangled with other concerns), or may even disappear as expressible entities. This contradicts the principle of SoC, because SoC implies that in order to build the required program, concerns should be specified and composed in a modular way.

SoC has been and probably still is the most important driving force in the creation of programming paradigms and techniques, aiming to provide possibilities to capture concerns on the design and implementation level in a modular way. With the help of procedural languages, some concerns can be separated into procedures, while the object oriented paradigm propagates the separation into objects. As a result of the so-called *tyranny of the dominant decomposition* [TOHS99], procedural and object-oriented decomposition techniques cannot capture all concerns in a modular way. These so-called *cross-cutting* concerns (also known as *aspects*) have two main properties, known to the aspect-oriented

community as *code scattering* (code that implements one concern is scattered across several classes) and *code tangling* (code implementing several concerns is tangled within the same class or method). For almost a decade, the research field of aspect-oriented programming (AOP) [KLM+97] has been dealing with capturing cross-cutting concerns in a modular way.

In this paper, we present an approach tailored to support SPLE, which allows a modular specification of concerns and their modular composition, in order to construct variants of an SPL. The approach uses concepts from generic and aspect-oriented programming and adapts them to the requirements imposed by SPLE.

When configuring an SPL, an end-user is interested in the concrete result, and not in how this is achieved. Therefore, the specification of a configuration should be *declarative*. Invalid configuration specifications should be *detected*. The resulting code should be *efficient*, i.e. it should be comparable with a manual implementation. Particulary, the resulting code should not contain any unnecessary indirections, introduced while configuring the SPL. A developer of an SPL is interested in *avoiding redundant code* as much as possible, especially considering maintenance issues.

The rest of this paper is organized as follows. Section 2 presents approaches which can be used to realize SPLs on the implementation level, by providing techniques to encapsulate concerns and/or compose variants of a SPL. Section 3 summarizes properties of concerns specific to SPLE. This serves as a rationale for the concern model used in our approach, which we present in section 4. How concerns represented using this model can be composed is shown in section 5. An example of an SPL implemented using our approach is presented in section 6, while section 7 concludes the paper.

2 Related Work

Probably the most prominent techniques used to implement SPLs are the ones based on text replacement and macro expansion. Tools such as the well known C-preprocessor CPP, but also template engines like Velocity, are often used in model driven development. SPLE-specific XML languages (e.g. the Variation Point Specification Language [Bec04]) also belong to this class of implementation techniques. Since these character-based techniques do not impose any constraints on the adapted software artefact, almost all of them can be processed. However, this also leads to one main drawback of character-based techniques - almost no assertions concerning syntactic or sematic correctness of the adaption result can be made. Furthermore, systematic specification and composition of concern implementations (especially for cross-cutting concerns) is not possible.

Using object oriented programming, concern implementations can be composed using delegation and inheritance. However, as pointed out in section 1, not all concerns can be captured in a modular way using object orientation. In order to compose a software system with alternative concern implementations, design patterns like *template method* or *bridge* must be used. This can lead to complex interfaces and unnecessarily complex system structures in the resulting software system. For example, this can be a serious problem

in embedded systems, where only limited processing and memory resources are available. Another problem is the poor support of the configuration and composition process, since the configuration of an SPL has to be written by the application programmer using non-declarative program code.

AOP extends traditional decomposition techniques with the possibility to capture and compose cross-cutting concerns in a modular way. From the viewpoint of SPLE, this is a big step towards composing variants of a SPL from modular concern implementations. Well-known tools that support AOP are AspectJ [Asp07] and the discontinued HyperJ [alp07]. However, these tools, as well as the concern model they implement, do not support all requirements imposed by SPLs. One problem is for example, that only *orthogonal* aspects are supported (i.e. aspects of aspects are not possible). In particular, the implementation of an aspect cannot be parameterized and adapted for different variants of an SPL, which is necessary, because aspect implementations can slightly differ for the different variants. Another requirement not addressed by today's AOP tools is the support for declarative configuration specifications and the detection of invalid configurations.

Meta-programming can be used to compose program fragments that belong to different concern implementations. Thus, meta-programming systems like the Meta-Environment [CWI07], RECODER [Rec07], COMPOST [Ass03] or INJECT/J [Inj07] can be used as base technologies. Especially in the context of SPLs implemented in C++, template meta-programming [CE00] is used to select and compose program fragments. A problem that is common to these base technologies, is that usually only syntactic correctness of the result can be guaranteed. Ensuring properties concerning the application semantics is usually out of scope of these base technologies. Furthermore, the specification of concrete configurations of an SPL is rarely declarative.

Model driven approaches can also be seen as SPLE approaches, although focussing in most cases on the feature *infrastructure*. Generators and templates used in model driven development are from this point of view a part of the domain implementation of an SPL (i.e. they specify the common concerns of the SPL, while the models specify the variable concerns). Generators can be realized using the techniques described above. Experience shows that generators based on template engines are the most common ones, having the identified drawbacks.

3 Concerns in Software Product Line Engineering

In contrast to engineering a single software system, SPLE differentiates between *domain engineering* and *application engineering* [CE00]. Domain engineering encompasses domain analysis, domain design and domain implementation. During domain analysis, the target domain is investigated in detail. This includes the so-called domain scoping, which means establishing which features and concepts belong to the domain addressed by an SPL. Furthermore, requirements as well as mandatory and alternative features of the domain are identified. In the domain design phase, a common software system architecture, shared by the variants of the SPL, is developed. Reusable assets (i.e., concern imple-

mentations) together with corresponding production plans are implemented in the domain implementation phase. While the goal of domain engineering is to create reusable assets from which variants of the SPL can be composed, application engineering deals with the derivation of a concrete SPL variant that suits the customer's needs. Application engineering encompasses requirements analysis, product configuration and integration. It reuses the results from domain engineering. In the requirements analysis phase, the results of the domain analysis are used to verify if customer needs can be fulfilled by the SPL. Based on the results of domain design, during product configuration, the features which have to be included in the resulting system are selected and dependencies between them are resolved. In a final step, the customer-specific system is built using the software assets implemented during domain implementation.

In the application domain, features are concepts found in the problem space (i.e., end-user visible requirements), while concerns are the corresponding concepts in the solution space (i.e. implementation-specific application requirements deduced from the user requirements). On the implementation level, a concern can be understood as a set of program fragments implementing the corresponding requirement.

Figure 2 shows how different variants of an SPL can be produced by composing different combinations of concern implementations. From an abstract point of view, two concerns can be composed using a composition operator. In contrast to opportunistic reuse (i.e. using COTS components which can be viewed as an implementation of one or more concerns), SPLE employs *predicted reuse*. With opportunistic reuse, a composition operator cannot be developed by a developer of a COTS component, since the context a component should be deployed in is usually not known. For an SPL, the situation is different, since the different implementations of a concern and their permitted combinations are known from domain engineering. Hence, composition operators are not arbitrary, but known for different variants of an SPL. This means that the different composition operators can already be supplied by the developers of the domain implementation. Another observation is that if only slightly different implementations of the same concerns should be composed, the composition operators also differ only slightly.

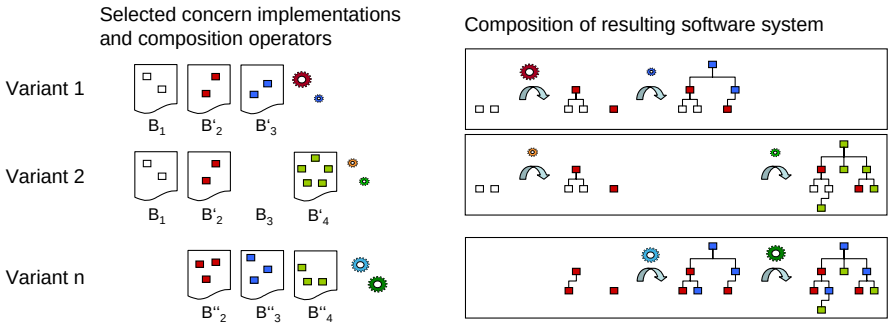

Figure 2: Similar concern implementations and composition operators in different variants derived from a SPL.

Based on this simple model, a domain implementation encompasses concern and composition operator implementations, possibly specific to particular variants of the SPL. Figure 3 schematically shows four concerns B_1 to B_4 with variant-specific implementations of concerns B_2, B_3 and B_4, as well as predefined composition operators.

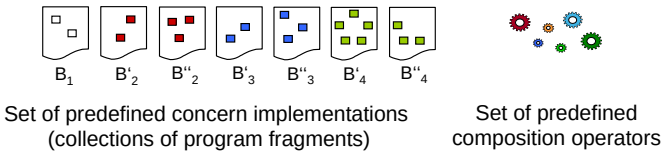

Figure 3: Domain implementation

However, a flat structure of the domain implementation does not support an application developer in an appropriate way. First, it should be possible that all program fragments implementing a concern are grouped together. This helps the application developer in selecting the needed program fragments, especially for cross-cutting concerns. Otherwise, it would require deep knowledge of the domain implementation to select the appropriate program fragments, which should be avoided. A second problem is the selection of the right variant-specific implementation of a concern, based on the configuration specified by the user. For example, if in figure 3 the user wants an SPL variant supporting concerns B_1, B_3 and B_4, which are the variant-specific concern implementations B_3', B_3'', B_4' or B_4'', that have to be selected? Even if this selection has been made, it is still unclear which matching composition operator has to be selected, in order to compose the program fragments specified by the concerns. Again, the selection of a composition operator requires deep knowledge of the domain implementation.

Our approach to solve these selection problems is to encapsulate composition operators and program fragments implementing a variant-specific concern implementation to a *logical unit*. For an application developer, this avoids the need for a deep knowledge of the domain implementation, since the complexity of program fragment selection and composition can be hidden inside this logical unit. Furthermore, if the composition technique used by the composition operator is capable of determining arbitrary insertion points for program fragments, even cross-cutting concerns can be represented in a modular way.

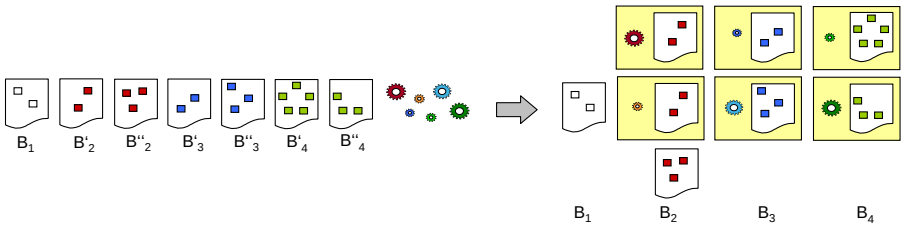

Figure 4: Encapsulation of concern implementations and composition operators.

A flat structure of the domain implementation not only hinders the application developer, but also the developer of the domain implementation. Two reasons are the complexity and redundancy within the implementation. Concern implementations as well as composition operators can become quite large. It would be almost impossible to create and maintain a concern implementation without the possibility for structuring the result. From a development and maintenance perspective, redundancy is also an important issue. Looking at the different variant-specific concern implementations, we can observe that they share a lot of equal or similar program fragments, a fact which is exploited by the well known generic programming. Equality and similarity of certain parts is not only true for program fragments belonging to a concern implementation, but also for the associated composition operators.

An established principle to deal with complexity is to divide the problem into smaller pieces, and to compose the solution from the solutions of the smaller problems. This can also be applied to the (probably cross-cutting) implementation of a concern, because as we have shown in [TK05], a concern may be coarser-grained or finer-grained. For example, the concern ”persistency” can be anything from the entire ”persistency” functionality for a complex model via the ”persistency” of a single class to the ”persistency” of a single field.

Based on this observation, our concern model supports nesting of concerns (figure 5). Concern nesting has several advantages. First, the complexity of the entire implementation of a concern can be divided into manageable parts. This applies to the set of program fragments implementing a (sub-)concern as well as to the composition operators associated with these program fragments. Concern nesting also supports information hiding. A composition operator for a given nesting level specifies how the program fragments of the current nesting level are composed to the resulting program. However, if this cannot be specified on the current nesting level, the composition operator of the surrounding nesting level can be used to solve this problem, since it has knowledge of more possibly interacting sub-concerns. Another advantage of concern nesting is the possibility to support concerns of concerns. In contrast to orthogonal concerns, which assume that a concern only interacts with one other concern, nesting of concerns allows the modelling of arbitrary concern interactions by specifying concerns of concerns, thus supporting non-orthogonal concerns. Furthermore, concern nesting also enables a seamless integration of the concept *concern parametrization*. Concern parametrization is important for SPLE, in order to adapt a concern implementation to different SPL variants. Our approach permits the parametrization of a concern by treating parameters as variable sub-concerns.

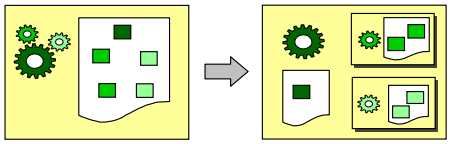

Figure 5: Concern nesting

In order to reduce redundancy found in different variants of an SPL, we propose to factor out the commonalities with respect to program fragments implementing the different

concern variants, as well as the associated concern composition operators. Again, concern nesting can help with the specification: possibly variable sub-concerns specify the differences, while the selection of the needed sub-concerns for a given variant is done by the composition operator. Differences in composition operators can be specified using variant-specific case differentiations. Figure 6 shows schematically how different variants of a concern can be aggregated into a single concern implementation. This also helps the application developer, since he does not need to distinguish the different implementations. Additionally, our approach provides the possibility to generate concrete program fragments based on context information, computed by a static analysis of the actual composition state and explicitly stated meta-data.

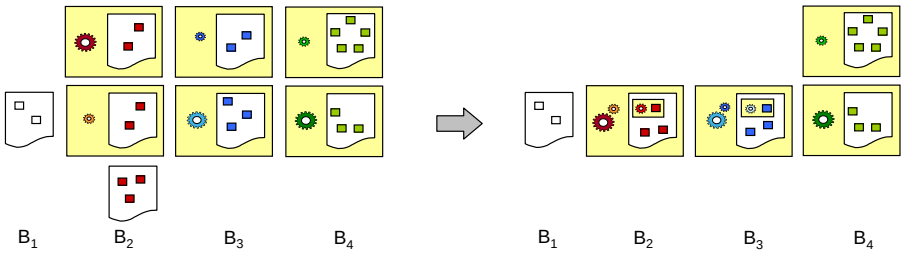

Figure 6: Concern aggregation

While concern nesting and concern aggregation helps in managing complexity and reducing redundancy, the problem of identifying invalid SPL configurations cannot be solved using these techniques. Since an application developer using an SPL does not necessarily have deep knowledge of the domain implementation, he is (among other things) interested in the answer to two questions: Is it actually possible to compose the selected concern implementations, and do the selected concerns implement the needed state transitions during runtime of the resulting program. To support the application developer, parts of these semantic properties should be checked automatically. To express such semantic properties in a machine processable way, we propose to give each concern a type, as well as to annotate a concern implementation with conditions specifying a contract.

In our approach, each concern is associated with a type. The type is on the meta-level of the resulting program. It marks all program fragments within the concern implementation as implementing certain state transitions, with respect to the conditions given by the application domain. Since a human associates a certain semantic behavior with a type name, it helps the application developer in selecting appropriate concerns. Applying the well-known type relation “inheritance” to these types allows the establishment of a type hierarchy, which can be used in conjunction with concern parameter types and type bounds, to constrain the set of valid sub-concerns. In addition to typed concerns, contracts specified using first-order predicate logic can be used to specify further semantic properties. These contracts can either be checked at composition time, or at execution time of the resulting program. Especially the checking of contracts at execution time is better known as Design-By-Contract [Mey00]. Using concern types and concern contracts, static identifi-

cation of invalid SPL configurations is reduced to type checking and checking of statically evaluable contracts.

4 Concern Model

The last section discussed the properties of concerns specific to SPL, as well as solutions to problems that arise from these properties. In this section, we introduce our concrete concern model, which is based on the proposed solutions.

Figure 7 shows the schematic concern model. It consists of a *composition interface*, a *composition operator* and one or more nested *sub-concerns*. A sub-concern can either be a concern, or as a terminal element a possibly generic program fragment. A generic program fragment is a program fragment which can be syntactically incomplete (i.e. other program fragments have to be inserted at so-called weave points[1]), in order to result in a syntactically complete program fragment.

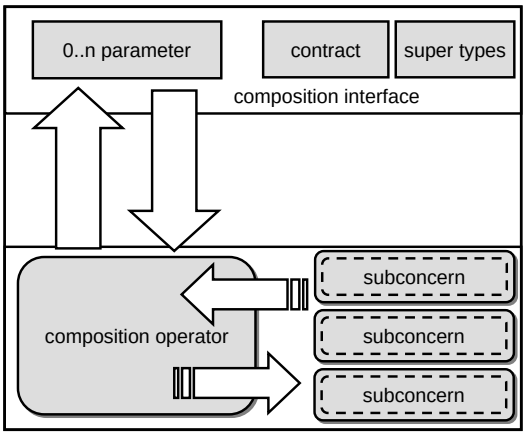

Figure 7: Schematic concern model

The composition interface consists of a set of parameters that allow the parametrization of the concern, a contract which allows the specification of additional composition constraints, a name for the concern, and a set of super types. From an application developer's point of view, the composition interface offers all information needed to specify a concrete SPL configuration.

As stated in section 3, each concern defines a new type on the meta-level of the resulting program. The new type is defined using the concern name and an optional set of super types. The notion of a type is similar to interface types in object-oriented languages. Ter-

[1]Especially character-based SPL implementation techniques often call these points *variation points*. They can also be compared to *join points* known from AspectJ.

minal concerns must have a super type defined by the meta-model (syntax) of the used programming language. For example, the super type `ClassFragment` indicates that a terminal concern will result in a class, while the super type `StatementFragment` denotes an arbitrary statement. By associating a concern with a type that is defined by the meta-model of the programming language, the concern type inherits the syntactic and static semantic properties of the language element. Using the concern name to which a human associates certain semantics, and the inheritance relationship, a concern type specifies the expected execution semantics of the concern implementation. For example, more execution semantics is associated with a concern named `TransferMethodFragment` and super type `MethodFragment` than with the general base type `MethodFragment`. Together with inherited properties such as inherited sub-concerns, composition operators and contracts, this allows domain and application developers to reason about the semantic properties of the concern. In particular, this also holds if a concrete implementation (i.e., a subtype) of a concern is implemented during application development. In this case, reasoning over concern properties is already possible during domain development, based on super type properties.

Concern parameters specify with which other concern types a given concern can or has to be composed. Parameters define variable sub-concerns. Furthermore, parameters specify which sub-concerns are available for the parametrization of the variable concerns. Thus, parameters describe a required and a provided interface. Each formal parameter has a type describing type bounds (i.e. a concrete concern must have a subtype of the stated type).

The last element of the composition interface is an additional contract, as discussed in section 3. The contract can be used to express further composition constraints, not expressible through the concern type system, such as dependencies between concern parameters. Constraints can be specified using first-order predicate logic. The concrete predicates are defined by a meta-model based on object-oriented meta-models. These predicates can range from simple information such as a class name, to complex information obtained from a static analysis of the current program composition state (e.g. all references of a given class within the current program).

The composition interface is central during application development, since it provides all the information that are needed by the application developer to derive a concrete variant of the SPL. The interface elements specify how to correctly parameterize a concern with other concerns, such that the program fragments provided by the concerns can be composed to the required program. How to compose the program fragments is specified by the composition operator. With respect to the resulting program, the composition operator is a meta-program. It is possible that a composition operator delegates composition operations to the composition operator of a sub-concern. Since such a sub-concern can encapsulate all its program fragments as well as all needed composition operations, this allows the modular specification of cross-cutting concerns. Delegation also allows deferring some of the design decisions associated with composition. For example, different subtypes of a cross-cutting concern may implement the expected behavior in different ways, thus needing different composition strategies. These different composition strategies can be encapsulated in the composition operator of the concrete sub-concerns, while the composition operator of the surrounding concern can ignore the different strategies.

The program fragments that belong to a concern can either be specified as source code, possibly containing explicit weave points, or they can be generated from templates, using context information provided by the aforementioned predicates and explicitly stated meta-information. The latter can be specified using program fragment annotations.

5 Concern Composition

To build a program during application engineering, the concern implementations have to be entangled properly. Because all concern implementations can be reduced to sets of program fragments, entangling them can be implemented through invasive program fragment composition [Gen04, Ass03] at implicit and explicit weave points. Program fragment composition is carried out in a bottom-up manner, based on a hierarchical configuration specification. This hierarchical configuration specification refers to the nesting of concerns. A configuration specification must at least ensure that all required concern parameters are bound. Figure 8 shows a simple example of three concerns `A`, `B` and `C`, which have to be composed using invasive program fragment composition. A configuration specification for this example is `A<C<..., B<>>>`. Concern `A` is parameterized using concern `C`. Concern `C` has to be parameterized with two (terminal) concerns, the first being a type reference and the second a code block with a certain type. In this example, the type reference is provided by concern `A`, thus it does not have to be specified in the configuration specification. Only a placeholder (`...`) is specified. The second required parameter for concern `C` is bound to concern `B`. The latter has two optional parameters, which are not used in this example. Note that the initially non-terminal concern `B` is a terminal one from the viewpoint of concern `C`, since it results in a program fragment of type ”code block”.

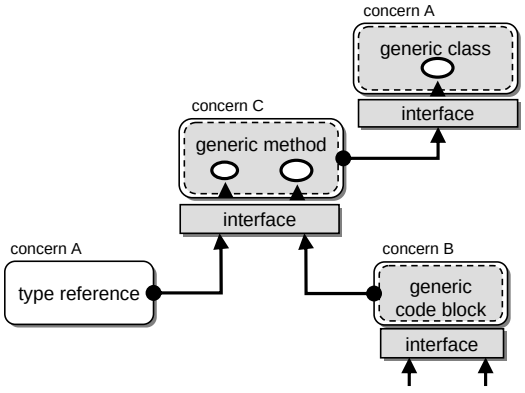

Figure 8: Bottom-up program fragment composition.

Bottom-up program fragment composition takes place along the hierarchy defined by the syntax of the programming language. For practical purposes, this means that program

fragments implementing a concern can be composed at different program fragment hierarchy levels. In the example shown in figure 8, program fragments specified by concern `A` are composed at expression level (type reference), as well as on class level.

6 Example of Use

Based on the model described in section 4 and the composition technique sketched in section 5, a prototype called CoCoSy (Concern Composition System) implementing the approach has been developed. The prototype has been built on top of the source code adaptation tool Inject/J [Inj07, Gen04]. As shown in figure 9, a CoCoSy editor which supports the developer in writing concern specifications and configurations is integrated within the Eclipse platform.

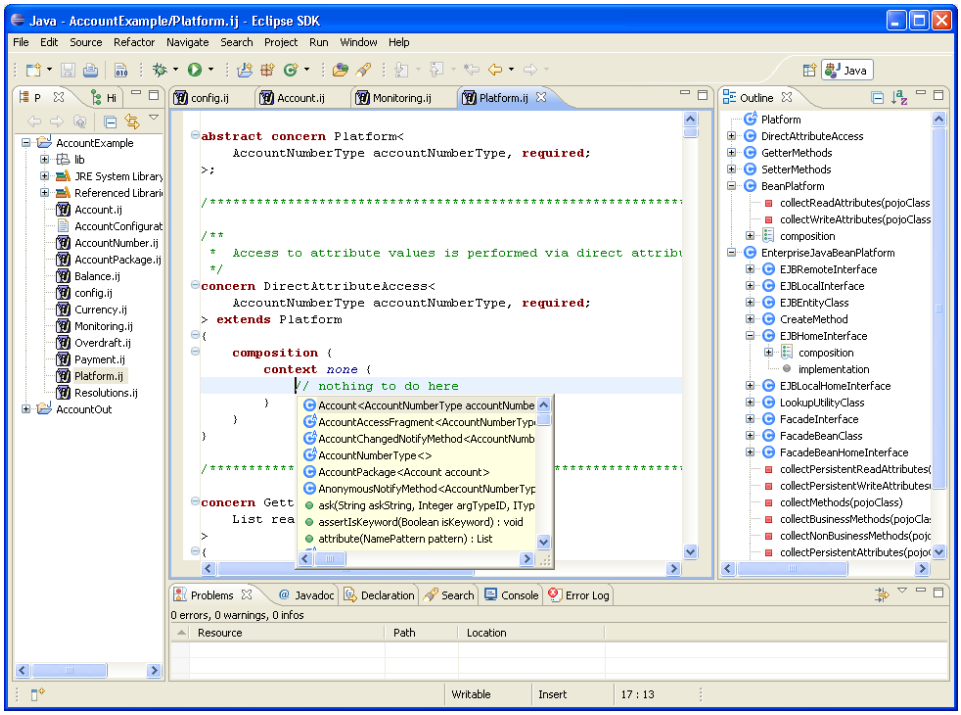

Figure 9: CoCoSy plug-in integrated in Eclipse.

Among other examples, this prototype has been used to implement an account SPL as described in [CE00, chapter 13], along with several extensions. The domain implementation consists of 33 top-level concern specifications, which can be subdivided into 8 concern interfaces defining concern super types, 17 simple concerns encapsulating only one (generic) program fragment, and 8 complex, cross-cutting concerns that contain several nested con-

cerns. An example for such a complex concern is `EnterpriseJavaBean`, a subtype of concern interface `Platform`. It encapsulates all program fragments and generation templates, as well as the needed composition operations which are necessary to transform a simple account class into an EntityBean, and an associated SessionBean. An example for a simple concern specification is shown in listing 1.

```
concern Account<
  AccountNumberType accountNumberType, required;
  BalanceType balanceType, required;
  Currency currency, required;
  ReferenceAccount<accountNumberType> referenceAccount, optional;
  Transfer<accountNumberType, balanceType, overdraftCheck,
           referenceAccount, ?> transferMethod, optional;
  CashWithdrawal<balanceType, overdraftCheck> withdrawMethod, optional;
  CashDeposit<balanceType> depositMethod, optional;
  Platform<accountNumberType> platformOperator, required;
  OverdraftCheck<balanceType> overdraftCheck, optional;
  Monitoring<accountNumberType, balanceType, ?> monitoring, optional;
> extends ClassFragment

contract {
  (implies( (transferMethod instanceof TransferToReferenceAccount),
            (referenceAccount!=null)));
   implies( (depositMethod!=null),
            (transferMethod!=null || withdrawMethod!=null) )
}
{ implementation ${
    class @(name) {
      // @annotation persistent="true" read="true" write="true"
      private @(balanceType) balance;
      ...
      @(transferMethod)
      ...
    }
  }$
}
```

Listing 1: Concern specification ”Account”

For the account SPL, all concerns could be captured in a modular way, even if they are cross-cutting with respect to the resulting Java program. As intended by our concern model, redundant implementations for each variant could be avoided. Differences in the implementation of a concern were covered by factoring out these differences into sub-concerns. Variant specific case differentiations in the selection and composition of sub-concerns could be expressed using the composition operator. For some concerns, variant specific generation of program fragments based on the result of a static analysis of the current composition state and on the usage of annotations helped to further reduce the redundant specification of similar program fragments.

A declarative specification of a concrete account configuration is possible. Listing 2 shows an example for such a declarative specification. An application developer only has to specify which concerns should be supported in the resulting account. He does not need to touch the various program fragments and complex composition operations necessary to build the account variant. Additionally, the concern type system helps the developer in choosing appropriate concerns. The type system also helps to detect semantically invalid configurations. The few invalid account variants which could not be detected using the

concern type system can be tracked down using additional composition contracts.

```
configuration Account<
  StringAccountNumber,
  DoubleBalance,
  CurrencyEuropeanUnion,
  null,
  TransferToArbitraryAccount<...,...,...,..., EJBAccountAccess>,
  CashWithdrawal<...,...>,
  CashDeposit<...>,
  EnterpriseJavaBeanPlatform<...>,
  CreditOperator<...>,
  Monitoring<...,..., NonAnonymousNotifyMethod>
>;
```

Listing 2: Configuration specification for a monitored account for an EJB platform

By using invasive program fragment composition to entangle concerns, the resulting source code does not contain unwanted indirections originating from the composition technique. For example, unwanted indirections could be delegations, design patterns etc., which are only necessary for composition. The resulting account source code has interfaces and implementations tailored and customized to the user's requirements.

For the account example, the criteria introduced in section 1 could be satisfied using our concern model and composition technique. Concerns could be specified in a modular way and redundancies could be avoided. The configuration of the account SPL can be specified declaratively and invalid configurations are detected. The resulting code is efficient in the sense that the resulting source code can be compared to a manual implementation. However, further case studies are necessary in order to prove that these promising results can be transferred to larger, real-world SPLs. Although the approach presented in this paper can help improve the implementation of an SPL, it does not solve the more general problem of SPLE, namely the high initial costs for a domain implementation.

7 Conclusions

In this paper, we investigated the nature of implementations of concerns specific to SPLE. Based on these observations, we developed approaches which solve SPLE-specific issues regarding the modular, redundance-poor specification and composition of concerns. These considerations resulted in a concern model tailored to support the implementation of an SPL. The model and the composition technique uses concepts from generic and aspect-oriented programming, and adapts them to the requirements imposed by SPLE.

The main contribution of this model is that it allows nesting of concerns. Based on this concept, arbitrary parameterizations of concerns (aspects) are possible. Furthermore, concern nesting allows the explicit modelling of concern interactions. Concern nesting also provides a universal method to deal with composition conflicts that arise from concern interaction due to non-orthogonal concerns.

A concern type system and explicit concern composition contracts allow us to reason about

semantically meaningful configurations of an SPL. Using concern types and concern contracts, static identification of invalid SPL configurations is reduced to type checking and checking of statically evaluable contracts.

References

[alp07] IBM alphaWorks. HyperJ web site. http://www.alphaworks.ibm.com/tech/hyperj, 2007.

[Asp07] AspectJ Team. AspectJ web site. http://www.eclipse.org/aspectj/, 2007.

[Ass03] Uwe Assmann. *Invasive Software Composition*. Springer Verlag, 2003.

[Bec04] Martin Becker. *Anpassungsunterstützung in Software-Produktfamilien*. PhD thesis, Technische Universität Kaiserslautern, 2004.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, May 2000.

[CWI07] CWI. Meta-Environment web site. http://www.cwi.nl/projects/MetaEnv/, 2007.

[Dij82] Edsger W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*, chapter On the role of scientific thought, pages 60–66. Springer, 1982.

[Gen04] Thomas Genßler. *Werkzeuggestützte Adaption objektorientierter Programme*. PhD thesis, Universität Karlsruhe (TH), 2004.

[Inj07] Inject/J Team. Inject/J web site. http://injectj.fzi.de/, 2007.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241. Springer-Verlag, 1997.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 2000.

[Rec07] Recoder Team. RECODER web site. http://recoder.sourceforge.net/, 2007.

[TK05] Mircea Trifu and Volker Kuttruff. Capturing Nontrivial Concerns in Object-Oriented Software. In *Proceedings of the 12-th Working Conference on Reverse Engineering*. IEEE, Nov 2005.

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*, 1999.