Facilitating Configurability by Separation of Concerns in the Source Code

Zoltan Fazekas

Faculty of Informatics and Information Technologies, Slovak University of Technology, Bratislava, Slovakia

Producing configurations of a software product, e.g. designed for different operating systems, using different database technologies or serving different groups of users is undoubtedly a time-consuming and error-prone process. In this paper we propose an approach facilitating the configurability of software using separation of concerns, which helps to eliminate unwanted parts of the source code whenever the corresponding requirements change without manual intervention and without the risk of corrupting the program. A prototype implementation provided with the approach demonstrates its powerfulness in the practice.

Keywords: concerns, separation of concerns, configuration construction, software configuration management.

1. Introduction

Everybody who has ever dealt with programming has probably faced the following situation. You implement the code strictly according to the specification, however, the requirements change in the meantime, e.g. the customer does not want some features any more. In addition to the changed requirements, some features of the program are not complete yet, i.e. they also have to be excluded from the configuration to be delivered. You try to remove the corresponding parts of the program, i.e. components as well as client code using them, however you recognize in the middle, that the configuration does not work properly because parts of the program that should remain in place depend on the parts you removed. In the end, after removing all unnecessary parts of the program, you get a working configuration that corresponds to the current requirements. However, you have invested plenty of time and energy to accomplish this. In this paper we propose an approach to reconfiguring programs, i.e. eliminating unwanted parts from them, whenever the corresponding requirements change, without manual coding effort and without the risk of corrupting the program.

A software product can exist in several configurations, each having a slightly different source code. Each of these configurations can support different operating systems, run on different application servers, use different database or communication technologies, provide different user interfaces, support different (human) languages, have different regional specificities or different capabilities for different kinds of users and so on and so forth. Configurations are assembled from versions of configuration items, i.e. components, of the software product. Each of the configuration items contributes to some characteristics of the software product, e.g. implements some capabilities and runs on some operating system. The goal of assembling configurations is to select all configuration items that contribute to the required characteristics, but have nothing to do with the other ones. Unfortunately, characteristics that a particular version of a configuration item contributes to are not always transparent. This makes assembling of configurations a non-trivial task. A heuristic approach to selecting the right components and their versions to be included in the configuration have been presented in [11]. Another popular model had been introduced earlier in the work on Adele [15]. Furthermore, versions of configuration items usually correspond

to more than one characteristic, each of which can be required or just undesired in a configuration. On the other hand, several characteristics are scattered along multiple configuration items. Consequently, it is not always possible to combine arbitrary characteristics in a configuration, since there might be configuration items that contribute to required, as well as undesired ones.

To come around these shortcomings, we explored if existing approaches to separation of concerns could be used to separate different characteristics of versions of configuration items from each other on the level of source code. The principle of separation of concerns is essential for any scientific discipline [1] since it helps to manage complexity and handle changes, which facilitates parallel work. The common basic concept of these approaches, the concern, refers to "those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders." [3] Indeed, existing approaches to separation of concerns facilitate the maintainability of software. Some of them, like AspectJ [4] or Composition Filters [9], let the programmer modularize otherwise dispersed crosscutting concerns in order to ease their maintenance and compose them later into the execution flow of programs. Some of them, like Hyper/J [5], even enable the coexistence of multiple decompositions of a program along different kinds of concerns. A useful application of these approaches in change management — representation of changes as aspects — is discussed in [2]. However, since the use of these approaches requires a fundamental shift in the way of designing and implementing systems and introduces a further level of complexity (see e.g. the notion of scattering in [24]), they are not always practicable. Another group of approaches provides dynamic view capabilities similar to those in Desert [17] and Sheets [16] to visualize concerns in source code (referred to as visual separation of concerns in [24]) instead of isolating concerns from each other. Its representatives, including Stellation [24] (formerly Coven [6]), Software Plans [14], CME [21], FEAT [19], SNIAFL [20], JQuery [22], the Aspect Browser [18] and the Aspect Mining Tool [13] enable the programmer to demarcate manually discovered concerns in the source code, or localize concerns, using static and dynamic code analysis techniques and not (least) visualize them, e.g. by highlighting, or view them as virtual modules to ease their comprehension and maintenance. A good overview of further research results facilitating the location of concerns in the source code using different techniques is presented in [13], [20] and [22]. The main difference between these approaches and ours is that while they focus on facilitating discovery of concerns in the code (i.e. defining of their scope) with a precision not enabling their removal from the program in a non-invasive way, our approach emphasizes precise (i.e. token-level) concern location enabling seamless concern removal for the price of more manual effort invested into concern location. Despite numerous advantages, all approaches and implementations we investigated showed to be insufficient for constructing configurations based on concerns. The majority of them was not able to represent concerns in the source code with a precision enabling their non-invasive removal without substantial manual effort necessary for fixing the resulting code. In addition, all of these approaches considered units of the program, such as member functions and methods, as the smallest unit of concerns. In particular cases, however, concerns manifest themselves only in a few important tokens such as the name of a class to be instantiated in order to initialize the corresponding database driver before its first use. Therefore, we formulated the hypothesis, that it would be necessary for the seamless construction of concern-based configurations to track concerns in the source code on the granularity level of single tokens. So unlike other approaches, ours established the association of concerns with elements of the program on various levels of granularity, from larger modules through smaller units down to single tokens. Under some circumstances the association of concerns to tokens could be automated using custom search rules and patterns. Additionally, tokens could be associated to concerns, representing units of the program they were part of, programmers who worked on them as well as timestamps of their creation and removal without manual effort. The rest of the concerns needed to be associated with portions of the source code manually. As next, we defined configuration items as a collection of tokens contributing to the same set of concerns (except the creation and removal time of tokens).

Versions (revisions) of configuration items valid at a particular point in time could be computed dynamically, based on the concerns associated with them, necessarily including their creation and removal time instead of being stored explicitly. We hoped to be able to assemble arbitrary configurations, i.e. the ones having arbitrary combinations of characteristics. This way, however, we soon recognized, that their syntactical correctness could only be assured if the source code had been written in a particular form and had been associated with concerns in a way fulfilling certain requirements. We also found out, that for particular combinations of concerns it did not make sense to be associated with the same token or to be present in the same configuration. Such cases required that semantic consistency of configurations be expressed in form of relationships between concerns. To use the strengths of other approaches to separation of concerns too, we designed our approach in such a way that it could be used on top of any existing programming paradigm including e.g. aspect-oriented programming [12]. In order to prove that our theory worked, we created a prototype implementation of it integrated into the Eclipse development environment. To mention at least one of the shortcomings too, our approach, just like many other approaches, still lacks a rational solution to the problem of logically handling a large amount of concerns arising already in relatively simple systems.

The rest of this paper is structured as follows. The second section introduces our approach in more details and provides some practical examples. The third section consists of a brief overview of the prototype implementation. Finally, in section four we draw some conclusions.

2. Concern-Based Configurations

"Reconfiguring the software modules comprising a system to add or to delete a feature typically requires substantial effort. This lack of flexibility increases the costs of building variants of a system, amongst other problems." [8] To come around this, we propose a new approach to configuration management, using lightweight separation of concerns. Why lightweight? Our approach relies on conventional programming paradigms and extends them by keeping track of concerns in the source code as it evolves [10] and generating its arbitrary configurations based on these concerns. Since portions of the source code contributing to different concerns are not isolated explicitly, we do not need to bother about their composition. Despite all insufficiencies of its implementations in Hyper/J, CME and Stellation (Coven) with respect to concern-based configuration construction, the main principle of multidimensional separation of concerns proved to be helpful to our problem. Its adaptation enabled us to organize changes in the program, i.e. insertions and removals of tokens, into versions along any concerns even different from the conventional ones such as the computing unit and the time of change. Here, changes were performed on a particular computing unit in a single step, i.e. between checking out the unit from and checking it in back into the source code management system, and got bundled in a version. Contrary to other known approaches to separation of concerns, we assign source code to concerns on the level of single tokens. Thereby, modifications performed within the extent of a token such as the deletion of characters do not affect already existing assignments of the token to concerns.

There is a variety of concerns appropriate for characterizing software configurations. Some of them are generic, i.e. suitable for any kind of software, others are specific to a particular domain or technology. Some typical kinds of concerns that proved to be suitable are:

— structural program elements such as packages, classes, modules, functions, units etc., depending on the programming language being used

— business-relevant information such as customer, address, account, stock, vehicle, gas pressure etc., depending on the problem domain

— business processes, services or functions such as creation of the balance of an account, computation of the optimal gas pressure, selection of customers for a marketing campaign, simulation of chemical reactions etc., depending on the problem domain

— technical data such as transaction context, session data, control bit, timestamp, delete flag, checksum, token, digital signature etc.

— technical computations such as opening a session, committing a transaction, reading a database record, cleaning up a buffer, transforming text to a number, sorting records, authorizing access etc.

— any kinds of requirements such as use cases and features, e.g. user login, password authentication, 7×24 availability etc.

— decisions concerning programming paradigms, languages and techniques, design patterns, architectural metaphors, coding conventions, communication protocols, computation methods and algorithms and many more

— compatibility aspects such as operating system, software and hardware platforms, technologies and products being integrated

— owners and authors of the code, such as a person, group, project team or an organization

— customers, user groups and user profiles, source code licensing policies, etc.

— software delivery items, such as products, product suites, their versions, major and minor releases, updates, patches, etc.

— validity and expiration of source code, i.e. creation and removal time, time of promotion in terms of the development process, e.g. from the coding stage to the testing stage, etc.

These are all concerns of especial interest for configuration construction. A more general purpose model of concern classification can be found in Cosmos [23].

We illustrate the usage of our approach on the example of the Customer Registration System (CRS), a simple Java application used for registering the contact addresses of customers ordering products through a hotline.

Let us assume that CRS is used by several reseller companies, so it is delivered in different configurations. For instance, one configuration has been designed for companies operating internationally, but only outside the US (see Fig. 2.), so customer data incorporate the *country*, but not the *state*. Companies having customers in the US too required another configuration considering the *state* as part of the customer address (see Fig. 1.).

Fig. 1. CRS configuration for US companies operating internationally.

Fig. 2. CRS configuration for non-US companies operating internationally.

Some other companies operating in one country outside the US only preferred a configuration of CRS considering neither the *country* nor the *state* as part of the customer address. To model their needs we first defined the concerns used for representing different requirements of these customers. We defined a concern covering the requirements of companies operating in US and called it simply *"_US-specific"* (the underscore is a naming convention used to distinguish custom concerns from automatically recognized ones more easily), and another one compromising the requirements resulting from international operation called *"_international"*. By combining only these two concerns it was possible to generate four meaningful configurations: one for companies operating in the US only, one for companies operating world-wide, one for companies having customers in a single non-US country only and, last but not least, companies with customers in multiple countries except the US. However, as we mentioned

above, concerns like the kind of customers being considered are only one of the aspects that can contribute to the variety of configuration. Some others that we do not address in this example for the sake of simplicity could be e.g. different operating systems or underlying database technologies.

The architecture of CRS follows the MVC design pattern. The *model.CustomerModel* class implements the model which represents a customer contacting the call center. Each customer has, besides the attributes *name*, *address*, etc., a unique identifier (*id*), which gets generated when the customer object is created. The list of already reserved ids is stored in the static variable *reservedIds*. The customers' data are persisted in files using the object serialization mechanism of Java. In order to accelerate searching for them, customers are indexed according to their attributes. The static variables *names*, *streets*, etc. contain index tables mapping the values of the corresponding attributes to ids of customers. We will need these details about the *model.CustomerModel* class later, for reasoning about the implementation of the above introduced concerns. Program elements such as the *address* and the *state* attributes involved in the implementation of those concerns occur also in many other classes of the CRS application, e.g. the *view.AppletView* class which implements the user interface displaying a customer in a Java applet and the *controller.AbstractController*, *controller.RegistrationController* and *controller.ViewController* classes coordinating the interaction between the view and the model. These issues will not be further discussed in this paper.

2.1. Meaningfulness of Configurations

If we claim to allow for non-invasive elimination of concerns from the source code, we must reason about the consequences such operations could have on the semantics of the program. Since concerns are usually not completely independent, i.e. some of them contradict or are prerequisites of each other, it is not appropriate to eliminate any of them. Semantic dependencies (also called relationships or predicates) between concerns have to be taken into account during the assignment of tokens to concerns, as well as during the selection of concerns to be included in a configuration. Otherwise the resulting assignment or configuration could become inconsistent or produce undesired behaviour at run-time. For instance, if there is a concern assigned to portions of the source code implementing trace messages about events occurring during the execution of the program, and another concern assigned to portions of the source code dealing with the optimisation of the performance of the application, they should neither be assigned to the same token nor be included in the same configuration, since tracking and performance are contradicting requirements. Another example could be a concern assigned to portions of the source code dealing with persistence, and another concern assigned to portions of the source code producing and handling exceptions. The persistence concern cannot exist without exception handling mechanisms, therefore each configuration that contains the persistence concern should also include the exception handling concern. Originally, we introduced concern relationships to prevent that one of the concerns requiring each other get excluded from a configuration and the others not, or that contradicting concerns get assigned to the same token or get included in the same configuration. However, concern relationships can also be used to automate the assignment of concerns to tokens. For instance, a concern relationship can make sure that each token belonging to a program unit (e.g. a method like *getName()*) gets automatically (i.e. without manual intervention) assigned to the concerns related to this program unit (e.g. the *name* concern). In the rest of this section, we introduce some common types of concern relationships:

— Assumption is an explicit relationship. It is set by the programmer between two concerns, if the inclusion of the first concern in a configuration of the program assumes (requires) the inclusion of the second concern in the configuration.

— Implication is an explicit relationship. It is set by the programmer between two concerns, if the assignment of any token to the first concern implies the assignment of the same token to the second concern.

— Partition is an explicit relationship. It is set by the programmer between at least two con-

cerns, if any token can be assigned to at most one of these concerns.

Some of the concern relationships, such as the assumption and implication, are transitive and can be derived automatically, based on already existing ones.

In this simple model we have only included concern relationships that proved to be particularly useful for our purposes. More general purpose models of concern relationships are discussed in [21] and [23].

2.2. Assignment of Tokens to Concerns

Manual assignment is the most common way of assigning portions of the source code to concerns. Code segments assigned to a particular concern are sometimes whole units like the method shown in Fig. 3., whole statements like the declarations in Fig. 4. or just a few tokens like in Fig. 5.

```
public String getState ( ) {
    return state ;
}
```

Fig. 3. A whole method assigned to the *"_US-specific"* concern.

In the example depicted in Fig. 4., the highlighted tokens are assigned to the concern called *"_US-specific"*, because both the *state* attribute of the customer, as well as the *states* index table, are required only in the variant desired for the US-market.

The assignment of tokens to concerns is best performed on the fly as tokens are being created; otherwise the quality of assignments might get negatively affected.

Manual identification and assignment of concerns to tokens consumes costly time of the programmer. To come around this, we use automatic recognition of concerns in the source

```
package model ;

import java . util . * ;
import java . io . * ;

public class CustomerModel implements Serializable {

    private static Vector reservedIds = new Vector ( ) ;
    private static Hashtable names = new Hashtable ( ) ;
    private static Hashtable streets = new Hashtable ( ) ;
    private static Hashtable cities = new Hashtable ( ) ;
    private static Hashtable states = new Hashtable ( ) ;
    private static Hashtable countries = new Hashtable ( ) ;
    private String id = null ;
    private String name = null ;
    private String street = null ;
    private String city = null ;
    private String state = null ;
    private String country = null ;
```

Fig. 4. The highlighted variable declarations are assigned to the *"_US-specific"* concern.

```
public static Vector find ( String name , String street , String city ,
                            String state , String country ) {
```

Fig. 5. The highlighted tokens are assigned to the *"_US-specific"* concern.

code. Unfortunately, this is only partially implemented in the current version of our prototype.

The pattern-matching approach is similar to the mechanism used by other approaches to separation of concerns for specifying join points. It matches code segments according to regular expressions specified by the programmer. Parts of the regular expressions associated with some concerns cause the corresponding parts of the matched code segments to be assigned to those concerns. For instance, all references to a particular attribute like *state* can be found easily, using a trivial regular expression consisting of its exact name. Using wildcards, all other program elements, associated with the *state* attribute like the *states* variable, can be found. The validity of concerns recognized this way is always temporary: they have to be re-evaluated each time the source code goes through changes.

The relationship-based approach uses existing concern relationships for deriving new assignments of tokens to concerns. For instance, if there is an abstract *"_customer management"* concern grouping everything that has to do with customers and a more concrete *"_US-specific"* concern related by an implication relationship, whenever the programmer assigns a token to the *"_US-specific"* concern, it gets automatically assigned to the *"_customer management"* concern as well. Concerns derived this way are only valid as long as the manual assignment of concerns does not change. Then, however, they have to be re-evaluated.

Some concerns can get recognized and assigned to tokens automatically, without the use of rules. Such concerns are usually derived from facts and events provided by the development environment. For example, they can represent program unit(s) the token belongs to, the programmer the token was created, modified or removed by, as well as the token's creation and removal time.

2.3. Construction of Configurations

We define configuration items as virtual collections of tokens associated with the same set of concerns. Configuration items defined this way are actually completely independent of the physical storage of the program. The whole source code can just be stored in the traditional way, in files representing units or even in one piece containing all tokens that have ever been created together with information about their association with concerns. The programmer gets to see only a subset of the source code consisting of tokens not yet removed. Although we defined the creation and removal time of tokens as concerns too, they are not used to characterize configuration items. Instead, they are used for the construction of versions (revisions) of configuration items. The version of a configuration item (which consists of tokens valid at the given point time), is computed dynamically by filtering out tokens from the configuration item that were created after the given point in time or were removed before it. Consequently, each token is by default part of exactly one configuration item, but can possibly belong to its multiple versions. Using this approach, we overcome redundant storage of source code in versions.

According to the definition above, for each set of concerns there is exactly one configuration item containing all tokens that contribute to all of the concerns contained in that set, but no others. Given a set of desired concerns, the corresponding configuration comprises all configuration items characterized by some subset of this set. This configuration, however, does not know the versions of the configuration items it contains. Just after it has been complemented by a point in time, configuration items contained in the configuration can be replaced by their version valid at the given point in time. If no other point in time is specified, the configuration corresponding to the current point in time is constructed.

It is important to assign those tokens to a concern, that can be removed later from the source code in a non-invasive way. Assigning more or assigning less tokens than necessary would probably corrupt the syntax of the program. As you will see later, it is not easy to meet this requirement. One important element helping to achieve it is the possibility to specify custom rules determining which configuration items tokens are to be included in. By default tokens get included only in the configuration item representing the combination of concerns assigned to them. They can, however, be included in any other configuration item that represents a subset of these concerns instead, or in addition to,

the default one. For each single token the programmer can define arbitrary number of subsets of concerns already assigned to it, which determine the configuration items the token gets included in.

There are some special cases that have to be treated very carefully by the programmer. A typical example is presented in Fig. 6. The first of the last two formal parameters, *state*, is assigned to the *"_US-specific"* concern, the second one, *country*, to the *"_international"* concern. To assure that the comma between them gets removed properly, it has to be assigned to both concerns and it should be included both in the configuration item characterized by both concerns as well as in the configuration item characterized by the *"_international"* concern only. Let us assume, that there would be an *"_intercity"* concern assigned to *city*, the third formal parameter from the end. Then the comma between the *city* and the *state* formal parameters would have to be assigned both to the *"_intercity"* concern as well as to the *"_US-specific"* concern, but it would only have to be included in the configuration item characterized by both concerns.

This problem can be generalized in the following way. Let us consider a list of n elements $E_1, E_2, \ldots, E_n$ separated by $n-1$ commas, one between each two adjacent elements E_i and E_{i+1}. Let us define the function $f : B^n \rightarrow B^{n-1}$ which computes for each n-tuple $P = (P_1, P_2, \ldots, P_n)$ where $P_i \in B$ and $P_i = 1$ if and only if E_i is present in a given sublist of the original list of elements, an $(n-1)$-tuple $(f_1(P), f_2(P), \ldots, f_{n-1}(P))$ where $f_i(P) \in B$ and $f_i(P) = 1$ if and only if the i-th separator (i.e. the comma between E_i and E_{i+1}) should be present in the concerned sublist. We call the function $f(P)$ the presence function of the list of n elements and its i-th component $f_i(P)$ the presence function of the i-th separator. A possible truth table of the function f for a four-element list is shown in Fig. 7. It is important to notice, that for the 4-tuples $(0, 1, 0, 1)$, $(1, 0, 0, 1)$, $(1, 0, 1, 0)$, $(1, 0, 1, 1)$ and $(1, 1, 0, 1)$ we have more alternatives to define the value of the function f. For instance, for $(1, 0, 0, 1)$ we could choose between $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$. We decided to follow the principle that each two elements that were not adjacent in the original list should be separated in the sublist by the comma directly preced-

P				f(P)		
P_1	P_2	P_3	P_4	$f_1(P)$	$f_2(P)$	$f_3(P)$
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	1
0	1	0	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	0	1	0
0	1	1	1	0	1	1
1	0	0	0	0	0	0
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	1	0	0
1	1	0	1	1	0	1
1	1	1	0	1	1	0
1	1	1	1	1	1	1

Fig. 7. The truth table of the presence function f for four parameters.

```
public CustomerModel ( String name , String street , String city ,
                       String state , String country ) {
    setId ( ) ;
    setName ( name ) ;
    setStreet ( street ) ;
    setCity ( city ) ;
    setState ( state ) ;
    setCountry ( country ) ;
}
```

Fig. 6. Tokens assigned to the *"_US-specific"* (light-colored background) and the *"_international"* (dark-colored background) concerns and to both of them (last comma in the parameter list).

ing the second one, so we chose $(0, 0, 1)$. We use Carnaugh maps shown in Fig. 8., Fig. 9. and Fig. 10. to find the appropriate normal disjunctive forms of the components $f_1(P)$, $f_2(P)$ and $f_3(P)$. It is easy to discover that $f_1(P) = P_1 \wedge P_2$, $f_2(P) = (P_1 \wedge P_3) \vee (P_2 \wedge P_3)$ and $f_3(P) = (P_1 \wedge P_4) \vee (P_2 \wedge P_4) \vee (P_3 \wedge P_4)$. Generalized for n elements this means, that $f_i(P) = (P_1 \wedge P_{i+1}) \vee (P_2 \wedge P_{i+1}) \vee \ldots \vee (P_i \wedge P_{i+1})$ for each $1 \leq i \leq n-1$.

P_3 \ P_1 \ P_4 \ P_2				
	1	1	0	0
	1	1	0	0
	0	0	0	0
	0	0	0	0

Fig. 8. The Carnaugh map of the presence function f of the first separator in a list of four elements.

P_3 \ P_1 \ P_4 \ P_2				
	1	1	1	1
	0	0	0	0
	0	0	0	0
	1	1	0	0

Fig. 9. The Carnaugh map of the presence function f of the second separator in a list of four elements.

P_3 \ P_1 \ P_4 \ P_2				
	1	0	0	1
	1	0	0	1
	1	0	0	0
	1	0	0	1

Fig. 10. The Carnaugh map of the presence function f of the third separator in a list of four elements.

It is often necessary to revise the assignment of tokens to concerns if their adjacency goes through changes. Fig. 6. shows a good example for this situation too. If a further formal parameter were appended to the end of the parameter list, the concerns assigned to the comma between the *state* and the *country* formal parameters would have to be revised.

After this modification to the code, the comma would only have to be included in the configuration item characterized by both the *"_US-specific"* concern and the *"_international"* concern, but not in the configuration item characterized by the *"_international"* concern only any more.

Our experiments have shown that it is less time-consuming and error-prone to choose which concerns should be missing from the configuration than to specify all concerns that should be included in it. Therefore, we propose a configuration specification strategy that by default includes all concerns into the configuration and lets the programmer select which ones should be omitted.

To stay with the example from Fig. 6, the configuration of the source code not including the *"_US-specific"* concern, but including all other concerns would look like as shown in Fig. 11.

The same source code segment of another configuration that contains neither the *"_US-specific"* nor the *"_international"* concern would look like as shown in Fig. 12.

```
public CustomerModel ( String name , String street , String city
                     , String country ) {
    setId ( ) ;
    setName ( name ) ;
    setStreet ( street ) ;
    setCity ( city ) ;
    setCountry ( country ) ;
}
```

Fig. 11. The sample code segment without the *"_US-specific"* concern.

```
public CustomerModel ( String name , String street , String city
                       ) {
    setId ( ) ;
    setName ( name ) ;
    setStreet ( street ) ;
    setCity ( city ) ;
}
```

Fig. 12. The sample code segment without the *"_US-specific"* and *"_international"* concerns.

2.4. Assuring Removability of Concerns

In order to apply the above described practices properly, sometimes it is necessary to take some constraints on writing the source code and assigning it to concerns into account. In general, programs written in almost any high-level programming language like C, C++ or Java, consist of some kinds of modules (e.g. functions or objects and their member functions), which in turn consist of declarations and statements complemented by reserved symbols and words for the sake of syntactical correctness. Although this intuitive model of programs is quite abstract, it provides a helpful basis for the following consideration. Declarations and statements consist of expressions and reserved symbols and words like = representing value assignment or ; representing the separator between statements in the Java language. Expressions consist of one or more constants and identifiers referencing declared elements of the program (e.g. variables, functions) combined by reserved symbols and words (e.g. operators like +, -, *). Let us take a deeper look into what happens when portions of the program get omitted because of the exclusion of a concern from the configuration. In general, statements can be removed without the risk of violating the syntax of the program. On the other hand, reserved symbols and words normally cannot be removed safely, unless the whole statement containing them is removed. Removal of declarations is possible, but more difficult than the removal of statements, because it requires the removal of all references to the declared elements from the program too. The latter one can actually be classified as removal of an expression. Removal of an expression from the program usually results in syntax violation, because e.g. an argument of an operation will be missing. However, there is a workaround for this problem: a special way of constructing expressions. Expressions always belong to an abstract data type like boolean, integer or object. For each abstract data type, there is a constant value, which all variables of the given type contain right after their creation, before they are assigned a value. For integers this is 0, for real numbers 0.0 and for objects null. Let us call this constant value the default element of the abstract data type. There are also operations defined on each abstract data type, like + for integer and real numbers or & for booleans representing the logical AND operator. For the abstract data types and operators originating from mathematics like integers and the + operator, there is a neutral value, in this case 0. In some cases the programmer has even multiple choices of the operation and the neutral element, e.g. for integers it could be the + operation with 0 as the neutral element or the * operation (multiplication) and 1 as the neutral element. Expressions that are subject to removal, i.e. are assigned to some concern, have to be extended in the following way. If E is such an expression and E is of the abstract data type for which there is an operation, e.g. +, with a neutral element, e.g. 0, E should be replaced by $E + 0$. If E is assigned to some concerns, the + operator has to be assigned to those concerns as well. If E belongs to an abstract data type without an operation having a neutral element, the statement $S(E)$ containing the expression E has to be modified as follows. If d is the default element of the abstract data type of E, the statement $S(d)$ has to be inserted in front of the statement $S(E)$ in a syntactically correct way. Instead of assigning E within the statement $S(E)$ to some concerns, the whole statement $S(E)$ has to be assigned to those concerns. The programmer could, of course, use any other constant value corresponding to the given abstract data type instead of the default element d. As a result of excluding the concerns assigned to the expression E from a configuration, E will be replaced

by the value 0 or the value d, depending on the abstract data type of E. Of course, the rest of the program relying on the value of E has to be robust enough to be able to process 0 and d instead of the original value of the expression of E.

```
public String toString ( ) {
    return "" + getName ( ) ;
}
```

Fig. 13. Return expression gets extended by a neutral element.

An example illustrating the necessary adaption of expressions described above is shown in Fig. 13. Let us suppose the $+$, *getName*, (and) tokens are assigned to some concern. If this concern were not included in the configuration, the above named tokens would disappear causing the source code to be syntactically incorrect. On the other hand, it is not possible to assign the whole return statement to the concern, because that would result in syntactically incorrect source code if the concern were missing from the configuration. In order to come around this problem, the returned expression has to be extended by the neutral element, in this case " ", using the operation $+$.

3. Prototype Implementation

The described approach will only be accepted by programmers if it is supported by easy-to-use tools closely integrated with the programming environment the programmers are used to. In order to prove the practical applicability of our approach, we implemented a prototype extension of an existing software development environment (see Fig. 14.) that supports the most concepts introduced in the preceding sections.

The software development environment we extended is Eclipse 3, an open integrated development environment. Using the plug-in facility we added new views for highlighting tokens in the source code, assigning concerns to them, selecting concerns to be included into the configuration, generating configuration items corresponding to the specified selection of concerns and, last but not least, defining semantic

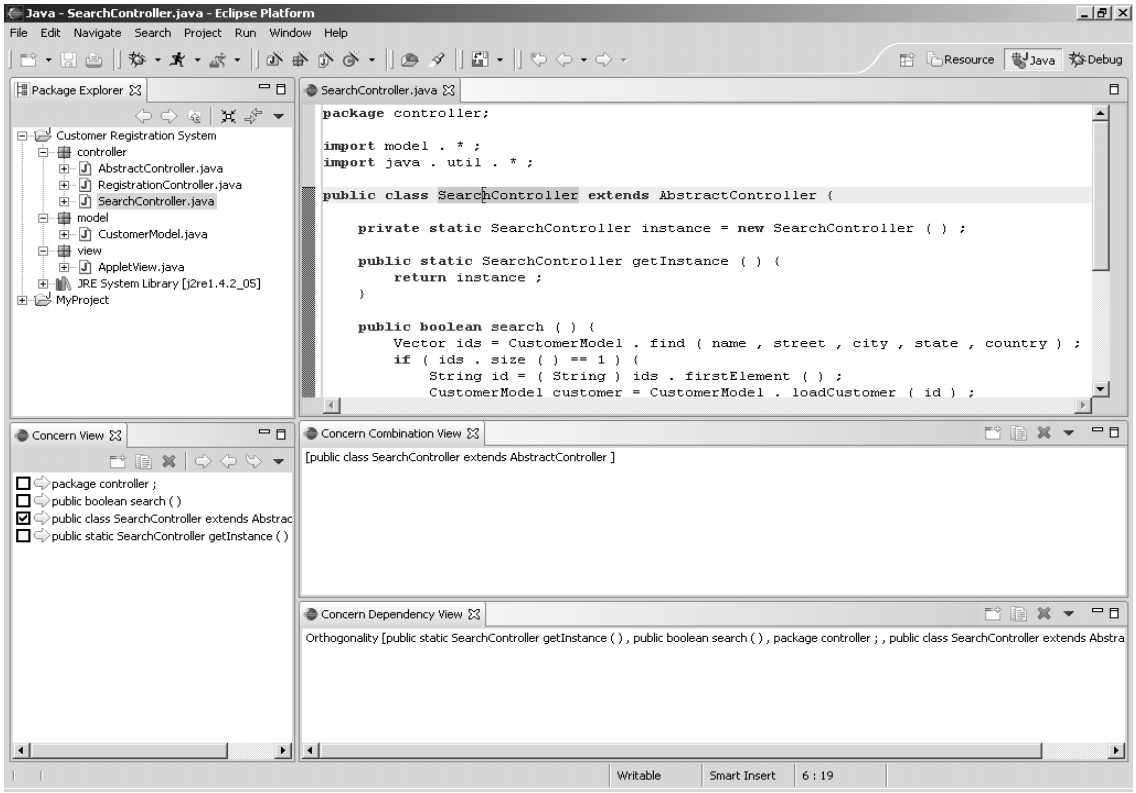

Fig. 14. The prototype implementation integrated into the Eclipse development environment.

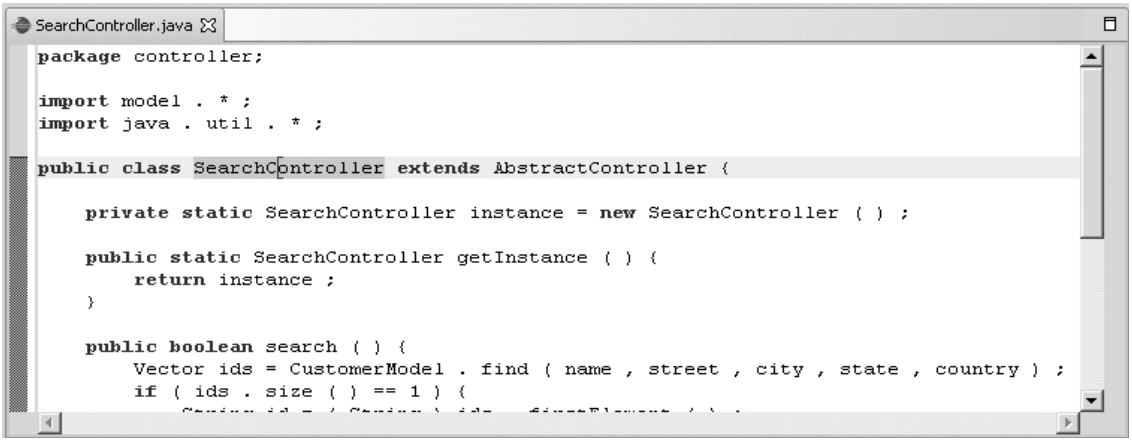

Fig. 15. The Java Source Editor highlighting the token pointed to by the cursor.

dependencies between concerns. In the rest of this chapter we introduce these and other useful features in more detail.

3.1. The Java Source Editor

The Java Source Editor (see Fig. 15.) is an editor for Java source code based on the editor provided by Eclipse.

It uses the JavaCC 3.2 parser compiler and a grammar, which currently supports version 1.1 of the Java language specification, for recognizing tokens in the source code. The positions of tokens are traced also when the source code becomes unparsable. After restoring the parsability of the source code, it gets reparsed and the old tokens get matched against the newly parsed ones according to their positions.

3.2. The Concern View

The Concern View serves, among the rest, for displaying concerns (see Fig. 16.). This view contains a list of custom concerns defined by the programmer, as well as of automatically recognized concerns representing units of the program, i.e. packages, classes and methods.

When defining new concerns, the programmer has to give the new concern a name, which has to be unique among all concerns. The Concern View is also used for assigning concerns to tokens. If a single token is highlighted in the Java Source Editor, the concerns assigned to it will be shown with checked state in the

Fig. 16. The Concern View displaying custom as well as automatically recognized concerns.

Concern View. The concern representing the unit of the finest possible granularity, the source code which includes the token, gets automatically assigned to each token. This assignment cannot be modified by the programmer. Other concerns can be assigned to the highlighted token by the programmer setting them in checked state. If one or more tokens are selected in the Java Source Editor, the tokens that are assigned to all of them are displayed with checked state in the Concern View. If the user sets some other concerns' state to be checked, they get assigned to all tokens that fall into the selected range.

If no dependency is selected in the Concern Dependency View (see 3.4. The Concern Dependency View) and no combination is selected in the Concern Combination View (see 3.3. The

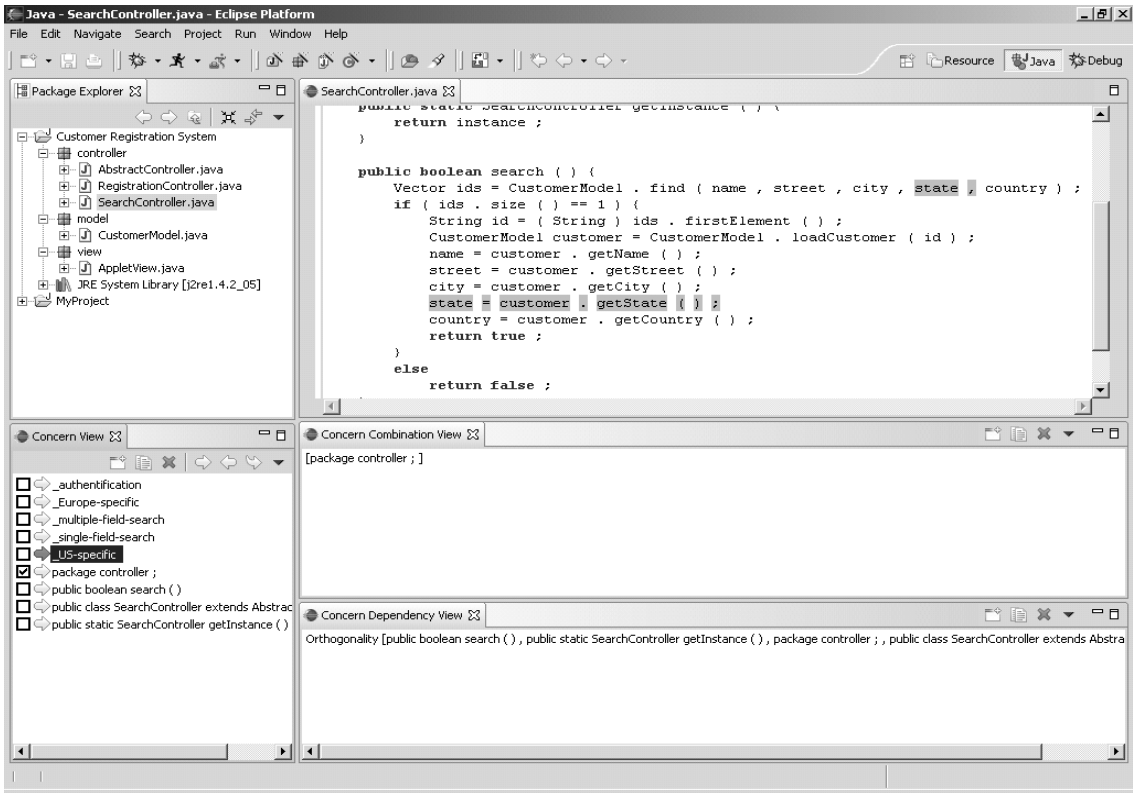

Fig. 17. Highlighting portions of the source code corresponding to concerns selected in the Concern View (on the left) in the Java Source Editor (on the right).

Concern Combinaton View), but at least one concern is selected in the Concern View, the tokens assigned to the selected concerns are highlighted in the Java Source Editor using the colour representing the corresponding concern (see Fig. 17.).

The Concern View also shows which concerns are currently included in the configuration of the program. If the arrow icon of a concern is directed to the right, the concern is included in the configuration, otherwise not. The programmer can turn any concern on or off by double-clicking on it. Then the source code in the Java Source Editor gets modified automatically to show only the tokens that are part of the corresponding configuration.

3.3. The Concern Combination View

Tokens become part of the configuration if there is any combination of concerns they are assigned to, that is fully included in the selected configuration. One combination of concerns gets defined for each token automatically. This contains all concerns the token is assigned to.

If there is no other combination defined by the programmer for this token, it will be included in the selected configuration only if all concerns it is assigned to are included in the configuration. However, the programmer can define arbitrary combinations of concerns — subsets of the concerns assigned to the token — additionally to the default combination. The programmer has to select some of the concerns in the Concern View, which are assigned to the currently highlighted token. Then he has to choose to create a new concern combination in the Concern Combination View (see Fig. 18.). The created combination will then contain all concerns the programmer has selected in the Concern View. If the programmer selects an existing combination in the Concern Combination View, the concerns it contains will be selected in the Concern View. The programmer can include or exclude concerns from this combination by modifying the selection in the Concern View. Furthermore, the programmer can choose to create a copy of the selected combination and modify it after-

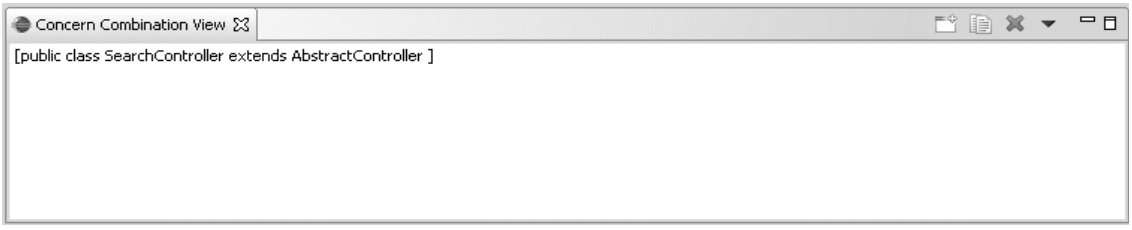

Fig. 18. The Concern Combination View shows for each token the combinations of concerns that determine which configuration items it will be included in.

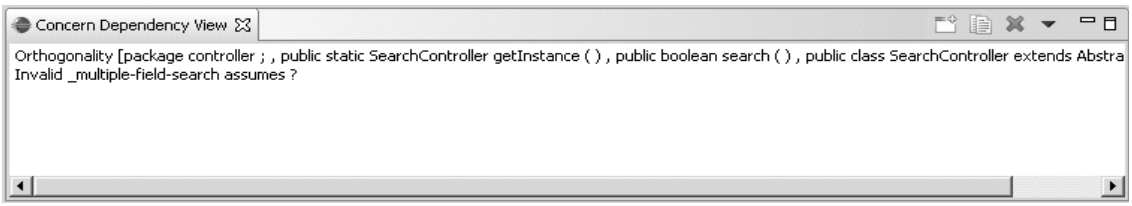

Fig. 19. The Concern Dependency View shows semantic relationships between concerns.

wards. The programmer can remove concern combinations, except the automatically defined one.

3.4. The Concern Dependency View

The Concern Dependency View (see Fig. 19.) displays concern dependencies, also called semantic concern relationships. Some of these dependencies such as the orthogonality between concerns representing units of the program can be recognized automatically. This is added to the Concern Dependency View automatically and updated each time the set of units of the program change.

The programmer can define further custom dependencies by selecting one or more concerns in the Concern View and choosing the dependency type from a drop-down list. If the number of selected concerns is not required by the type of the dependency, the Concern Dependency View displays the dependency as invalid. If the programmer selects a dependency in the Concern Dependency View, the concerns constituting the dependency get selected in the Concern View. The programmer can then add or remove concerns from the selected dependency by modifying the selection of concerns in the Concern View. The programmer can remove any dependencies from the Concern Dependency View, but the automatically generated ones.

4. Conclusion

We have proposed an approach to automated generation of software configurations based on separation of concerns. As our experiments have shown, existing approaches to separation of concerns are not suitable for generating such configurations, because they consider concerns in the source code on a too coarse level of granularity. Our approach associates each token of the source code to concerns used as the basis for describing software configurations. Furthermore, we have proposed to use semantic relationships between concerns to assure semantic correctness (i.e. meaningfulness) of the generated configurations. We have developed a method for writing source code in a way that enables us to re-configure it flexibly without the risk of corrupting its syntax. Our approach has, among others, the advantage, that it lets the programmer work with conventional code (e.g. object-oriented in Java) and does not introduce any new paradigm. With appropriate tool support like the one we developed for demonstration purposes, the overhead produced by activities necessary for designing and maintaining concerns is not significantly higher than the time needed to document the source code. In return, the time needed to (re)produce an arbitrary configuration of the source code that is compilable,

works correctly and does exactly what is specified in the means of concerns, takes only a few seconds.

Acknowledgement

The work reported here was partially supported by Slovak Scientific Agency, project No. VG 1/0162/03.

References

[1] E.W. DIJKSTRA, *A Discipline of Programming*, Prentice-Hall, 1976.

[2] P. DOLOG, V. VRANIC, M. BIELIKOVA, Representing Change by Aspect, *ACM SIGPLAN Notices*, 36(1), December 2001.

[3] IEEE Std 1471-2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE, 2000.

[4] J. IRWIN, G. KICZALES, J. LAMPING, J.-M. LOINGTIER, C.V. LOPES, C. MAEDA, A. MENDHEKAR, Aspect-Oriented Programming, Published in *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP), Springer Verlag, 1997.

[5] P. TARR, H. OSSHER, *Hyper/J™ User and Installation Manual*, IBM Corporation, 2000.

[6] M.C. CHU-CARROLL, Separation of Concerns in Software Configuration Management, *ICSE 2001 Workshop on Advanced Separation of Concerns*, March 2001.

[7] E.L.A. BANIASSAD, G.C. MURPHY, C. SCHWANNINGER, Determining the "Why" of Concerns, Published in *Proceedings for Advanced Separation of Concerns Workshop* at the *23rd International Conference on Software Engineering* in Toronto, Canada 2001.

[8] A. LAI, G.C. MURPHY, M.P. ROBILLARD, R.J. WALKER, Separating Features in Source Code: An Exploratory Study, *IEEE 0-7695-1050-7/01*, 2001.

[9] M. AKSIT, L. BERGMANS, Solving the Evolution Problems Using Composition Filters, *ECOOP 2001 tutorial, TRESE e-tutorial series 02*, TRESE Group, University of Twente 2001.

[10] Z. FAZEKAS, Concern-Based Software Evolution, Published in *Proceedings of SOFSEM 2005*, *31st Annual Conference on Current Trends in Theory and Practice of Informatics*, January 2005, Liptovsky Jan, Slovak Republic.

[11] M. BIELIKOVA, P. NAVRAT, Approach to improving software configuration management, In *Fifth European Conference on Software Quality*, pp. 374–383, Trinity College, Dublin, Ireland, 1996.

[12] K. CZARNECKI, U.W. EISENECKER, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, Boston (2000), Chapter 7: Aspect-Oriented Decomposition and Composition, pp. 189–250.

[13] J. HANNEMANN, G. KICZALES, Overcoming the Prevalent Decomposition in Legacy Code, *Workshop on Advanced Separation of Concerns (Proceedings)*, *International Conference on Software Engineering*, May 2001, Toronto, Canada.

[14] D. COPPIT, B. COX, Software Plans for Separation of Concerns, *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, March 22, 2004. Held in conjunction with the *Third International Conference on Aspect-Oriented Software Development* (AOSD 2004), Lancaster, UK.

[15] J. ESTUBLIER, R. CASALLAS, *The Adele Configuration Manager, Configuration Management*, Edited by Tichy, John Wiley & Son Ltd, 1994.

[16] R. STOCKTON, N. KRAMER, *The Sheets Hypercode Editor*, Tech. Rep. 0820, CMU Department of Computer Science, 1997.

[17] S.P. REISS, Simplifying Data Integration: The Design of the Desert Software Development Environment, In the *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, 1996.

[18] W.G. GRISWOLD, Y. KATO, J.J. YUAN, *Aspect Browser: Tool Support for Managing Dispersed Aspects*, Technical Report CS99-0640, Department of Computer Science and Engineering, University of California, San Diego, December 1999.

[19] M.P. ROBILLARD, G.C. MURPHY, *Evolving Descriptions of Scattered Concerns*, Technical Report SOCS-TR-2005.1, School of Computer Science, McGill University, Canada, January 2005.

[20] W. ZHAO, L. ZHANG, Y. LIU, J. SUN, F. YANG, SNIAFL: Towards a Static Non-Interactive Approach to Feature Location, *Proceedings of the 26th International Conference on Software Engineering*, Volume 00, pp. 293–303, 2004.

[21] W. HARRISON, H. OSSHER, S.M. SUTTON, P. TARR, *Concern Modeling in the Concern Manipulation Environment*, IBM Research Report RC23344, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 2004.

[22] D. JANZEN, K. DE VOLDER, Navigating and Querying Code Without Getting Lost, *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, Boston, Massachusetts, pp. 178–187, 2003.

[23] S.M. SUTTON, I. ROUVELLOU, Modeling of Software Concerns in Cosmos, *Proceedings of the 1st International Conference on Aspect-oriented Software Development*, Enschede, The Netherlands, pp. 127–133, 2002.

[24] M.C. CHU-CARROLL, J. WRIGHT, A.T.T. YING, Visual separation of concerns through multidimensional program storage, *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, Boston, Massachusetts, pp. 188–197, 2003.

Received: August, 2004
Revised: March, 2005
Accepted: May, 2005

Contact address:
Zoltan Fazekas
Faculty of Informatics and Information Technologies
Slovak University of Technology
Ilkovičova 3
842 16 Bratislava
Slovakia
e-mail: fazekas@fiit.stuba.sk

ZOLTAN FAZEKAS received his Mgr. (MSc.) in 2001 and his postgraduate degree RNDr. in 2002, both in information technology and both from Comenius University in Bratislava. Since 2002 he is a PhD. student at the Faculty of Informatics and Information Technologies at Slovak University of Technology in Bratislava. His research interests include separation of concerns and software configuration management.