

Advanced Separation of Concerns

ICSE 2001

Separation of Concerns at the Source

Abstract: *This document describes a different approach to the merging of concerns than currently available. In contrast to other approaches, here, the merging is performed on the source code level. This approach satisfies not only the requirements defined by the separation of concerns but also those defined by outside constraints.*

1. SoCatS: What are our goals in software development?

Just a short list, as we all know what I am talking about here...

- Easy extensibility
- High level of reusability
- Separation of Concerns
- Easy tracing and tracking of errors during test and the productive cycle

Contents

1. What are our goals in software development?.....	1
2. Additional Outside Constraints.....	1
3. How can we fulfil these requirements?.....	2
4. A really short overview over Hyper/J and AspectJ.....	3
5. Solution: A new tool.....	4
6. What do we plan for the future?....	9

2. SoCatS: Additional Outside Constraints

In addition to the items listed above, there are additional constraints imposed on the development process from the outside.

2.1. Different testing cycles

During development, white-box and black-box tests of all components occur. After they have been done, most testing is black-box.

Then, as the software is given over to quality assurance, a new cycle of white-box and black-box testing occurs, to ensure that the software meets the standards and requirements set by the project.

2.2. Quality Assurance

In addition to these tests, the quality assurance team has the responsibility to judge the complete software for its feasibility and capability to solve the tasks at hand.

In all projects I have done until now, QA always required the complete sourcecode of the actual running software, not »a set of aspects« which are merged in a way which can not be checked by QA.

While QA always did see the merits of the Separation of Concerns, they always had the requirement to get a single running copy of the program in source **and** compiled

code where the content of the source **exactly** reflects the content of the compiled code.

2.3. Designers and Developers

Designers and developers are generally uncomfortable with the idea of »not getting the program in one piece of code«.

3. SoCatS: How can we fulfil these requirements?

In Java, the above goals can be achieved through a number of ways, each with it's own merits:

3.1. Observers and Listeners

One approach uses observers and listeners to achieve an extremely low coupling between different components, allowing for a generic approach to solve a problem.

3.1.1. Problem:

With this approach, many different kinds of »events« can take place. Either these events are encapsulated in different event classes, resulting in a **lot** of classes and many different event handling methods, or the events are handled with only a few event classes and a great deal of if-then clauses to determine just which event has taken place.

Both approaches are in no way bad in themselves, but to develop an application that way is, imho, overkill.

Also, below a certain level, this kind of coupling becomes overkill. Just try imagining a class which notifies all it's listeners of the execution of every single method of it's code for logging purposes and you get the idea.

3.2. Inheritance

A different approach is the intelligent use of inheritance, stacking one concern on the other. This approach allows a tool to run in »normal« mode, or through subclasses, in »debugging« mode, or through yet another subclass in »logging« mode.

```
MainFunction
Logging extends MainFunction
Debugging extends MainFunction
```

3.2.1. Problems

3.2.1.1. Changes to the inheritance structure

If you now have to change the inheritance structure to incorporate a new concern like networking, you have a great deal of changes you have to adjust to.

```
MainFunction
Networking extends MainFunction

Logging extends what?
Debugging extends what?
```

3.2.1.2. Additional concerns

Introduce additional logging of the main functions

```
MainFunction
Logging extends MainFunction
Networking extends Logging
```

3.2.1.3. Special constructs

In addition to that, many constructs, like singletons, do not handle subclassing gracefully. Of course, that can also be compensated for, but this is just a pattern to circumvent what is basically a design flaw.

3.3. Hyper/J, AspectJ and other, similar tools

A third approach is presented by tools like Hyper/J or AspectJ, which recompile the programs in accordance to a macro language, depending on the concerns involved, creating a customised, specialised version of the program for each need, be it logging or debugging, additional functionality, different customers, etc.

4. SoCatS: A really short overview over Hyper/J and AspectJ

4.1. Hyper/J:

4.1.1. Pro:

Hyper/J uses external configuration files to merge standard java files. This allows us to better separate code from configuration, enhancing the aspect of reusability.

4.1.2. Contra:

Hyper/J doesn't allow for white box testing of any already interwoven module as the weaving is done on the bytecode level. This makes it difficult for a piece of software to pass the testing cycle performed by the developers before the code is given to QA for further testing.

4.2. AspectJ:

4.2.1. Pro:

AspectJ allows the generation of interwoven source code, which then can be used to perform white box testing and black box testing. This is a powerful feature.

4.2.2. Contra:

On the downside, though, the configuration, just how the aspects are to be combined, is done within the aspect itself, i.e. within the code.

This is good for the exact matching of aspects but severely lacks when different modules are to be used in different projects. Then, any interweaving has to run on the base of facade objects, which then call the actual functionality.

4.3. Summary

The above tools are well up to the task of effectively merging and weaving of concerns.

Unfortunately, the outside constraints, as defined in (2) can not easily be fulfilled by any of the tools.

Also, they lack a good user interface, being basically »developer tools«, with developer defined as »the ones who write the code«.

5. SoCatS: Solution: A new tool.

When all available tools don't give us the functions desired, a new tool must be developed.

Such a tool needs to emulate all capabilities of at least one of the tools described above and possibly feature new, additional functions.

I have decided to first tackle the functionality of Hyper/J.

5.1. Basic Functionality

These functions exist in Hyper/J and should be emulated in a new tool.

5.1.1. Basics

Classes and packages containing different concerns need to be merged to form a whole. The result then is stored in new packages which contain all specified classes from all relevant concerns, and within these classes, all specified methods from these concerns.

If I speak of methods or classes »with the same name« here, this either indicates that they really have the same name and signature or that they have been explicitly matched.

5.1.2. Example

Let's take a look at an example to clarify the different functions provided.

This example features only one class in three concerns and does only clarify the merging of methods. The merging of whole classes or single statements does work along the same lines but is too long to be handled within this document.

Example
<code>package org.someorganisation.logging</code>
SomeClass
<pre>public void log(Object anObject, int aLogLevel, String aMessage) [Logging code] public int doSums(int intA, int intB) [Logging for doSums] public int doDivisions(int intA, int intB) [Logging for doDivisions] public int doMultiplications(int intA, int intB) [Logging for doMultiplicationa]</pre>
<code>package org.someorganisation.debugging</code>
SomeClass

```

public void debug(Object anObject, int aLogLevel, String aMessage)
    [Debugging functions]
public int doSums(int intA, int intB)
    [Debugging for doSums]
public int doDivisions(int intA, int intB)
    [Debugging for doDivisions]
public int doMultiplications(int intA, int intB)
    [Debugging for doMultiplicationa]
public int preDoDivisions(int intA, int intB)
    [Pre bracket for for doDivisions]
public int postDoDivisions(int intA, int intB)
    [Post bracket for doDivisions]

```

```
package org.someorganisation.accounting
```

SomeClass

```

public int doSums(int intA, int intB)
    [add intA and intB]
public int doDivisions(int intA, int intB)
    [divide intA through intB]
public int doMultiplications(int intA, int intB)
    [multiply two ints]

```

5.1.3. Merging

This function merges packages and classes.

It takes all specified classes from the specified concerns and copies them to the target packages. If a class exists in two concerns, all methods which occur in only one of the concerns are copied as well. Methods appearing in more than one concern are ignored, as they need to be handled by one of the other functions.

Let's merge the concerns Accounting and Logging without explicit matching:

Example: Merging

```
package org.someorganisation.composite
```

SomeClass

```

public int doSums(int intA, int intB)
    [add intA and intB]
public int doDivisions(int intA, int intB)
    [divide intA through intB]
public int doMultiplications(int intA, int intB)
    [multiply two ints]
public void log(Object anObject, int aLogLevel, String aMessage)
    [Logging code]

```

5.1.4. Appending

Appending handles all methods / classes which appear in more than one concern. All methods of the same name will be appended in the sequence the concerns are matched. Methods of different names and signatures are ignored as they are handled by the »Merging« functionality.

Automatic merging (as an extension to »Appending«) can be enabled.

Let us merge the concerns Accounting and Debugging here:

Example: Appending
package org.someorganisation.composite
SomeClass
<pre> public int doSums(int intA, int intB) [add intA and intB] [Debugging for doSums] public int doDivisions(int intA, int intB) [divide intA through intB] [Debugging for doDivisions] public int doMultiplications(int intA, int intB) [multiply two ints] [Debugging for doMultiplications] </pre>

As you can see, only those methods occurring in the Accounting concern are matched, all others are ignored.

5.1.5. Overwriting

Overwriting allows us to supplant a class or method in one concern with another class / method of another concern.

The classes / methods need to be the same name and signature, or be explicitly matched, which allows the name to be different.

Taking the above example, let's overwrite `doSums` in the accounting concern with `doSums` in the debugging concern.

Example: Overwriting
package org.someorganisation.composite
SomeClass
<pre> public int doSums(int intA, int intB) [Debugging for doSums] public int doDivisions(int intA, int intB) [divide intA through intB] public int doMultiplications(int intA, int intB) [multiply two ints] </pre>

This is actually quite a silly example as we're eliminating the functionality of the method `doSums`, but think of an application where you define the customisations for different customers in different concerns and thus can supplant the default behaviour with the customised versions.

5.1.6. Bracketing

Bracketing allows us to bracket the execution of a method in one concern with two methods in a different concern.

This function is very powerful and complicated.

Methods of the same signature can be automatically matched as seen here. We'll match Accounting and Debugging again:

Example: Bracketing 1
package org.someorganisation.composite
SomeClass
<pre> public int doDivisions(int intA, int intB) [preDoDivision code] [divide intA through intB] [postDoDivision code] </pre>

In addition to the above, automatic matching, the bracketing classes / methods can be explicitly mapped as well. As long as the signature is identical, there is no problem.

5.2. Enhanced functionality

Here, I'll describe what kind of additional functions I have introduced.

5.2.1. Finer granularity

Instead of matching only packages, classes and methods, I wanted, in conjunction with (5.2.2) to be able to match concerns down to single line of code.

This is done through the use of an XML structure for the java code and the use of XPath as query language.

5.2.2. Regular Expression Support

XPath statements are used to match concerns. This allows for a great deal of flexibility in the matching. The matching thus follows the guidelines for XPath, which allow most of the currently available regular expressions.

5.2.3. Enhanced standard matching functionality

To ease the use of the above functions, some assumptions are made.

- Classes, which have the same public and protected signature (i.e. their methods have the same names and signatures) can be matched without explicit mapping.
- Methods of the same signature but in classes of different names can be matched by specifying the classes which are to be mapped.
- Any other mapping needs to be explicitly specified.

5.2.4. Enhanced Bracketing

The new enhanced version of bracketing allows us to bracket methods with other methods of a different signature.

If they do have the same signature, a wrapper method is generated which calls the three methods in sequence. The parameters are passed on and return values can be processed.

Example: Enhanced Bracketing
<code>package org.someorganisation.debugging</code>
SomeClass
<code>public void firstBracket(Object anObject) [code for the start of the bracket] public void secondBracket(int intA, Object anObject) [code for the start of the bracket]</code>
<code>package org.someorganisation.composite</code>
SomeClass

```

public int doDivisionsBracketed(int intA, int intB, Object anObject)
{
    firstBracket(anObject);
    int result = doDivisions(intA, intB);
    secondBracket(intA, anObject);
    return result;
}
private int doDivisions(int intA, int intB)
private void firstBracket(Object anObject)
private void secondBracket(int intA, Object anObject)

```

All calls to the doSomething method will automatically be changed to doSomethingBracketed.

This will result in many errors in the resulting code and is generally heavily discouraged, but can be quite usable in the creation of facade objects.

5.2.5. Additional functionality

There are several additional steps which will ease the use of the tool and further its acceptance within a team of developers.

➤ **Generation of a powerful and flexible user interface to manage concerns**

Using a management interface allows the abstraction from the code level.

Thus, no longer a matching of class A and class B in the packages »org.someorg.a« and »org.someorg.b« takes place, but the matching of the concern »Accounting« and »Booking«.

After the initial matching (package / class to concern) has been performed, the more abstract approach can be used to manage the concerns.

This allows better handling and the concerns can be more easily communicated to outsiders which do not know the codebase.

➤ **Creation of robust storage structures**

The storage structures for this tool need to be as robust as possible and need to allow for multiple users and versioning of the initial data, configurations and resulting data (Do I hear someone say CVS here?).

➤ **Human-comprehensible merging process.**

After all definitions are done, the concerns need to be merged to form a whole.

This merging can **and** should happen on the lowest possible level (down to at least the method level, possibly even lower, down to single statements).

This process needs to be human comprehensible at any point to allow for smooth development and productive use. This is achieved through the use of XML as case for the merging process.

5.2.6. Additional Enhancement

There are additional enhancements which just ease the development process and are not part of this document.

6. SoCatS: What do we plan for the future?

6.1. Getting all features working

The next couple of months, I will spend trying to get all features completely working, adding inner class support, for instance, and enhancing the usability of the user interface.

After that, a stable, full featured beta will become available.

6.2. Production Environment Testing

Beginning with stable beta phase, the software will be employed in at least one of my commercial projects to gather information about the usability of the software in a production environment.

The findings gathered from this project will be incorporated in the gamma version.

During the project, I will, at regular intervals, report on the project–website. These findings may prove insightful to other users of the tool.

6.3. Product enhancements based on production environment requirements

The functionality of the tool will be enhanced as we go along the development process. Currently, the functions basically mirror those of Hyper/J, but I already have plans to incorporate much of what AspectJ can do.

These enhancements will take place in Gamma phase.