In: *Workshop on Advanced Separation of Concerns (Proceedings), International Conference on Software Engineering* (May 2001, Toronto, Canada)

Overcoming the Prevalent Decomposition in Legacy Code

Jan Hannemann and Gregor Kiczales
University of British Columbia
{jan|gregor}@cs.ubc.ca

The potential benefits of advanced separation of concerns (ASOC) techniques are well known and many programmers find the idea of using them appealing. For new software engineering projects these modularization mechanisms offer guidelines of how to structure the system modules. But how can legacy systems profit from them? Code related to concerns not represented in the current modularization has to be carefully identified and extracted while preserving system integrity.

This paper presents a refactoring tool that aids in the extraction of concerns that are ill-represented in the prevalent OOP decomposition[1].

Mining for Concerns

While contemporary modularization techniques such as OOP have proven to be successful, their approach of modularizing software systems according to a single concern is inherently insufficient and might not provide enough structure for developing complex systems [6, 7, 8]. Concerns not represented in the current system decomposition can decrease the code quality, as they have to be "pressed" into the primary decomposition. We call such concerns *hidden concerns (HCs)*. Code related to these concerns can show two symptoms of poor modularity: it can be *scattered* over the whole project or it can be *tangled* with other code. Code tangling is a state where lines related to different concerns are interwoven.

ASOC techniques promise to overcome these problems by providing constructs to represent otherwise hidden concerns. However, regardless of which ASOC technique is used, software developers face the same problems when applying these paradigms to legacy systems: How to identify and extract the code related to a hidden concern? Due to the scattered nature of hidden concerns, searching for them in existing code is a non-trivial task.

Text-Based Mining

The complex *traces* of HCs[2] can prevent them from being found using traditional, text-based analysis techniques (i.e. pattern matching). Text-based techniques work best if consistent naming conventions for types, methods, variables and classes are carefully followed. When not strictly adhered to, these methods may fail altogether and important parts of hidden concerns might be missed. Unfortunately, legacy code might not comply with such conventions.

Text-Based Analysis

Works better for: Many instances of the same types, with strict naming conventions
Not helpful if: Naming conventions are only partially followed (or not at all)

What is worse, these methods fail quietly. If the majority of the code adheres to naming conventions while the rest does not, the results may be convincing enough not to question them. If the discovered lines were extracted into a module while the others were not, the modularity of the system would not improve but might even get worse.

As an example, consider the code in figure 1. It is a method that creates the button menu for an application that handles files and allows for user-defined system settings. Consequently, we have two different kinds of buttons: those related to system settings and those related to files. Using text-based methods to identify all occurrences of settings buttons would be trivial if naming conventions were followed. The expression

[1] For object-oriented software, the dominant decomposition is into classes (data concerns).

[2] Traces describe the way code related to a hidden concern is spread throughout the project.

`*SettingsButtons` would find them all. On the other hand, if we assume that the code was revised later on (see comment "*Added in rev. 2*" in the figure) and the new code does not comply with the old conventions, we would miss `resetSettings` in (A). Similarly, it would be difficult to find the `newFile` JButton (B) if we are looking for buttons related to file management (`FileButton`)[3].

Type-Based Analysis

Works better for: Many similarly named objects of the different types, no naming conventions

Not helpful if: Many instances of the same type are used for different purposes

Type-Based Mining

An alternative method to identify possible hidden concerns is to analyze the usage of types in the sources. This approach does not suffer from convention-less coding since naming of objects, classes, and variables becomes irrelevant. It is especially helpful if many similarly named types are to be differentiated.

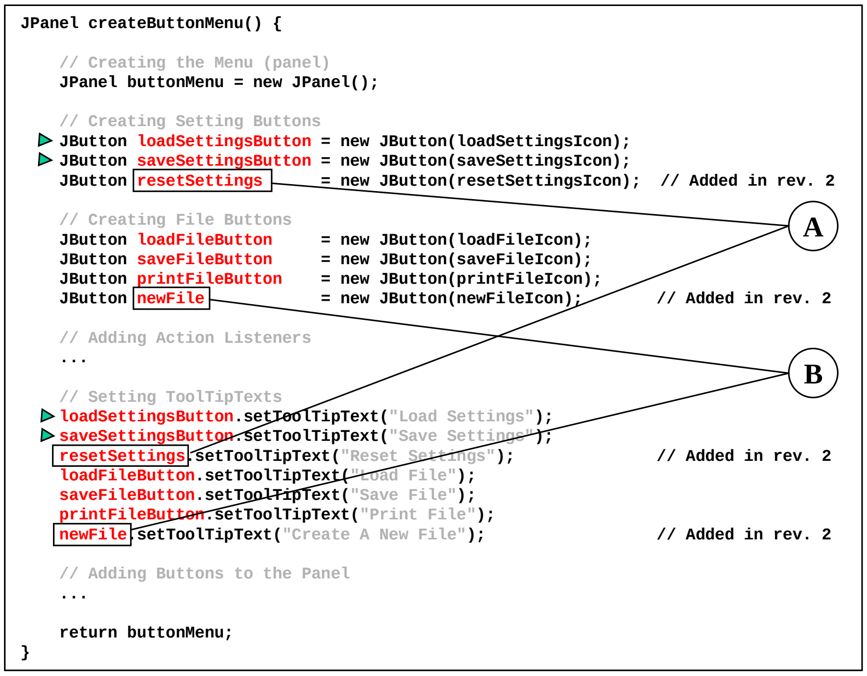
Furthermore, the usage of types can provide hints about meeting modularization design goals. If a programming project is well modularized, subsystems and modules should show high coherence and low coupling [4]. A single module that uses only a few types is likely to have high coherence (strong dependencies within the module since few external types are used) and low coupling (low dependencies between this and other modules). Therefore, type-based analysis can show where these design goals are met and where the project might profit from further decomposition.

Besides, type-based analysis can help to identify code tangling. Repeated, interwoven usage of specific types is a possible indication of tangled code.

But type-based analysis has limitations, too. If we were to search for the same objects as in the example above, we could not differentiate between buttons for settings and those for files. Searching for occurrences of JButton objects would produce too many false positives (marked red in figure 1). However, if we could combine a text-based search for `*Settings*` with a type-based search for JButton, we would find exactly the lines in question.

The comparison above shows that the two analysis methods are not exclusionary but do in fact complement each other quite well. Furthermore, a combination of these techniques can be used to identify method signatures (i.e. a specific type as the method receptor, certain types as method arguments and a text-based search for the method name).

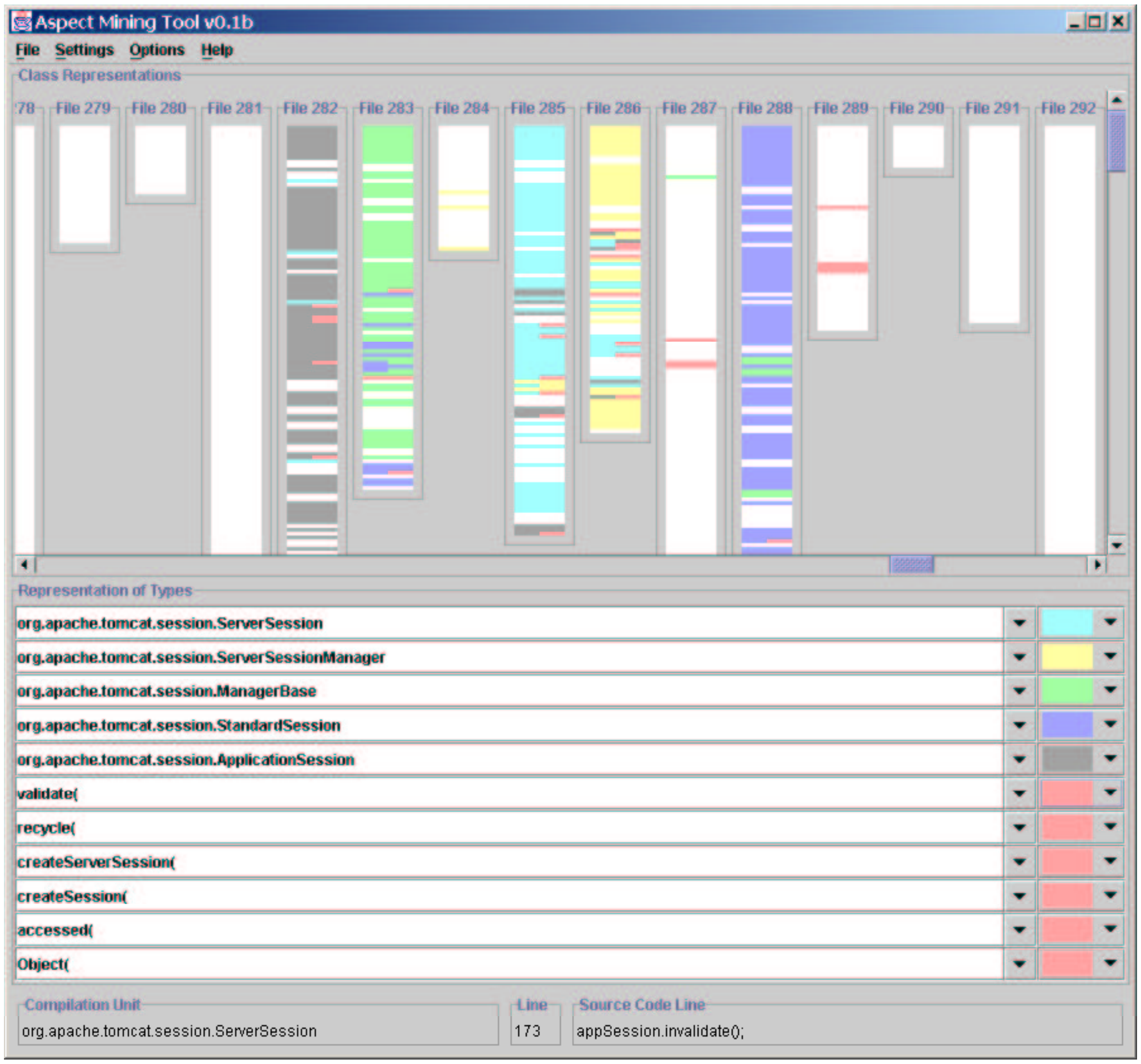
These findings motivated us to develop a multi-modal analysis tool. With the potential to combine different kinds of queries we think the tool is less likely to produce false positives or to miss important lines (false negatives).



```
JPanel createButtonMenu() {

    // Creating the Menu (panel)
    JPanel buttonMenu = new JPanel();

    // Creating Setting Buttons
▷   JButton loadSettingsButton = new JButton(loadSettingsIcon);
▷   JButton saveSettingsButton = new JButton(saveSettingsIcon);
    JButton resetSettings      = new JButton(resetSettingsIcon);  // Added in rev. 2

    // Creating File Buttons
    JButton loadFileButton     = new JButton(loadFileIcon);
    JButton saveFileButton     = new JButton(saveFileIcon);
    JButton printFileButton    = new JButton(printFileIcon);
    JButton newFile            = new JButton(newFileIcon);          // Added in rev. 2

    // Adding Action Listeners
    ...

    // Setting ToolTipTexts
▷   loadSettingsButton.setToolTipText("Load Settings");
▷   saveSettingsButton.setToolTipText("Save Settings");
    resetSettings.setToolTipText("Reset Settings");                 // Added in rev. 2
    loadFileButton.setToolTipText("Load File");
    saveFileButton.setToolTipText("Save File");
    printFileButton.setToolTipText("Print File");
    newFile.setToolTipText("Create A New File");                    // Added in rev. 2

    // Adding Buttons to the Panel
    ...

    return buttonMenu;
}
```

Figure 1: Limitations of text-based and type-based analysis

[3] Searching for other patterns like `*Settings*` or `*File*` would create too many false positives (similarly named Icons, ToolTipText, etc.).



Representation of Types	
org.apache.tomcat.session.ServerSession	🟦
org.apache.tomcat.session.ServerSessionManager	🟨
org.apache.tomcat.session.ManagerBase	🟩
org.apache.tomcat.session.StandardSession	🟪
org.apache.tomcat.session.ApplicationSession	⬛
validate(	🟥
recycle(	🟥
createServerSession(	🟥
createSession(	🟥
accessed(	🟥
Object(	🟥

Compilation Unit	Line	Source Code Line
org.apache.tomcat.session.ServerSession	173	appSession.invalidate();

Figure 2: AMT highlights lines associated with Session Expiration in the TomCat project

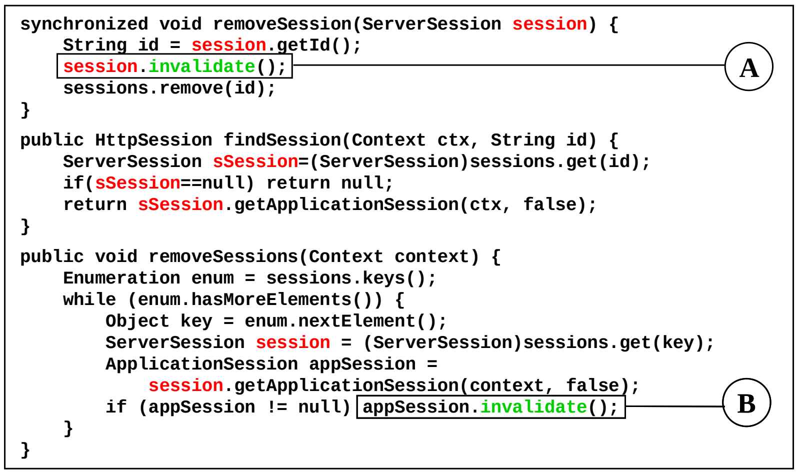
The Aspect Mining Tool

Our Aspect Mining Tool (AMT) is an open multi-modal analysis framework. It was originally developed for mining aspects [6, 3], but is well-suited for identifying hidden concerns in general. It currently provides type-based and text-based analysis techniques, and is extensible. Other analysis methods are under investigation, such as signature-based searches.

Program sources are represented using the Seesoft concept [1,2]. Each compilation unit (i.e. class) itself is represented as a collection of horizontal strips that correspond to the relevant lines of source code[4]. The tool allows user-defined queries based on type usage and regular expressions, displaying matching lines in specific colours. If a line matches more than one criterion, it will be separated into two or more differently coloured parts.

AMT uses a modified version of the AspectJ [3] compiler to extract the type information for each line of source code. To analyze sources, the AMT compiler first collects all the information relevant to the lines of code. The system then displays the extracted information graphically. The user can highlight lines that use selected types, match specific patterns, or a combination of the two.

[4] For the purpose of analysis, "relevant" source code is stripped of comments and empty lines.



```
synchronized void removeSession(ServerSession session) {
    String id = session.getId();
    session.invalidate();
    sessions.remove(id);
}
public HttpSession findSession(Context ctx, String id) {
    ServerSession sSession=(ServerSession)sessions.get(id);
    if(sSession==null) return null;
    return sSession.getApplicationSession(ctx, false);
}
public void removeSessions(Context context) {
    Enumeration enum = sessions.keys();
    while (enum.hasMoreElements()) {
        Object key = enum.nextElement();
        ServerSession session = (ServerSession)sessions.get(key);
        ApplicationSession appSession =
            session.getApplicationSession(context, false);
        if (appSession != null) appSession.invalidate();
    }
}
```

Figure 3: `ServerSessionManager`: A combination of the two analysis techniques identifies the lines of interest

Information Extraction

As an example of the kinds of searches AMT is intended to handle, consider a well-known open source project: TomCat, which is the Official Reference Implementation for Java Servlet 2.2 and JavaServer Pages 1.1.[5]

As most services provided by TomCat have to deal with different (HTTP) sessions, it needs to keep state information associated with them. However, as HTTP itself is stateless, TomCat must explicitly maintain any state information itself. As a starting point for the analysis we suspected that the code associated with session states is likely to be scattered throughout the project.

We focussed specifically on code related to *Session Expiration*, i.e. code responsible for killing sessions when they have been inactive for too long. Session Expiration basically has to deal with three different kinds of session state types: `StandardSession`, `ServerSession`, and `ApplicationSession`. Objects of type `ManagerBase` and `ServerSessionManager` manage the session expiration. Methods related to this concern create, delete or access the types named above.

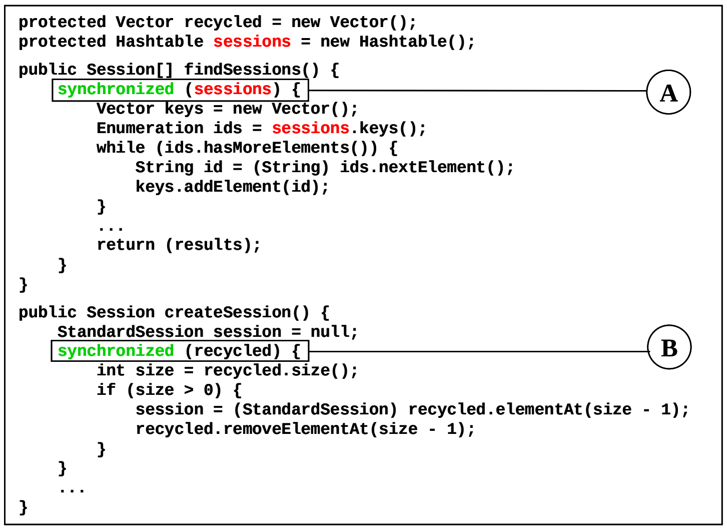
To identify the lines that are likely to be associated with session expiration, we combined type-based and text-based analysis. Figure 2 shows a snapshot of the AMT tool. Lines of interest match both type and text query (i.e., they have two colours).

Looking at the figure, it is apparent that the code related to this concern is spread over a number of different classes. Extracting the relevant lines into a single module may enhance the modularity and maintainability of the code.

Hidden concerns are not always that apparent. If the programmer has no intuition as where to start and what to look for, type-based analysis can help, too. Just by browsing the types used in the project, the developer can see where and in how many classes they are used. If the usage of a specific type is not well localized, it can indicate a possible hidden concern and thus a starting point for extracting HCs.

Benefits of Multi-Modal Analysis

Figure 3 shows a short piece of source code.[6] We are looking for calls of the "invalidate()" method sent to an object of type `ServerSession`. Usages of the `ServerSession` type are in red (identified by type-based analysis), calls to the `invalidate()` method are shown in green



```
protected Vector recycled = new Vector();
protected Hashtable sessions = new Hashtable();

public Session[] findSessions() {
    synchronized (sessions) {
        Vector keys = new Vector();
        Enumeration ids = sessions.keys();
        while (ids.hasMoreElements()) {
            String id = (String) ids.nextElement();
            keys.addElement(id);
        }
        ...
        return (results);
    }
}

public Session createSession() {
    StandardSession session = null;
    synchronized (recycled) {
        int size = recycled.size();
        if (size > 0) {
            session = (StandardSession) recycled.elementAt(size - 1);
            recycled.removeElementAt(size - 1);
        }
    }
    ...
}
```

Figure 4: `ManagerBase`: Serching for synchronized accesses to `Hashtable` objects

[5] TomCat is part of the Apache project [5]. For this example, we used version 1.3 of TomCat

[6] The excerpt is from: `ServerSessionManager.java` in: `org/apache/tomcat/session`

(identified by text-based analysis).

Text-based analysis alone would not have found just the appropriate line (A), but produced false positives. The `invalidate()` method can be received by both `ServerSession` and `ApplicationSession` objects (B). Similarly, type-based analysis would not have been able to identify the proper method call as `ServerSession` objects are used throughout the `ServerSessionManager` source. In this specific case, signature-based analysis would have been ideal, but would be less flexible.

The example also shows that naming conventions for `ServerSession` objects were not adhered to, as even in the short code example we have two different names for `ServerSession` objects: `session` and `sSession`. Type-based methods would manage to find all `ServerSession` objects, but only the combination of the type-based search for `ServerSession` and the text-based search for "invalidate()" manages to capture the affected line of code.

Figure 4 illustrates another useful application of a combination of type-based and text-based searches. Given that we want to identify all synchronized accesses to a specific type (here: `HashTable`), we can simply search for lines both using the `HashTable` type and matching the "synchronized" string (A). Each method alone can produce too many false positives (e.g.: B) in a large project.

Conclusion

Improving modularity of legacy code can be a complex task. Applying ASOC techniques implies extracting code related to concerns that are ill-represented in the systems current decomposition. While some of these hidden concerns are intuitive to identify, others have complicated patterns. No one method excels in identifying all affected fragments. We propose using multi-modal analysis techniques.

The authors introduced type-based analysis as an alternative to text-based analysis and showed its potential benefits for mining hidden concerns. The examples provided show that the two analysis techniques complement each other well and that a combination can prevent false positives in source code analysis.

The Aspect Mining Tool is available for download at

`www.cs.ubc.ca/~jan/amt/`

In its current version, AMT offers text-based and type-based analysis techniques. The tool is still in active development, with future extensions planned, such as hot spot detection (sections of code that use too many different types). Other analysis techniques like signature-based methods are under investigation.

References

1. Griswold, W. G., Kato, Y. and Yuan, J. J. *Aspect Browser: Tool Support for Managing Dispersed Aspects*. Position paper for the First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (at OOPSLA '99)
2. Eick, S. G., Steffen, J. L. and Sumner, Jr. E. E. *Seesoft – A Tool for Visualizing Line-Oriented Software Statistics*. IEEE Transactions of Software Engineering 18(11): 957-968, 1992
3. Kiczales, G., *Aspect-Oriented Programming with AspectJ*. Foundations of Software Engineering, 2000
4. Bruegge, B., Dutoit, A. H. *Object-Oriented Software Engineering*. Prentice Hall, 2000
5. The TomCat project is part of the Apache project. The website is: `http://jakarta.apache.org/`
6. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. and Irwin, J. *Aspect-Oriented Programming*. In: proceeding of the European Conference on Object-Oriented Programming (ECOOP), 1997
7. Tarr, P., Ossher, H., Harrision, W. and Sutton, S. M., *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. In: proceedings of the International Conference on Object-Oriented Programming (ICSE), 1999
8. Harrison, W. and Ossher, H., *Subject-Oriented Programming (A Critique of Pure Objects)*. In: proceedings of the conference of Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), 1993