

The Tyranny of the File Decomposition

João Cachopo,¹ António Menezes Leitão,¹
and António Rito-Silva²

¹Instituto Superior Técnico — DEI
Av. Rovisco Pais, 1049-001 Lisboa, Portugal
{jcachopo, aml}@gia.ist.utl.pt

²INESC-ID/Technical University of Lisbon
Rua Alves Redol nº 9, 1000-029 Lisboa, Portugal
Rito.Silva@inesc-id.pt

Abstract

The unit of concern for most development tools is the file: Text editors open files; Compilers compile source contained in files; Version control systems keep track of changes made to files. Yet, for many tasks, the disposition of code in files is not the best for what developers want to accomplish, but tools do not allow them to change it easily. We call this the tyranny of the file decomposition and propose to replace files with smaller code units, arranged dynamically into code views by the development environment under the guidance of the developer. This permits code arrangements better suited for different tasks, improving development productivity.

1 Introduction

The development of large software systems is a complex task that begs for good software engineering techniques. One such technique—the separation of concerns (SoC) approach—was identified long ago as a fundamental principle [Parnas, 1972], yet proved difficult to be put into practice. Many of the advances in programming languages improved on the support given to the SoC approach. Unfor-

tunately, development tools did not keep the pace with this development: Unlike programming languages, current development tools are not much different than what we had thirty years ago. Yet, they play a central role in modern software engineering. Even more so with the advent of aspect-oriented programming (AOP) [Kiczales et al., 1997].

Using AOP we can develop programs by considering only some concerns (aspects) of the problem at a time. When done properly, this approach will yield concerns simple enough to be well understood and, therefore, developed. This means that large and complex systems will have a huge quantity of such concerns. Although they may be developed independently, they must be put together to build the whole system. To navigate among all this code, a developer needs a helpful development environment. Unfortunately, current environments are rather poor.

Consider, for example, a very simple program in AspectJ [Kiczales et al., 2001] where we have a class to represent rectangles and one aspect, as shown in figures 1 and 2.

Now, assume that a programmer has to look at the `setUpLeftX` method, perhaps to change it somehow. To understand what it does, she/he has to look also to the advice introduced by the `UpdateScreen`

```

class Rect {
    int upLeftX;
    ...

    void setUpLeftX(int ulx) {
        upLeftX = ulx;
    }
    ...
}

```

Figure 1: Rectangle class

```

aspect UpdateScreen {
    pointcut changesFigure():
        receptions(void Rect.setUpLeftX(int));

    after(): changesFigure() {
        DisplayManager.invalidate();
    }
}

```

Figure 2: Aspect

aspect. Ideally, the programmer should view simultaneously both the method and the advice, and nothing else. But each one of those is in a different file, with additional code. This means that, to accomplish such a simple task, the programmer will have to: (1) realize that there is a separate advice for the method; and (2) open both files to view them simultaneously, assuming a window-based environment which supports such things. In such a simple example this is not that hard, but not very productive, either. However, this approach does not scale well for large systems. Two distinct problems prevent that:

- In general, with a large code base the programmer will not be aware of all the aspects that are relevant for a particular method.
- If there are several aspects there will be, correspondingly, several windows (one for each file), which means that the programmer spends most of the time managing windows and jumping between them than in understanding the code. Moreover, each jump to a new window is a context switch that needs an extra effort to find the relevant code and ignore the surrounding irrelevant code.

Clearly, this is an issue to be handled by the devel-

opment environment. The first problem was already addressed by the AspectJ developers. They provide extensions to some development environments that inform the programmer of the aspects that are relevant for a given method. Yet, the second problem remains and it is the more difficult to solve properly.

We would like to view, in a single window, all the relevant code for our task: the method and all the (possibly) relevant aspects. Unfortunately, development tools and files prevent us from doing this. Spreading code over different files according to a particular criteria hinders all the tasks that need a different arrangement of the same code. We call this **the tyranny of the file decomposition**¹ and will expand on this in the next section.

2 The tyranny of the file decomposition

We put the emphasis on the file organization because files are **the editing unit** in development environments, i.e., they are the only unit of code most tools can handle. Unfortunately, files group code together into a unit that is rather large, too small, or just plain wrong for many development tasks. We cannot dismiss this problem by saying that it is just a matter of choosing a good decomposition to start with. The problem is that there is no single best decomposition.

2.1 Why code arrangement matters

The way code is put together is of utmost importance for many tasks of the software development process:

- When editing a class, the developer wants to see all the class code.
- When documenting the program and its design, code should be presented in an order that pro-

¹This is, in fact, a particular case of what Tarr et al. [1999] call the tyranny of the dominant decomposition, applied to the concerns of the development process.

motes its understanding at different abstraction levels.²

- Compilers pose restrictions on the code sequence (for instance, definition of functions before their use) and may be able to optimize code if all the relevant code is contained in a compilation unit.
- During a debugging session, the most relevant code is the active methods within the program current execution.
- Before a program change, like adding a parameter to a function, evaluating the consequences of the change is best accomplished if all the relevant code fragments (in this case, all the call points) can be examined.

The above list is not exhaustive. However, it is enough to the conclusion that different arrangements of code are desirable but, in many cases, they are incompatible with each other. Unfortunately, when we write the code for the first time, we commit ourselves to a particular file decomposition, which becomes cast in stone, therefore making the other tasks more difficult.

3 Code views

As we saw, there is no single code arrangement that works well for every task. Instead, different tasks would benefit from rearrangements of code. Therefore, we argue that a central requirement for a helpful development environment is that it provides ways to generate **code views** dynamically. We call code view to a sequence of small code fragments: the **code units**.

The code units should be the smallest fragments of code that are still meaningful to a developer: Fragments such as methods or field declarations, instead of whole classes.³

²For instance, as taught by the literate programming discipline.

³The use of the term “source code” in this text includes, obviously, its documentation.

Likewise, code views will usually correspond to a meaningful arrangement of code units, suitable for a particular task. For instance, in a debugging session a useful code view during a breakpoint would be the sequence of methods active in the current execution point ordered by their calling sequence.

Actually, according to this definition, files are just particular code views. The code shown previously in figures 1 and 2 are two particular code views. These code views are suitable to feed to the AspectJ compiler, but also as meaningful abstractions to the developer. However, if a developer wants to concentrate on what is executed during a call to `setUpLeftX`, the code view shown in figure 3 may be more appropriate. As a matter of fact, it may be even more perspicuous to see the method as a combination of the method body and all its advice, as shown in figure 4.

```
void setUpLeftX(int ulx) {
    upLeftX = ulx;
}

after(): changesFigure() {
    DisplayManager.invalidate();
}
```

Figure 3: Code view for the `setUpLeftX` method

```
void setUpLeftX(int ulx) {
    upLeftX = ulx;
    DisplayManager.invalidate();
}
```

Figure 4: Code view for the combined `setUpLeftX` method

The code views discussed so far are just a few examples of the many ways that developers want to look at code. Some of these code views may be predefined in the development environment. For instance, all the calls of a given function. However, it is not reasonable to expect that all the relevant code views are preprogrammed in the environment. Therefore, the developer should be able to query the environment to produce the desired code views.

4 Related work and concluding remarks

Back in the eighties, both the Interlisp-D [Teitelman and Masinter, 1984] and the Smalltalk-80 [Goldberg, 1983] programming environments separated the editing, displaying and use of code units from their file representation (for backup and sharing purposes). Code was manipulated using code browsers and editors that managed the code repository on behalf of the developer.

Nowadays, programming environments strongly depend on files. In itself, this is not bad. The problem, as we saw, is when that dependency constrains what developers can do because the only view of the code they have is the file view.

More recently, proposals for programming environments already acknowledge the need for different code views. For instance, the white-paper for the Gwydion programming environment [Fahlman, 1993] mentions that a program should be a complex linked data structure stored in an object-oriented database. Moreover, they advocate the use of multiple ways of viewing and editing code, customized to match the user's preferences and immediate needs. Unfortunately, the Gwydion project was canceled before it accomplished all its goals (delivering only a prototype [Sheets programming environment]).

Many reasons concurred to the demise of sophisticated programming environments, e.g., their market model, the computational power they needed, and the programmers expertise required to operate them. Currently, most of these reasons are no longer valid. Therefore, the time is ripe to rethink interactive programming environments.

References

Scott E. Fahlman. Gwydion: An integrated software environment for evolutionary software development and maintenance. URL <http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/gwydion/docs/htdocs/gwyd%ion/gwydion-overview.html>. 1993.

A. Goldberg. *Smalltalk 80: The Interactive Programming Environment*. Addison-Wesley, 1983.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoaka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, Jyväskylä, Finland, June 1997.

David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. ISSN 0001-0782.

Sheets programming environment. URL <http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/gwydion/docs/htdocs/gwyd%ion/Sheets/>.

Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, Los Angeles, USA, May 1999. ACM Press.

W. Teitelman and L. Masinter. The Interlisp programming environment. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*. McGraw-Hill, 1984.