

Seminar Program Analysis and Transformation

Program Slicing and Sliding for Refactoring

Mirko Stocker, me@misto.ch

January 7, 2009

Slicing is an approach to divide a program into chunks that share a common property, like their contribution to the result of a calculation. Slicing can be used to leverage the power of refactoring tools by providing a deeper understanding of the code. For example, the Extract Method refactoring used to extract a series of statements can—with the aid of a slicing algorithm—be enhanced to allow the extraction of non-contiguous statements. This paper summarizes Ran Etinger’s thesis about “Refactoring via Program Slicing and Sliding”, where he introduces the notion of sliding, a visualization of the slicing procedure based on transparent overhead-projector slides. I also show how several well-known refactorings can benefit from a slicing algorithm.

1 Motivation

Refactoring is the technique to improve the structure, readability and maintainability of source code, or as stated by Martin Fowler [Fow99]: “Refactoring is the process of changing a software system in such a way that it does not alter

the external behavior of the code yet improves its internal structure.”

Every mature integrated development environment should support automated refactoring; helping the user perform refactorings faster and less error-prone. Current refactoring tools are already quite powerful in their abilities to change the representation of source code, but there is a lot of potential for improvement.

For example, the *Extract Method* refactoring—used to extract a series of statements or an expression into a new method—can typically only be applied to contiguous statements, whereas the distinct steps for a single computation are not necessarily consecutive but can be interleaved with other statements that are irrelevant for this very computation. Unraveling such a series of interweaved statements is not possible with current refactoring support in typical integrated development environments.

Program slicing, first defined by Mark Weiser [Wei81], is a means to separate such a strand of different computations.

A slicing algorithm separates code in a given scope into distinct slices. Such a program slice contains only the parts of the program that affect the value of a variable, or a set of variables, and *fades out* the rest of the program. As an example, this can be of assistance in debugging to help concentrating on an interesting subset of the program, as described by Zeller [Zel05].

Having multiple computations mixed into each other is certainly not a property of well designed software, which is why program slicing can be of assistance in refactoring.

The ideas in this paper are mainly based on Ran Ettinger's Ph.D. Thesis [Ett06]. The rest of this paper is organized as follows. First, I will give a coarse overview on the major program slicing paradigms in Section 2, to set the stage for Ettinger's work. In Section 3 I shall explain the concept of program sliding and show the various stages of algorithm development and refinement in the thesis, with special attention to the applicability in practical refactoring tool support. The next part, Section 4, will then take a different view on slicing. On the basis of several refactorings, I show what a slicing algorithm should provide to be of assistance in a refactoring. Section 5 concludes.

It is my hope that this paper can communicate some concepts and ideas for the next generation of more powerful refactoring tools, to make sure developers have the right tools available to keep up with the ever growing complexity of software.

2 Slicing Introduction

There are several specializations of slicing techniques for different usage scenarios. This section shall give an overview to slicing and introduce some terminology we will need later on.

2.1 Static Slicing

Static slices form the basis for many other forms of slicing.

A static slice consists of a point in the program P and a variable V (or a set of variables), the so-called *slicing criterion*. A slicing algorithm should now be able to remove all program statements that do not contribute to the values V at this point P .

This kind of slice is also called a *backward-slice*, since it is concerned with all statements prior to P . In contrary to backward slices, forward slices show which statements and variables are dependant on V at P , enabling a prediction about the effects of manipulating V at P .

There are also other forms of slicing criteria, one could omit V and slice with all variables occurring at P , or by selecting several statements and variables at once.

2.2 Dynamic Slicing

Dynamic slicing takes additional data—runtime information like parameters and concrete values of variables—into account when determining a slice, with the expectation of yielding a smaller slice and thus being more helpful to the programmer working with the code. When using slicing as an assistance in debugging, dynamic slices are a sound re-

finement, but not so much in building tools for refactoring, which typically work purely on data from static analysis.

Other forms of dynamic slicing include conditioned slicing and amorphous slicing. *Conditioned slicing* does not depend on concrete values to narrow down a dynamic slice but works with (e.g. user specified) conditions to describe relevant values. ConSIT [DFHH00] is an implementation of a conditioned slicing system.

Amorphous slicing, as described by Harman and Danicic [HD97], is different from the other presented slicings in that it does not work solely with statement deletion to obtain a slice but, as the name implies, can execute arbitrary transformations on the code, with the goal to simplify a series of statements as much as possible, potentially revealing information that was not obvious from the original syntax.

2.3 Slicing Algorithms

Since slicing was introduced over a quarter century ago, many slicing algorithms for various applications were developed. Many of them based on a representation called a *program dependence graph* (PDG). A PDG illustrates the data and control dependencies of a program. Nodes in the graph show statements and two kinds of directed edges are used to show the data a statement depends on and the control structures that influence the statement [OO84]. To slice such a graph, one simply takes a node and computes all reachable nodes.

Figure 1 shows the *Hello World* of the slicing community [Ett06], a sin-

```
1 sum = 0
2 prod = 1
3 i = 1
4
5 while i <= 5
6     sum += i
7     prod *= i
8     i += 1
9 end
10
11 print sum
12 print prod
```

Figure 1: The *Hello World* of the slicing community, calculating sum and product from 1 to 5 in a single loop.

gle loop that calculates both the sum and the product in a range of integers. The corresponding PDG can be seen in Figure 2 on the next page. Blue-dashed edges show the control flow dependencies, green-solid lines indicate data flow dependencies. To make a slicing example, let us remove every statement that does not contribute to the calculation of the sum. This results in Figure 3 on the following page. As shown by the dimmed vertices and edges, the statements on lines {2,7,12} can safely be ignored for the calculation of sum.

Different program slicing algorithm implementations can also be combined to solve a particular problem. For example, Vidács and colleagues [VJABG08] present a preprocessor slicer that can be used with other C++ slicers: “the structure of macros is defined by using sets and relations, and a dependency graph is defined based on macros”.

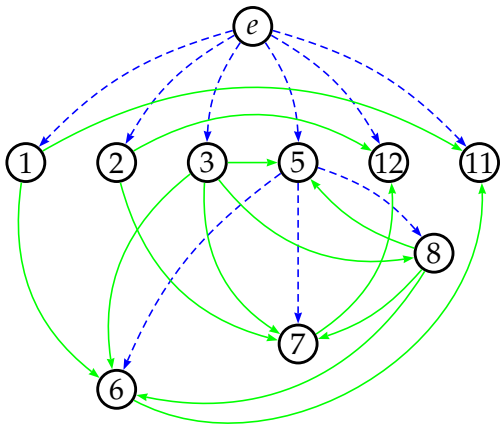


Figure 2: The full PDG of the code in Figure 1 on the preceding page. The names of the vertices are the respective line numbers from the listing, e marking the artificial entry point of the program.

2.4 Static Single Assignment

The static single assignment (SSA) form [Ett06] of a program is typically used in static analysis of source code: each assignment to an already assigned variable creates a new variable name, which is usually indicated by an increasing integer suffix. Figure 4 shows an example of code in SSA form. The SSA form is useful to slicing because it allows to create slices when a single variable is—often unnecessarily—reused in several computations. The refactoring to amend this is called *Split Temporary Variable* [Fow99] and is discussed in Section 4.2.2 on page 15.

The SSA form can also be deconstructed, even after reordering of statements, provided that the instances of a variable have not been shuffled into each other. In other words, considering all in-

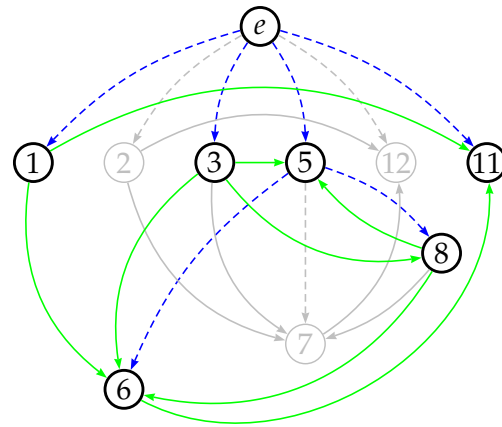


Figure 3: The sliced version of the PDG in Figure 2. The dimmed vertices $\{2, 7, 12\}$ indicate statements that are irrelevant for the calculation of `sum`.

stances with the same subscript as one block, it is safe to move these blocks and de-SSA them afterwards.

2.5 Layout Preservation

It is of importance to the user of a refactoring tool that it does not change more of the source code than absolutely nec-

1	<code>a = 1</code>	<code>a = 1</code>
2	<code>b = a + 1</code>	<code>b = a + 1</code>
3	<code>a = 2</code>	<code>a₂ = 2</code>
4	<code>c = a + 1</code>	<code>c = a₂ + 1</code>

Figure 4: The left column shows the original code and on the right we can see the same in SSA form. On line 3, `a` is assigned a new value and is therefore given a new name—`a2`—that is subsequently used on line 4.

essary. It is thus very valuable to have a suitable representation of a program slice that makes it possible to retain the user’s formatting. While refactoring, we also must not forget elements of the source code that do not have any semantic meaning, like comments, which are usually discarded while generating other representations of the source code. Some strategies on how to preserve comments are described by Sommerlad and my colleagues in [SZCF08]. This goal can be achieved with program sliding.

3 Evolution of the Sliding Algorithm

In this section we shall examine the steps that Ettinger took to develop a provable correct slicing algorithm. Ettinger developed a theoretical framework to prove correct program transformations based on refinement calculus [Bac80] and adopting Dijkstra’s guarded command language [DS90]. I will not delve into the theory too much but just show the algorithm on the basis of examples. The first step duplicates the program and further transformations try to remove as much of this overhead as possible.

3.1 Program Sliding

Ettinger proposes a new approach to visualizing slices based on the metaphor of *transparent slides on an overhead projector*. A union of slices can then be imagined to be a stack of slides, aligned on top of each other. With this metaphor in mind, more operations on slides can be visualized—for example, slides can be duplicated and

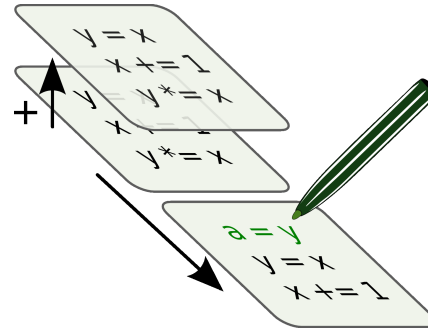


Figure 5: Visualization of program sliding. Code printed on slides can be duplicated, moved around and annotated with a pen.

moved around. Using a water soluble pen, one can write compensatory code onto a slide, and remove it later on during optimization steps. See the Figure 5 for a visualization of the sliding concept.

Slides can be obtained in different ways. One is to create a slide for each variable, including all the control guards that determine whether the assignment is executed. Hence, slides encompass the *control dependence*. Using the SSA form, we can also create a slide for each variable in the SSA form, allowing a finer grained segmentation of the program.

Data dependence is expressed through the notion of slide dependence and independence. Slides are called dependent if their sets of variables are not disjoint.

Stacking all the slides results back in the original program representation.

3.2 Co-Slice

Separating a slice from a program also creates a complimentary slice [GL91], a so-called co-slice, that contains the rest of the statements that are not extracted (if

our goal is to extract the slice). It can be unavoidable to create some duplication in the extracted slice and the co-slice, but a good algorithm should try to minimize the duplication, as we shall see in the course of the next sections.

3.3 Statement Duplication

Statement duplication is the first, quite primitive, step in the evolution towards a more sophisticated algorithm and is only able to duplicate whole statements, no code is deleted yet.

From the set of variables V that are used in the (potentially compound) statement S , we want to extract the computation of the variable w . Because no code is deleted, we decide whether a variable is calculated in the extracted slice or the co-slice only through the assignment (and overwriting) of backup variables.

First, we need to introduce some new variables: for each variable in V , create a new one (this is the set V_{backup}) to hold a copy of the initial values and introduce a variable to hold the result of the extracted computation: w_{backup} .

We can now extract the slice of w by creating a new program that performs the following steps:

1. Initialize V_{backup} .
2. Copy all variables in V to V_{backup} .
3. Execute S , the original compound statement.
4. Backup the computed w to w_{backup} .
5. Restore V from V_{backup} .
6. Execute S again.

```

1 sum, prod, i = 0, 1, 0
2
3 while i <= 5
4   sum, prod, i = sum + i, prod * i, i++
5 end

```

Figure 6: The listing shows the starting position for our duplication example, it is similar to the listing in Figure 1 on page 3, slightly reformatted for conciseness.

7. Restore w from w_{backup} .

All variables in V now have exactly the same values as before the duplication, with the addition that the final value of w is calculated in the first execution of S . Figure 6 and Figure 7 on the next page show an example application of this algorithm.

This first algorithm can also be expressed in the terms of the sliding metaphor: the original slide is copied, both are composed sequentially, and compensatory code for the backup and final variables is written on the slides with a pen.

Unfortunately, this algorithm considerably increases the size of the code and creates a lot of duplication, which is contradictory to our original goal of improving the design of the code. Therefore, the next step is to remove unnecessary code from the duplicated statements. A means would be to slice both S and its second execution with regards to sum and $prod$, respectively. This is based on a liveness analysis, and will be applied in a later stage of the slicing algorithm.

In the next sections, we will contin-

```

1 sumb, prodb, ib = sum, prod, i
2
3 sum, prod, i = 0, 1, 0
4
5 while i <= 5
6   sum, prod, i = sum + i, prod * i, i++
7 end
8
9 sumresult = sum
10
11 sum, prod, i = sumb, prodb, ib
12
13 sum, prod, i = 0, 1, 0
14
15 while i <= 5
16   sum, prod, i = sum + i, prod * i, i++
17 end
18
19 sum = sumresult

```

Figure 7: The listing shows the produced program after statement duplication: it contains the backup variables (indicated with a subscript _b) on line 1, which are used to restore the initial state on line 11. Lines 9 and 19 show how the backup and restoration of the extracted computation of sum works. The background color highlights the duplicated slides.

```

1 sum, i = 0, 0
2
3 while i <= 5
4   sum, i = sum + i, i + 1
5 end
6
7 prod, i = 1, 0
8
9 while i <= 5
10  prod, i = prod + i, i + 1
11 end

```

Figure 8: Calculating the slides for sum, the algorithm can only get rid of the underlined prod variables. The second loop, although it is not needed anymore, is still included because sum unconditionally depends on i.

uously refine this crude algorithm and show how the number of backup variables can be reduced and how to get rid of unnecessary calculations.

3.4 Flow-Insensitive Slicing

The flow insensitive slicing algorithm yields correct slices, but not very elegant ones. Starting from a variable's slide V , it includes all the slides of other variables that V depends on. All variables that are not included in the slide are independent of V and can thus be discarded. Figure 8 shows an example of this.

To find a smaller slice, we need to turn to flow-sensitive slicing.

3.5 Flow-Sensitive Slicing

Making use of the SSA form, the same algorithm as described before, can now isolate just the first loop (lines 1–5 in Figure 8 on the previous page), as one would expect (remember that the SSA form basically splits a variable into all its usages, creating distinct variables). Similarly, if applied to slice `prod`, only the lines 7–11 remain.

3.6 Co-Slicing

In its current form, the algorithm still creates too much duplication: any calculation that is used by both, the extracted slice and its complement, will be duplicated. Also, if an extracted variable— w —is used in the complement, it will be calculated again (in the second execution of the statement, say w_2), which is completely unnecessary. Thanks to the backup variables we created and stored in w_{backup} , the results of the extracted variables can be reused. And because their values do not change anymore, all final usages of w_2 can be replaced by w_{backup} .

Although this alone does not decrease the amount nor simplify the code—quite the contrary, it forces a rename step from w_2 to w_{backup} —it renders the duplicated calculation of the extracted variable w_2 unnecessary. Non-final variables can also be offered for reuse in the complement. For now, we assume that the user of the algorithm, viz. the refactoring, has to decide which variables are to be reused. We will lift this restriction later on.

This substitution step is called *final-use substitution*, because the variable is guar-

anteed to have received its final value. The dead code can then be removed by slicing on the variables from the complement.

A detailed example of this procedure can be found in Figure 9 on the following page.

While the final result of the transformation contains less duplicated code than before, it still has an awful lot of backup variables and potentially some variables had to be renamed. In the next section, we will take a look at penless sliding to help eliminate the compensatory code.

3.7 Penless Sliding

In this section we shall examine penless sliding—that is, creating slides without the need of using a pen to write backup variables on the slides.

In our previous example in Figure 9 on the next page, we can still see a lot of backup variables in the rightmost column. The same algorithm that is used to convert code back from the SSA form can also be used to merge the backup variables, “if they are never simultaneously-live, never defined in the same assignment, and one is never defined in an assignment where the other is live-on-exit” [Ett06, Page 116].

A variable is *live* at a point P in the program if there is a usage of the variable between P and its last (re-)definition. Similarly, a variable is *live-on-exit* if it is used after some point and before it is re-defined. A variable that is not live is *dead*.

For example, considering `sumresult` and `sum`: `sumresult` is live from lines 13–27, but in this range, `sum` is dead, so these

<hr/> <pre> 1 i, sum, prod = 0, 0, 1 2 3 while i <= 5 4 sum, prod, i = 5 sum+i, prod*i, i+1 6 end 7 8 print sum, prod </pre> <hr/>	<hr/> <pre> 1 i, sum, prod = 0, 0, 1 2 3 <u>sum_b, i_b, prod_b =</u> 4 <u>sum, i, prod</u> 5 6 i, sum = 0, 0 7 8 while i <= 5 9 <u>sum, i =</u> 10 <u>sum+i, i+1</u> 11 end 12 13 <u>sum_{result} = sum</u> 14 15 <u>sum, prod, i =</u> 16 <u>sum_b, prod_b, i_b</u> 17 18 i, sum, prod = 0, 0, 1 19 20 while i <= 5 21 <u>sum, prod, i =</u> 22 <u>sum+i, prod*i, i+1</u> 23 end 24 25 print sum, prod 26 27 <u>sum = sum_{result}</u> </pre> <hr/>	<hr/> <pre> 1 i, sum, prod = 0, 0, 1 2 3 <u>sum_b, i_b, prod_b =</u> 4 <u>sum, i, prod</u> 5 6 i, sum = 0, 0 7 8 while i <= 5 9 <u>sum, i =</u> 10 <u>sum+i, i+1</u> 11 end 12 13 <u>sum_{result} = sum</u> 14 15 <u>sum, prod, i =</u> 16 <u>sum_b, prod_b, i_b</u> 17 18 i, prod = 0, 1 19 20 while i <= 5 21 <u>prod, i =</u> 22 <u>prod*i, i+1</u> 23 end 24 25 print <u>sum_{result}</u>, prod 26 27 <u>sum = sum_{result}</u> </pre> <hr/>
---	---	---

Figure 9: Starting from the listing on the left, applying flow-sensitive slicing would yield the code in the second listing. Notice the underlined (superfluous) calculation of `sum`, which can be eliminated by replacing the final uses of `sum` with `sumresult` and eliminating the now dead code by slicing on the other variables, which results in the listing on the right.

two can be safely merged.

In the same manner, we can eliminate all other backup variables: `prod` on line 15 is dead, so we can remove the assignment. For the listing in Figure 9 on the preceding page, all backup variables can be eliminated, as well as the unneeded assignment on the first line.

Unfortunately, penless sliding is not possible in every case. From the following listing, we'd like to slide out the second line.

```
1 y = x+1
2 x *= 2
3 y += x
```

The problem is that `y` depends on an intermediate value of `x`, so this transformation would be wrong:

```
1 x *= 2
2
3 y = x+1
4 y += x
```

In this case, we need backup variables to restore `x`'s initial state for the computation of `y`—for example, this might result in the following non-penless transformation:

```
1 xb = x
2 x *= 2
3 xresult = x
4 x = xb
5 y = x+1
6 y += xresult
7 x = xresult
```

Unfortunately, this transformation more than doubled our original code size.

3.8 Optimal Sliding

Until now, we assumed that the subset of extracted variables to be made reusable in the complement is known in advance—for example by user selection. In the last refinements of the algorithm, Ettinger introduces an algorithm for finding the largest set of variables that can be reused. It begins by trying to reuse all variables and then gradually removes variables that do not lead to a penless slide.

Another potential for the reduction of redundant computations exists if a variable's value is final after the execution of the extracted slice, this is the case when the complete slice of a variable is fully contained in the extracted slice. This means that final-use substitution can be applied to all variables that are fully computed in the extracted slice, which will result in a smaller complement.

3.9 Conclusion and Limitations

Ettinger developed a provable correct slicing algorithm to extract slices of code. Through various refinements, side effects (artificially introduced backup variables) of the transformations were mitigated, resulting in slices with a high reusability of computed results and little to no duplication. But we must not forget that the algorithm is proved to work only on a toy language with several limitations.

In the early stages of the algorithm, variables were cloned to hold initial- and final values. Because cloning of whole objects is generally not feasible, penless sliding works without cloning, so the transformation would still be doable or

needs to be rejected, depending on the situation.

Exceptions are another problematic area. Reordering statements could affect the order of potentially thrown exceptions, which might have an influence on the program's behavior. In this case, a different extraction strategy could be applied, as described by Ettinger and Verbaere [EV04] (the approach is to create a method object that holds a method per contiguous sequence of statements from the extracted slice and insert calls to the method object in the original source, therefore exactly preserving the sequence of calls). Similar concerns arise with regards to concurrency, where reordering of statements might not be desired.

Another restriction of the language is the exclusion of side effects because of unforeseeable consequences caused by duplicated statements. The requirement of having a language with only pure functions could be mitigated by employing algorithms to decide whether a certain function is side-effect free. For example, Finifter and colleagues [FMSW08] present such a technique that decides functional purity for a subset of Java.

Apart from these restrictions, the metaphor of slides is a very nice illustration for slicing algorithms and operations on slices. Sadly, nobody is currently working on a continuation of this work; "Moving on to the real world," Ettinger wrote me, "we dropped the framework (at least for the moment) and moved to base the transformations on the (somewhat related) plan calculus" and that "the proof of correctness for the slicing algorithm is based on some form of op-

erational semantics, not predicate transformers. And the sliding proof of correctness is similar in nature, but not completed yet" [Ett08].

Using what we have learned up to this point, the remainder of this paper will focus on refactorings and show how a slicing algorithm can be of use.

4 Refactoring's Perspective

In this second half of the paper, we will take the refactoring's perspective and try to evaluate, for several refactorings, the requirements for a slicing algorithm so the refactoring can benefit from it. The conjecture is that requirements differ between refactorings and that some refactorings can already benefit from a low fidelity slicing algorithm.

The analysis started from Martin Fowler's list of refactorings [FDH⁺08], most of which can also be found in Fowler's book [Fow99]. In a first iteration, I went through all refactorings, collecting those that could probably benefit from slicing—from the initial ninety refactorings, about a dozen remained. Unsurprisingly, these refactorings can be separated into two categories—*extract* refactorings and those who work in the scope of a single method.

Local scope refactorings are easier to work with than larger ones, but they are also less useful to the user; everything that happens within a few lines of code can also be done by hand without invoking a tool. Of course, slower and more error prone, but still manageable. Larger scale refactorings are far scarier, and therefore often neglected by devel-

opers who do not trust their codebase (be it for the lack of tests or understanding). It is also harder to overview and visualize refactorings that work on the scope of whole classes distributed over several files and containing hundreds of lines of code.

However, in large scale reorganizational refactorings lies far more potential to significantly improve code quality and to make a bigger impact on the overall structure of the code.

4.1 Extract Refactorings

All refactorings that extract code are primary candidates for slicing, as both activities have the effect of taking the code apart. Slicing allows a much finer grained control over what to extract.

It can also be of assistance to the user when used to visualize dependencies in the object under dissection.

4.1.1 Extract Method

The typical slicing example and also one of the most used refactorings (at least for Java development in Eclipse, according to [MKF06]). Current IDEs only support the extraction of continuous statements or a single expression. Slicing adds the benefit of extracting non-contiguous statements—for example, slicing allows us to extract only the first and last statements of this code snippet:

```
1 name = "John"  
2 age = 25  
3 print name
```

Slicing can also give a new view on the refactoring by automatically provid-

ing sensible candidates or expansion of the user's selection. Many programming languages suffer from the limitation of having only one single return value, slicing's dependency analysis capabilities could help by including more statements to reduce the number of needed return values—for example by making suggestions to expand the user's selection. In this listing, it might make sense to also extract the declaration of age instead of passing it into the new method through an extra argument.

```
1 name = "John"  
2 int age  
3 age = 25  
4 print age  
5 print name
```

When slicing results in new methods, duplication might even be acceptable—for example for control structures that have previously been shared (see the *Split Loop* refactoring on page 14).

4.1.2 Extract Class or Package

According to the GRASP patterns [Lar04], a class should have a high cohesion—that is, having one single purpose and just the necessary functionality and data to fulfill it. A low cohesion value means that the class has too many responsibilities that don't have strong interdependencies. By slicing the whole class, clusters of fields and methods that belong together can be identified—and extracted if necessary. Figure 10 on the next page shows an example of the refactoring.

Chen and Xu [CX01] present an algo-

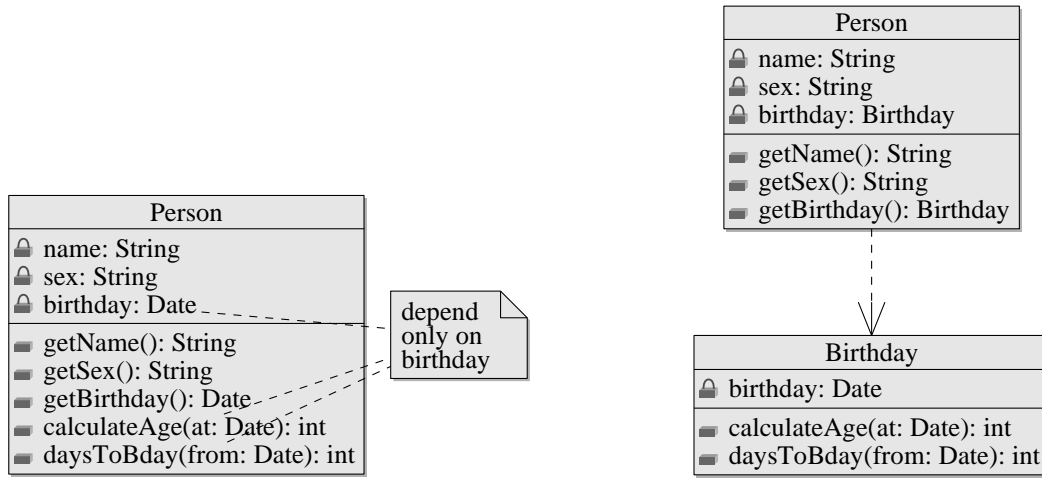


Figure 10: We start with one large class on the left and extract the birthday related code into a separate class, resulting in the drawing on the right.

rithm to slice Java programs. They work with one PDG per method and model dependencies between methods via their parameters, using a call graph to find the correct order of evaluation. Using this mapping, it should be possible to identify disjoint subsets of a class' PDGs.

4.1.3 Extract Subclass

On first glance, extracting a class and keeping it in the same hierarchy seems not much different from the already discussed *Extract Class* refactoring. But the perspective is what distinguishes them—*Extract Class* separates functionality that negatively influences cohesion into a separate class. When extracting into a subclass, what to extract is determined by the clients of the class. Functionality only used by some clients should be moved into a subclass. After all, the ability to specialize behavior through inheritance is a key paradigm of object oriented construction.

A suitable algorithm should slice a class by taking statements of client code and then figure out the relevant parts of the class, from this client's view. Performing this for several clients, a refactoring should then be able to see different patterns of usage and determine the parts of the class that can be moved into a subclass.

Frank Tip and colleagues present in [TCFR96] an algorithm for C++ that can eliminate unused parts of a class, even for complicated—diamond shaped—inheritance structures. The intent is to reduce the size of library classes when only a subset of the functionality is used, thus decreasing the overall size of a program. It might be possible to adapt their algorithm for an intelligent *Extract Subclass* refactoring.

4.1.4 Extract Superclass

Extract Superclass is not simply the opposite of *Extract Subclass*. While the later ex-

tends a class hierarchy, *Extract Superclass* merges completely independent classes, or even whole hierarchies of classes. In a traditional implementation of this refactoring, slicing isn't needed. But thanks to the ability of a slicing algorithm, we do not need to look for common methods that can be moved into a superclass, but simply (series of) statements that are equal and separable from the rest of the method.

This is similar to *Pull Up Method* and *Form Template Method*, where we can automatically extract common code and then move it into a new superclass.

Problematic are single inheritance languages that do not support the concept of mixin-class composition, or modules, that can hold shared code.

4.1.5 Pull Up Method

Pull Up Method works very similar to *Extract Superclass*, just without introducing a new common superclass and only for a single method, but the same strategy of finding sub-method similarities applies as well.

4.1.6 Form Template Method

Given “two methods in subclasses that perform similar steps in the same order, yet the steps are different” [Fow99, Page 280], then you should apply the *Form Template Method*.

Slicing can help to divide a method into smaller chunks to make it easier to find *similar steps* in calculations. Once these steps are identified, applying *Extract Method* and *Pull-Up Method* complete the refactoring.

Komondoor and Horwitz [KH01] have implemented a tool to detect duplicated pieces of code based on PDGs. Applying such an algorithm to a class hierarchy should find duplications which the refactoring could then try to merge.

Figure 11 on the following page shows the classes `XmlTag` and `WikiTag` with the method `render`. They both perform a very similar task—concatenating some string. In both `render` methods, slicing can yield the different parts and the refactoring can create new methods from the slices. The now identical `render` method can be moved to the parent class, resulting in the hierarchy on the right.

4.1.7 Remove Parameter

Remove Parameter doesn't need a slicing algorithm, but it is included in our discussion because it can benefit from a simple live-analysis to determine whether a parameter is dead. Of course, and that's the refactoring's responsibility, removing parameters mustn't break any contracts given by interfaces or parent classes.

4.2 Local-Scope Refactorings

Not all refactorings have large influences on the codebase, local-scope refactorings make changes only in the body of a single method. The advantage of these refactorings is, with regards to slicing, that they can already use a simple algorithm that does not need to know about classes.

4.2.1 Split Loop

The example we used in most listings in this paper is an application of the *Split*

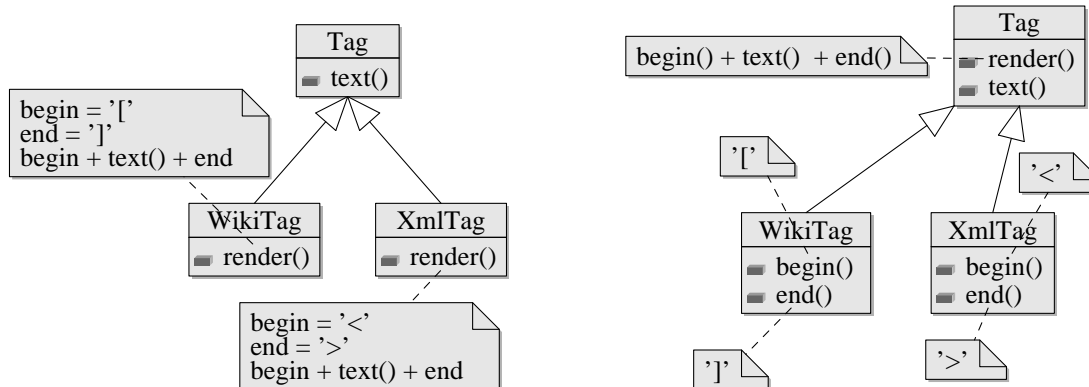


Figure 11: An example of the *Form Template Method* refactoring. In the original classes on the left, both children perform a similar operation. After the refactoring, the common behavior (the *template*) is moved to the parent class.

Loop refactoring and shown for example in Figure 9 on page 9.

4.2.2 Split Temporary Variable

We call a variable temporary if it holds intermediate results from an ongoing computation, hence they are usually local to a method. If multiple temporary values are assigned to a single temporary variable, *Split Temporary Variable* gives each a distinct name—except for loop variables and other collecting variables, which a refactoring implementation has to take care of.

The conversion to the SSA form (as described in Section 2.4 on page 4) does exactly the same thing as the refactoring—giving a distinct name to each usage of a variable. This refactoring can therefore be a side product of a more complicated refactoring implementation, as it was the case in my diploma thesis [CFSS07].

This code is a typical example of the *Split Temporary Variable* refactoring:

```
1 temp = 2 * (_height + _width)
```

```
2 print temp
3 temp = _height * _width
4 print temp
```

The code can be made much more expressive when each usage of `temp` has a name that actually describes its purpose.

4.2.3 Replace Assignment with Initialization

A local variable is declared or defined at the beginning of the method, but not used until much later in the code—usually a bad leftover habit from older languages (e.g. C90) that required this style of declaration. Slicing can be used to identify all variables that are introduced too early, simply by determining whether there are other statements in the calculation between a declaration and its first usage, as shown in the following listing.

```
1 def method
2   i, prod, sum, result = 0, 1, 0, 0
3
4   ... calculations not involving result
```

```

5
6   result = prod + sum
7   return result / 2
8   end

```

The `result` variable isn't used until much later in the method, there is no reason to define it at the beginning of the method. Apply the *Replace Assignment with Initialization* refactoring to amend.

The refactoring must take care not to accidentally reduce the scope of a variable, this is the duty of the next refactoring, *Reduce Scope of Variable*.

4.2.4 Reduce Scope of Variable

Having variables with a broader scope than necessary—for example a variable used only in a loop but defined outside it and not used to count or carry state—is not good style and should be fixed. The refactoring is very similar to *Replace Assignment with Initialization* described in the preceding section.

A slicing algorithm can be used on each subscope to find variables that are defined or declared in the outer scope. The refactoring needs to be aware of collecting variables that are used over multiple runs of a loop.

The refactoring could also be extended to consider instance variables that are only used in a single method and not visible outside the class.

4.3 Refactoring Hints

We have seen that many refactorings can profit from a slicing algorithm. More than that, applying slicing can even lead to refactorings that do not need any user

input anymore—for example *Replace Assignment with Initialization* or *Reduce Scope of Variable*, but also *Split Temporary Variable*. On a larger scale, one might even include the various extract refactorings, given helpful metrics to determine when to apply the refactoring—for example a cohesion analysis. Chae and Kwon [CoST98] introduce the *most cohesive component analysis*, “the most cohesive form of a class”, and “the connectivity factor to indicate the degree of the connectivity among the members of a class”, which might be used to find classes that could be split.

These refactorings could be suggested proactively by an IDE, running an analysis on the source code in the background and displaying a *refactoring hint* to the user whenever an application of a refactoring seems adequate.

Let's take a look at an example that shows the idea, some factorial calculations and printing.

```

1  first, second, result, i = 1, 1, 0, 1
2
3  for i in 1..10
4    first *= i
5  end
6
7  for i in 1..20
8    second *= i
9  end
10
11 result = second * first
12
13 print first + "*" + second + "=" + result

```

There are several smells in this code: (1) all variables are declared and initialized at the beginning, (2) variable `i` is never needed in the scope it is currently

defined, (3) variable `result` is dead until redefined at line 11 and (4), there are far too many things happening for just one single method. After applying the refactoring hints your future IDE gives you, (1), (2) and (3) should be fixed and the code now looks like on the following listing.

```
1 first = 1
2 for i in 1..10
3   first *= i
4 end
5
6 second = 1
7 for i in 1..20
8   second *= i
9 end
10
11 result = second * first
12
13 print first + "*" + second + "=" + result
```

Now the code looks much more compact, but the first two loops share no data. The IDE determines this by slicing on `first` and `second`. Another refactoring hint might now bring to the programmer's attention that the first two calculations could be extracted into their own methods, finally resulting in this listing (assuming the extraction can also detect duplication and introduce parameters as needed).

```
1 def factorial x
2   result = 1
3   for i in 1..x
4     result *= i
5   end
6   return result
7 end
8
9 first, second = factorial(10), factorial(20)
```

```
10
11 result = first * second
12
13 print first + "*" + second + "=" + result
```

This concludes our example on refactoring hints. We now have a new method with a clear purpose and a few lines of code that use it. Of course, for this example, it might be even better to have a refactoring that analyzes the semantics of our code and uses an already existing factorial library function, but that is out of the scope of this paper.

5 Conclusion and Outlook

In the first half of the paper, we learned how Ettinger's provable correct sliding algorithm was developed in his thesis. We started from a crude duplication algorithm and arrived at penless and optimal sliding.

The second half of the paper looked at some refactorings and showed how slicing can be of use to improve these refactorings. We also introduced the notion of refactoring hints—letting the IDE notify the user when a refactoring might be sensible and concluding with a larger example on refactoring hints.

It is my hope that this paper was able to communicate some ideas that could lead to the next generation of automated refactoring tools, towards development environments that not only do the developer's bidding but are also able to assist him and point out possible improvements in the code.

References

- [Bac80] Ralph-Johan Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, The Netherlands, 1980.
- [CFSS07] Thomas Corbat, Lukas Felber, Mirko Stocker, and Peter Sommerlad. Ruby refactoring plug-in for eclipse. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 779–780, New York, NY, USA, 2007. ACM.
- [CoST98] Heung Seok Chae, Korea Advanced Institute of Science, and Technology. A cohesion measure for classes in object-oriented systemsyong rae kwon. In *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*, page 158, Washington, DC, USA, 1998. IEEE Computer Society.
- [CX01] Zhenqiang Chen and Baowen Xu. Slicing object-oriented java programs. *SIGPLAN Not.*, 36(4):33–40, 2001.
- [DFHH00] Sebastian Danicic, Chris Fox, Mark Harman, and Rob Herons. Consit: A conditioned program slicer. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 216, Washington, DC, USA, 2000. IEEE Computer Society.
- [DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.
- [Ett06] Ran Ettinger. *Refactoring via Program Slicing and Sliding*. PhD thesis, University of Oxford, 2006.
- [Ett08] Ran Ettinger. private communication, 2008.
- [EV04] Ran Ettinger and Mathieu Verbaere. Untangling: a slice extraction refactoring. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 93–101, New York, NY, USA, 2004. ACM.
- [FDH⁺08] Martin Fowler, Gerard M. Davison, Mats Henricson, Ashley Frieze, Ivan Mitrovic, Dave Whipp, Marian Vittek, and Bill Murphy. Alpha list of refactorings. www.refactoring.com/catalog, last visited November 2008.
- [FMSW08] Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable

- functional purity in java. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 161–174, New York, NY, USA, 2008. ACM.
- [Fow99] *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.
- [HD97] Mark Harman and Sebastian Danicic. Amorphous program slicing. In *WPC '97: Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, page 70, Washington, DC, USA, 1997. IEEE Computer Society.
- [KH01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag.
- [Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.
- [MKF06] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83, 2006.
- [OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGSOFT Softw. Eng. Notes*, 9(3):177–184, 1984.
- [SZCF08] Peter Sommerlad, Guido Zraggen, Thomas Corbat, and Lukas Felber. Retaining comments when refactoring code. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 653–662, New York, NY, USA, 2008. ACM.
- [TCFR96] Frank Tip, Jong-Deok Choi, John Field, and G. Ramalingam. Slicing class hierarchies in c++. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 179–197, New York, NY, USA, 1996. ACM.

- [VJABG08] László Vidács, Judit Jász, Árpád Beszédes, and Tibor Gyimóthy. Combining pre-processor slicing with c/c++ language slicing. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 163–171, Washington, DC, USA, 2008. IEEE Computer Society.
- [Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
- [Zel05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.