Refactoring Tools and Their Kin

Friedrich Steimann(✉)

Lehrgebiet Programmiersysteme, Fernuniversität in Hagen, Hagen, Germany
steimann@acm.org

Abstract. Refactoring is the process of changing a program in such a way that its design improves with respect to some specific goal, while its observable behaviour remains the same. Trivially, the latter includes the preservation of the program's well-formedness, since arguably, a malformed program has no behaviour to be preserved.

While the problem of refactoring is easily stated, casting it into fully functional refactoring tools for contemporary programming languages is surprisingly hard. In fact, most refactoring tools in use today cannot even guarantee to preserve well-formedness, let alone behaviour, not even for some of the most basic refactorings (such as RENAME or PULL UP MEMBER).

In Part I of this briefing, I will report on some of the most promising techniques for implementing correct refactoring tools. Common to these techniques is that they give up the notion of behaviour preservation in favour of the more basic (and less demanding) notion of invariant preservation: to be correct, a refactoring tool must not accidentally change the binding of names, the overriding of methods, the synchronization on a monitor, etc. Preservation of well-formedness is then the preservation of invariants relating to well-formedness.

With invariant preservation tackled, it is straightforward to transfer refactoring technology to other programming tools, including tools for automatic repair and completion of programs, mutation testing, and program generation. How these are related to refactoring tools, and how they can be developed in concert, I will propose in Part II of this briefing.

Part I: Refactoring Tools

The term *refactoring* refers to at least:

– a discipline (e.g., when it is used as the label of a session at a conference),
– an activity (when somebody is practicing that discipline),
– the result of such an activity (e.g., one program is said to be a refactoring of another),
– a pattern of such an activity (for instance, an element of a refactoring catalogue; e.g., RENAME FIELD),
– an instance of such a pattern (when the pattern is applied to concrete code; e.g., "after a RENAME FIELD refactoring"); and
– a programming tool (as in "Which refactorings does your IDE come with?").

© Springer International Publishing AG 2017
J. Cunha et al. (Eds.): GTTSE 2015, LNCS 10223, pp. 179–214, 2017.
DOI: 10.1007/978-3-319-60074-1_8

In Part I of this briefing, I will focus on *refactoring tools*, and will use this term throughout for disambiguation from all other meanings of *refactoring*. That said, I will start with some observations and remarks on refactoring in general.

1 Origins of Refactoring

Refactoring as a tool-supported discipline goes back to the independent works of William Griswold (with the late David Notkin; see, e.g., [19]), and of William Opdyke (with Ralph Johnson; see, e.g., [30]). Although Griswold's PhD thesis on program restructuring ([18], from 1991) pre-dates Opdyke's ([29], from 1992), the latter is usually cited as the origin of refactoring, not least because it has the term in its title (another reason may be that with C++, Opdyke addressed a more widespread language than Griswold, who addressed Scheme). Although Opdyke reports that the term was coined some time earlier, it wasn't before the implementation of the Smalltalk refactoring browser [34], the adoption of refactoring as a core practice in XP [4], and Fowler's widely recognized book [12] that it became commonplace.

Refactoring as a manual activity is probably as old as programming itself [26]. Especially in the old days, when computing resources where scarce, programs had to be restructured regularly so as to reduce memory usage (both program and data) and execution time. In these days, instruction sets and programs were small, and development environments consisted largely of brains, pens, and paper; refactoring was an inherent part of coding, which required the code to be concise enough to meet the tight space requirements of the machine.

In later years, the task of making the most out of the given hardware resources was shifted from programmers to compilers, specifically to optimizing ones. Refactoring shares with compiler optimization the goal (and problem!) of behaviour preservation. However, compiler optimizations are often local and can be switched off. The latter is a concession to the state-of-the-art, namely that guarantees of behaviour preservation are hard to give. Refactorings, on the other hand, are often non-local; in fact, while coding along, the non-local changes disrupt the workflow most, so that their automation promises to be the most rewarding.

2 The Current Refactoring Crisis

Following its original conception, refactoring is today still mostly perceived as *improving the design of a program while preserving its observable behaviour*. Naturally, this definition is challenged by two questions:

1. What does *improving the design* mean?
2. What does *preserving the observable behaviour* mean?

While both questions appear natural, if not mandatory, to ask in purely academic circles, the current mindset of the refactoring community, a sound mixture of researchers and practitioners, is perhaps best characterized by a third one:

3. Who cares?

2.1 The Elusiveness of Design Improvement

Many refactorings presented in Fowler's catalogue [12] are complemented by reverse refactorings: Extract Method — Inline Method; Pull Up Field — Push Down Field; Replace Inheritance with Delegation — Replace Delegation with Inheritance, to name a few. This fact alone suggests that improving design is not in the nature of specific refactoring patterns: what results in good design in one context can result in bad design in another. Furthermore, since refactoring tools today usually tackle only fairly small changes from which bigger refactorings can be manually composed, each application of a refactoring tool by itself may result in unimproved design: as with solving Rubik's cube, intermediate steps may let the program look dramatically worse temporarily.

A more neutral goal of refactoring is therefore to make subsequent changes easier. However, even this goal is not universal: Refactorings introducing parallelization, for instance, do not target at better changeability, but at improved utilization of the underlying hardware (see, e.g., [10]; but note that this could be considered an optimization rather than a refactoring). Also, it is conceivable that refactoring is performed for obfuscation, i.e., a design that makes (informed) code changes largely impossible (which presents an improved design if unchangeability is the goal).

2.2 The Elusiveness of Behaviour Preservation

An answer to the second above question is often given as "the program still compiles and passes the same tests the program passed before the refactoring" (see, e.g., [12]). While pragmatic, this answer merely suggests a post-hoc check of whether some concrete changes actually represent a refactoring; for refactoring tool builders, it translates to "for all possible applications to all possible programs with all possible test suites, the resulting program must still compile and pass the tests". Surely, this cannot be proven experimentally[1], but would require some abstract, formal argument, which is however hard to give. In practice, therefore, tool builders rely on testing their tools, and on users submitting bug reports.

Even with testing in place, the notion of behaviour that is to be preserved by refactoring is not unchallenged. For instance, the refactoring Replace Conditional with Polymorphism [12] may adversely affect program performance (by replacing explicit branching with dynamic dispatch), and this deteriorated performance may mean an intolerable change of behaviour in certain contexts. While such a deterioration may be detected by test cases (leading to a subsequent rejection of the refactoring), it cannot lead to a general abandonment of Replace Conditional with Polymorphism as a refactoring, as other

[1] I have gathered some first-hand experience with this, which drove me to lamenting "whenever we believed that we had made correctness of the refactoring plausible, testing it on a new project revealed a new problem we had not previously thought of" [24].

users may not conceive of slower performance as a behavioural change. On the other hand, a change of performance may be the very purpose of a refactoring: for instance, when refactoring for parallelization, a better user experience, and hence externally perceivable behaviour, may be the very goal. While one could argue again that such a change represents an optimization, this does not seem enough to expel corresponding work from the refactoring realm.

2.3 Ignoring the Unresolved Correctness Problem

Although correctness of refactoring tools has been a concern from the very beginning of the discipline (see, e.g., [18,29]), it seems that the builders of contemporary refactoring tools have surrendered to the complexity of the problems (see, e.g., [15,39,40]; Sect. 4 will give a concrete taste of the complexity of the problems one may encounter). While this has sparked off some research on how the correctness problems can be tackled (see, e.g., [3,5,13,47]; also, Sect. 6 is devoted to this entirely), I also observe that the refactoring community has some sympathy for downplaying the correctness problems (see, e.g., [7]), focusing on other topics instead. Particularly popular seem empirical investigations exploring the use of refactoring tools (mostly suggesting that the correctness problem is not one; see, e.g., [28,50]); other work focuses on increasing the utility of existing (even though buggy) refactoring tools, for instance by automatically discovering manual refactoring activities and completing them with tool support [11,14], or by automatically synthesizing larger refactorings from smaller ones [33]. Since the defectiveness of the underlying refactoring tools is not ironed out by automatically applying or combining them, the tools assembled from them are also defective. This however is almost consistently ignored.

2.4 The Easy Way Out: Liberation from Academic Chains

The more popular refactoring is becoming, the more its definition is being challenged (with arguments partly given above). Specifically, practitioners more and more suggest that refactoring amounts to automated program change, with behaviour preservation and design improvement, if at all desired, being left to the responsibility of the user. This culminates in the view that the laxer the preconditions of a refactoring tool, the higher its utility, even if this means that the tool introduces errors that then need to be fixed manually. Given this mindset, it may not be so surprising that sentences like "Even though our approach is neither sound, nor complete, it is still useful." (cited from a refactoring paper presented at a highly respected conference) make it into the academic literature.

I do not condemn this departure from the refactoring ideal — whichever tool works best for a programmer is good (although I maintain that the question, what works best for the programmer?, cannot be decided by the programmer alone, but must also be judged by the quality of the result; see [7] for a recent discussion). However, I grant myself the freedom of pursuing a more scholarly perspective here (which includes the liberty to choose my challenges independently from the purported programming practice), and to uphold the original definition of refactoring. I do however concede that whether or not design is actually improved is not a part of the definition of a refactoring (more precisely: a refactoring pattern), only of its application (an instance of the pattern).

3 The Generic Nature of Refactoring Tools

Following the school of Opdyke and Johnson [29,30], implementing a refactoring tool requires

1. implementing a check of the *preconditions* of the refactoring and
2. implementing a sequence of changes, also called the *mechanics* [12] of the refactoring.

Postconditions are usually considered dispensable by this school, unless refactorings are chained, in which case postconditions are to provide guarantees that the next refactoring's preconditions are met by the outcome of the present one [35]. Of course, this view ignores that every refactoring has a purpose, which is naturally reflected in its postcondition. A more pragmatic argument for dismissing the need for postconditions is that they follow from the preconditions and the mechanics of the refactorings and hence are redundant; this of course ignores that the mechanics could be flawed (see above), or the preconditions too weak. Both are however not uncommon for today's refactoring tools.

For this briefing, I will adopt a more fundamental viewpoint and regard refactoring tools as *metaprograms*, specifically as programs that implement source-to-source program transformations [18,21,27]. Metaprograms are programs and hence are specified using preconditions *and* postconditions. While this may seem overly academic, the reader will learn below that the pre- and postconditions of refactoring tools are, to a large extent, generic so that identifying and expressing them for a specific tool should not present too much of an effort. At the same time, the reader will (hopefully) join me in appreciating the ready availability of pre- and postconditions for refactoring tools as a (rare) occasion of being able to derive an implementation directly from its specification.

3.1 Generic Pre- and Postconditions

A *generic precondition* of all refactoring tools is that input programs must be well-formed.[2] *Generic postconditions* are that

[2] Practitioners may find this precondition too strong. Indeed, it seems that it could be relaxed to requiring well-formedness only for the parts of the program that are in some way connected to the intended refactoring. However, it seems difficult, if not impossible, to decide if a malformed part of a program is connected to a refactoring. For example, what if the refactoring makes the formerly malformed part well-formed, for instance by renaming a declared element so that a formerly unbound reference now binds to this element?

- the refactored program is still well-formed, that
- it behaves the same, and that
- the program either exhibits at least the changes immediately associated with the refactoring (the *refactoring intent*), or remains unchanged.

3.2 Specific Pre- and Postconditions

Beyond the generic preconditions, preconditions specific to a concrete refactoring tool are to protect the tool from input (programs to be refactored and user-supplied parameters of the refactoring) that it cannot handle, either because the refactoring is undefined for them, or because of unresolved technical challenges (including the impossibility to guarantee behaviour preservation, if this escapes the current capabilities of static program analyses). If preconditions are violated, the refactoring tool should leave the program unchanged, and report the violation to the user, who can then try to prepare the program manually (by performing required changes, arguably refactorings) for successful tool application.

The postconditions specific to a concrete refactoring assert that the changes associated with the refactoring are actually seen in the refactored program. Basically, they have the form "the refactored program shall exhibit property X", where X expresses a change (such as a change in the type hierarchy etc.). Some (if not most) refactorings require additional changes to be made, changes that complement the refactoring intent to restore the program's well-formedness or its behaviour (see Sects. 4.3 and 6 for examples). These changes are typically not part of the specific postconditions; indeed, computing them is the hard part of refactoring tool implementation. However, as we will see, the required additional changes can sometimes be derived from the (generic and specific) postconditions.

3.3 Generic Refactoring Invariants

That a program must be well-formed before and after a refactoring, and that the behaviour must remain the same (conditions included in the generic pre- and postconditions of a refactoring tool) can be viewed as *generic invariants*. If behaviour is specified in terms of a test suite, preservation of these invariants is easily checked: by running the compiler and test suite before and after the refactoring.

If however a test suite sufficient for checking behaviour preservation is unavailable, or if behaviour preservation is to be specified independently of any given program, checking invariant preservation requires the following generic procedure:

1. check well-formedness
2. extract the behaviour-critical invariants
3. perform refactoring
4. check well-formedness
5. check extracted invariants

Here, it is understood that the behaviour-critical invariants extracted from the program before the refactoring (Step 2) hold at the time of extraction. A failure of any the refactoring invariants after a refactoring can be interpreted as the violation of specific preconditions, which are however not explicitly specified (see Sect. 6.1 for a brief discussion of the pros and cons of this). This observation (limited to the behaviour-critical invariants) was already made by Max Schäfer [36], and also by Jeffrey Overbye [31].

4 Why Building Refactoring Tools Is Hard: A Case Study

In his refactoring book [12], Fowler provides the following synopsis for the Replace Inheritance with Delegation refactoring:

A subclass uses only part of a superclasses interface
or does not want to inherit data.

Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.

The prototypical example of a class one might want to rid of its superclass using Replace Inheritance with Delegation is that of `Stack` extending `Vector`:

```
class Stack                          class Stack {
    extends Vector {                   Vector elems = new Vector();
  void push(Object o) {                void push(Object o) {
    add(o);                 ⇒            elems.add(o);
  }                                    }
  ...                                   ...
}                                    }
```

Fowler prescribes the following mechanics for this refactoring [12]:

1. Create a field in the subclass that refers to an instance of the superclass. Initialize it to `this`.
2. Change each method defined in the subclass to use the delegate field. Compile and test after changing each method.
3. Remove the subclass declaration and replace the delegate assignment with an assignment to a new object.
4. For each superclass method used by a client, add a simple delegating method.
5. Compile and test.

What the prescription does not say is what to do if the program does not compile during step 2 or 5, or if any of the test cases fail. A yielding reaction would be to undo all changes and give up on the refactoring (assuming that the refactoring was not intended for the given case or, more formally, that the program did not meet the preconditions of the intended refactoring); a persisting reaction would be to find out the source of the problems, and work around them (suggesting that the mechanics failed to cover the special conditions — in the community often marginalized as "corner cases" — that led to the failure). Either way, a user of this refactoring, and even more so a tool builder, is left alone with learning its particulars.

(a)

```
package a;
class A {
  protected A(...) {...}
  protected void m() {...}
}
package b;
class B extends A {
  B(...) { super(...); }
  void n() { m(); }
}
```

(b)

```
class A implements I {}
class B extends A {}
I i = new B();
```

(c)

```
class B extends Throwable {...}
void m() throws B {...}
```

(d)

```
class A { int i; }
class B extends A {}
B b = new B(); b.i = 0;
```

(e)

```
class Thread() {
  void run() {}
  void start() {run();}
}
class B extends Thread {
  void run() {...}
}
B b = new B();
b.start();
```

(f)

```
class A {
  void m() {
    notify();
  }
}
class B extends A {
  synchronized void n() {
    wait();
  }
}
```

Fig. 1. Programs for which Fowler's Replace Inheritance with Delegation [12] does not work out-of-the box.

4.1 The Precondition Surprise

Although Fowler's treatise of Replace Inheritance with Delegation does not mention preconditions, it gives a few clues as to the refactoring's applicability. A trivial precondition that can be derived from Step 3 of its mechanics given above is that the inheriting class (the subclass to which the refactoring is to be applied) has a superclass other than `Object`, since in Java at least, every class implicitly inherits from `Object`. Other, slightly less obvious preconditions are that the superclass must be instantiable (i.e., not abstract) and that the superclass constructors (called from the subclass either implicitly or using `super`) must be accessible from the subclass even when it is no longer a subclass (Fig. 1(a)); also suggested by Step 3. Steps 2 and 3 together suggest a similar requirement: replacing `this` with the delegate field means that the members accessed via this field must still be accessible after the inheritance has been removed. This is not the case, for instance, if superclass and subclass reside in different packages and superclass members are declared `protected` (Fig. 1(a)). Not mentioned in Fowler's tractate is that the program must not require assignment compatibility between the subclass and the superclass; specifically, no instance of the (former) subclass must occur where an instance of the (former) superclass is expected.[3] While one could argue that removing subclassing, and with it subtyping, is the very purpose of the refactoring (which therefore cannot be applied when subtyping is required), the situation is actually less clear-cut for instance when the superclass implements interfaces (including marker interfaces) the subclass no longer implements (Fig. 1(b)). Also, subclassing cannot be removed without breaking the program if the subclass inherits from a class with special semantics, on which the program relies (Fig. 1(c)). Last but not least, the refactoring will fail if clients access fields (rather than methods as in Step 4) of the (former) superclass through the (former) subclass (Fig. 1(d)). This is so since in Java at least, field access cannot be delegated. For this, it would be necessary to introduce accessor methods for the fields first (the ENCAPSULATE FIELD refactoring), and to let the clients use them.

While violations of the above preconditions of the REPLACE INHERITANCE WITH DELEGATION refactoring are unveiled by the error reports of the compiler (if only after the fact; see Sect. 3.3 for how this relates to invariant preservation), the really nasty preconditions are discovered only by testing behaviour preservation. One is that the refactoring does not replace inheritance with *delegation* (as it claims), but with *forwarding*. For true delegation, it would be necessary that the use of `this` in a delegated method call refers back to `this` in the delegating method (the delegator), whereas with forwarding, `this` refers to the object being delegated to. This is a problem when the (fake) delegation calls a method on `this` that used to be overridden in the (former) subclass, as exemplified in Fig. 1(e) (using a home-brew implementation of `Thread`). This overriding, and with it the dynamic dispatching to the subclass, are however gone, usually resulting in changed behaviour.

Another hard to discover precondition arises in the context of multithreading. In Java, synchronized method calls are guarded by a monitor associated with the receiver of the method call. After application of the REPLACE INHERITANCE WITH DELEGATION refactoring, however, invocations of methods formerly inherited are now "delegated" (actually: forwarded) to a different object, which has a different monitor. Synchronization may therefore fail, as in the case of Fig. 1(f).

One might argue that both of the above are corner cases that will rarely occur in practice, so that their neglect can be tolerated. However, it is somewhat assuming to claim that certain constructions are or will be rarely used[4]; at the same time, no one can predict the harm their neglect may cause.

[3] This includes `instanceof` tests, which will become ill-typed.

[4] In a study conducted by the author, dynamic dispatching affected 41%, and synchronization affected 3.5% of all attempted applications of REPLACE INHERITANCE WITH DELEGATION [24].

4.2 The Mechanics Adventure

While Fowler leaves the preconditions of Replace Inheritance with Delegation mostly for the discovery by others, he leaves only little doubt as to its mechanics, i.e., what needs to be done to perform the refactoring. The only pitfall is hidden in Step 4, which requires that all method calls from clients of the (formerly) inheriting class are identified (so that the required delegating methods can be introduced). Precise identification of the calls is crucial to the success of the refactoring: adding delegating methods that are never called by clients counteracts the very purpose of the refactoring (the deflation of the class interface), while missing out required delegating methods will lead to compile errors. Unfortunately, an analysis of the class members required by the class's clients is only seemingly simple; factually, it requires the type analysis underlying the Extract Interface refactoring [48], which is not trivial. Without such an analysis at hand, performing the refactoring will be a trial and error adventure (delegating methods are added until all type errors are resolved).

Generally, to keep the mechanics of a refactoring simple, the definition of strong preconditions seems a good idea. However, as suggested by the preconditions of Replace Inheritance with Delegation derived from the examples of Fig. 1, it may render the refactoring unusable in too many cases, making the user perform preparatory refactorings required for doing the refactoring anyhow. Figuring out precisely which preparatory refactorings are required is an adventure in its own right, in particular when considering that each preparatory refactoring may suffer from the same problem recursively.

4.3 The Tool User's Dream: Relaxed Preconditions

Given the above, rather long list of preconditions for Replace Inheritance with Delegation, it is indeed questionable whether a refactoring tool requiring them all will be useful in practice, or will deny its service on too many occasions[5]. Also, given that at least some of the precondition violations seem easy to work around (for instance, access modifiers can be adjusted, accessor methods can be introduced), it is foreseeable that users will ask for a refactoring tool that can fix these issues by itself, rather than suggest corresponding manual changes (see [50] for some evidence of this). However, as other work has shown, changing access modifiers consistently is not as easy as it may seem, and in Java can even lead to changes of behaviour (by changing binding) [45]. Also, setter invocations cannot generally replace for field assignments, so that both changes are rather complex refactorings by themselves. As for the remaining preconditions: Even if there are ways to do away with them, as we will see below this takes far more than can be straightforwardly handled in an imperative (as opposed to declarative) fashion, that is, through a sequence of steps ("mechanics"). *If building correct refactoring tools is hard, relaxing their preconditions is harder.*

[5] In the same study [24], they prevented 84% of all naive refactoring applications in four subject programs.

4.4 The Tool Builder's Nightmare: Evolving Languages

While creating correct refactoring tools for programming languages as complex as Java or C$^{\#}$ is already hard, evolving them to keep up with the further development of these languages is a nightmare. This is not only so because, after the compiler has been updated, the refactoring tools need to follow to accommodate the same set of new language features, but also because it raises expectations regarding tool support for migrating now legacy programs to the new language version. For instance, the introduction of generics to Java not only broke literally all type-related refactorings, it also led to the formulation of new refactorings introducing generics to legacy code [48]. Not surprisingly, developing these tools occupied some of the brightest minds in our field, and still left us with tools that are, strictly speaking, neither sound nor complete.[6]

5 Current Refactoring Practice and Research Challenge

Given the hardness of the refactoring problems exposed by the above case study, and given that most contemporary refactoring tools have not found good means of dealing with these problems, refactoring practice today often follows the pattern

1. Perform the refactoring as specified (using a tool, if available).
2. If the refactored program exhibits compile errors or changed behaviour,
   (a) either undo the refactoring or
   (b) perform corrective changes compensating for shortcomings in the mechanics of the refactoring.

In case 2(a), violation of the generic postconditions (Sect. 3.1) and assuming that the performed mechanics are correct suggest that the preconditions of the refactoring have not been met. In this case, the user can try to prepare the program manually for the refactoring. If the preconditions are not explicitly specified (as in Fowler's above specification of Replace Inheritance with Delegation), the reported compile errors or failed test cases may provide some hints for the necessary preparation; in any case, they are the sole instance deciding that the program is ready for the refactoring as implemented by its mechanics.

In case 2(b), the task of complementing the mechanically performed refactoring with the required additional changes is guided by the compiler and test suite, which serve as oracles of manual task completion. If the refactoring user is happy with this situation, one may indeed suggest that preconditions are relaxed as much as possible — as long as the resulting program can be easily fixed, it does not matter whether violations of the generic postconditions are due to violated preconditions or to shortcomings in the specification of the mechanics. From a tool builder's perspective, this is a pleasant prospect, since it makes the implementation of refactoring tools a much simpler task.

[6] I freely admit that I spent one summer trying to understand what it takes to cover Java's generics in every detail, and gave up highly frustrated.

No matter whether the tool user opts for (a) or (b): In either case, making a failing refactoring work relies on the knowledge encoded in the compiler and test suite. From an academic perspective, it is somewhat saddening that this knowledge is not exploited by the refactoring tools, for computing *all* changes required by a refactoring upfront. In fact, I find the prospect of being able to do so, so intriguing that it leads me to posing the following

Research challenge for the future of refactoring tools:

to evolve the decision procedure

"*does this change constitute a refactoring?*"

(as implemented, e.g., by the compiler and test suite) into a search procedure

"*which additional changes are required to make this change a refactoring?*"

Ideally, we can use the same implementation used for solving the decision problem for solving the search problem also. *This would not only greatly reduce the effort required to create new refactoring tools, it would also allow us to keep compiler and refactoring tools so closely coupled that changing (fixing or evolving) one suffices for both.*

In the next section, I will shed some light on systematic approaches to implementing refactoring tools known from the literature, with a special focus on how they exploit (program-independent) knowledge also encoded in the compiler. I will not address in the following how the knowledge captured in test cases (which is program-dependent) can be exploited; however, I do point out here that the Smalltalk Refactoring Browser can actually make some use of it [35].

6 Principled Approaches to Implementing Refactoring Tools

Considering the nature of refactoring tools as delineated in Sect. 3, it seems clear that any principled implementation of a refactoring tool must observe the generic pre- and postconditions of refactoring or, equivalently, preservation of its invariants. The research challenge phrased above additionally suggests that a refactoring tool should rely on the language expertise implemented in the compiler. The approaches presented in the following do both.

6.1 Dependency Preservation

In light of the problems with framing behaviour preservation (Sect. 2.2), it seems advisable to replace it with a notion that is better tractable. *Dependency preservation* as put forward by Schäfer [36] is such a notion.

Dependency preservation abstracts from behaviour preservation in that it promises to maintain all behaviour-critical relationships between program elements. For instance, it is perfectly plausible to require that, except for deliberate changes, after a refactoring

- all names in a program should bind to the same declarations,
- all method calls should be synchronized on the same monitors, and
- all methods should override the same methods

as before the refactoring. Indeed, any accidental change of binding, synchronization, or overriding (collectively referred to as a change of dependency by Schäfer) may lead to a change of behaviour and hence provides a reason for the rejection of the refactoring that causes it. Schäfer demonstrated the effectiveness of dependency preservation by implementing a large number of refactoring tools with correctness scores surpassing that of the Eclipse JDT's built-in refactoring tools, as measured by their own test suites [36].

Dependency preservation also solves some of the problems of the Replace Inheritance with Delegation refactoring presented in Sect. 4. For instance, in the code of Fig. 1(a), the method invocation `m()` in the body of `B.n()` is bound to the definition of `m()` in class `A` before the refactoring; since it cannot be bound after the refactoring (because `m()` has become inaccessible for `n()`), a dependency of the name on its declaration could not be preserved. Similarly, the binding of the field access `b.i` to `A.i` cannot be preserved, since after the refactoring, class `B` no longer offers such a field (Fig. 1d).

Going beyond name binding, the loss of synchronization arising from naively refactoring the code of Fig. 1(f) can be detected by the fact that `wait()` and `notify()` are now invoked on different objects (making synchronization depend on different monitors), whereas they were invoked on the same before. While such a change of dependency is hard to detect statically in the general case, in the case of Replace Inheritance with Delegation it is fairly simple, since `this` in the delegating class and `this` in the class being delegated to can never point to the same object. For the preservation of dynamic binding (Fig. 1e), the situation seems more complex, as it would require a static analysis of dynamic dispatching behaviour even for Replace Inheritance with Delegation; however, requiring that all overriding dependencies are preserved (independently of the fact whether or where overriding actually leads to dynamic binding) is sufficient for guaranteeing binding invariance (even though it may be too strong a condition in certain cases).

Thus, we have that dependency preservation can cover a broad spectrum of conditions that are otherwise difficult to express. However, as these examples also suggest, much of the art of implementing correct refactoring tools using dependency preservation relies on identifying and being able to extract the dependencies that guarantee behaviour preservation for arbitrary programs. Particularly for refactorings that change the control or data flow of a program, this may prove beyond reach.

6.1.1 Technical Enforcement

It is fairly obvious that dependency preservation is a special case of (generic) invariant preservation as delineated in Sect. 3.3. Technically, it is enforced by recording all dependencies before the refactoring (replacing for Step 2 in the procedure of Sect. 3.3), and by re-computing and comparing them after the refactoring (Step 5). If any dependency has changed as a result of the refactoring, it is rejected and all associated changes are undone. Since computing the dependencies can usually be trusted to the compiler, the refactoring tool implementation is spared from repeating some of the language specification in its own code. As noted by Schäfer, this is a huge advancement over traditional precondition checking, which often requires laborious reverse engineering of the language specification. On the downside, however, the fact that violated preconditions are now implicit makes it harder for the refactoring tool user to figure out what exactly led to a refusal.

For large programs, retrieving and storing all dependencies can be rather expensive. Therefore, in all practical applications of dependency preservation, only those dependencies that can be affected by a refactoring will actually be recorded. Unfortunately, deciding which these are is a problem in its own right; dependencies may stretch across several modules, and are not always obvious. Making mistakes here will make refactorings relying on dependency preservation unreliable.

6.1.2 Actively Preserving Dependencies

While being able to replace explicit precondition checking with attempting dependency preservation is certainly an advancement for the conscientious refactoring tool builder, it still leaves the tool user with the problem of "too strong preconditions", i.e., the rejection of a refactoring in cases in which some moderate additional changes would have made it possible. However, as has also been shown by Schäfer [36], in certain cases the compiler can be exploited to compute these additional changes also.

The original example of how this can work was given by Schäfer in his implementation of the RENAME refactoring [36]. The idea here is to let the refactoring tool compute the inverse of the binding function implemented by the compiler: rather than computing for a given name in a given location the declaration to which it binds (the binding function), a name is computed from a given declaration (the one originally bound to) and a given location (where the name is to be used; the same location as that of the original name) such that name, if it exists, is guaranteed to bind to the declaration. This not only propagates a change of the name of a declared entity to all references to (or accesses of) it, it also introduces name qualification where needed.

A simple example showcasing the power of active dependency preservation is given by the following code snippet (taken from [36]):

```
class A {
  int x;
  A(int newX) {
    x = newX;
  }
}
```

Supposing that the formal parameter `newX` is naively renamed to `x`, the declaration of the field of the same name, `x`, will be shadowed inside the constructor, so that the left-hand side of the assignment (now reading `x = x`) will also bind to the formal parameter, likely changing the behaviour of the program. However, computing the fully qualified name of field `x` referenced from the location of the assignment yields `this.x`; replacing the left-hand side of the assignment accordingly keeps the program intact. As Schäfer showed, this naming function can be constructed systematically with little effort and high accuracy, by reversing the name lookup function implemented by the compiler (see Sect. 6.3.2 for a constraint-based account). Whenever this lookup function needs to be adjusted (for instance, because the language evolves), its reverse can be adjusted in parallel, keeping all refactoring tools relying on it up-to-date (cf. Sect. 4.4).

It would seem that reversing name lookup can be extended to repair the broken binding introduced by applying Replace Inheritance with Delegation to the example of Fig. 1(a) also. Indeed, the compiler knows that for a non-inherited method to be accessed across package boundaries, the method must be declared `public`. It would therefore seem feasible to introduce a second function which computes, for a given location and declared entity to be accessed from that location, the set of access modifiers granting this access. *However, making a corresponding adjustment affects a declaration, rather than a reference (as above insertion of a qualified name did); it will therefore affect all other references to this declaration, too, and may interfere with other existing declarations.* While this may not seem problematic at first glance, as has been shown elsewhere [45], changing access modifiers in an ad-hoc fashion may not only lead to malformedness (for instance, in presence of overriding), but can also break binding dependencies. This will be picked up again in Sect. 6.3.

And yet, active dependency preservation is not limited to adjusting names at reference sites. For instance, as Schäfer demonstrated, synchronization dependencies can also be actively preserved, by making sure that method invocations remain synchronized on the same monitors as before a refactoring. Transferred to the synchronization problem of Replace Inheritance with Delegation (as exemplified by applying it to the code of Fig. 1e), dependency preservation requires that the delegating object is passed (as a parameter) to the method being delegated to. Following this advice, the naively refactored code below on the left (which exhibits the lost synchronization) is changed to that on the right (which preserves the original dependency):

```
class A {                          class A {
  void m() {                         void m(Object o) {
    notify();                          o.notify();
  }                                  }
}                                  }
class B {                     ⇒    class B {
  A a = new A();                     A a = new A();
  void m() { a.m(); }                void m() { a.m(this); }
  synchronized void n() {            synchronized void n() {
    wait();                            wait();
  }                                  }
}                                  }
```

6.2 Language Extensions and Restrictions

While dependency preservation is a powerful concept, Schäfer also showed that it gets even more powerful when combined with language extensions and restrictions [36]. For instance, he observes that the `synchronized` method modifier in Java merely provides syntactic sugar for the more general `synchronized` block: a synchronized instance method is equivalent to a non-synchronized method whose body is wrapped by a block explicitly synchronizing on `this`. Java without synchronized methods, but with synchronized blocks, is thus a restricted language to which any Java program can be straightforwardly transformed. This restricted language is helpful, for instance, when performing the Move Method refactoring, as exemplified by moving method `m()` in the following code from class `A` to class `B`:

```
class A {
  synchronized void n() {}
  synchronized void m() {
    n();
  }
}

class B {}
```

$\Downarrow$

```
class A {
  void n() {
    synchronized(this) {}
  }
  void m() {
    synchronized(this) {
      this.n();
    }
  }
}

class B {}
```

$\Rightarrow$

```
class A {
  void n() {
    synchronized(this) {}
  }
}

class B {
  void m(A a) {
    synchronized(a) {a.n();}
  }
}
```

$\Uparrow$

```
class A {
  synchronized void n() {}
}

class B {
  void m(A a) {
    synchronized(a) {a.n();}
  }
}
```

Here, the first step (indicated by the down arrow) is to convert the classes to Java without synchronized methods (and without assuming `this` as the default receiver). In the next step (right arrow), the method `m()` is moved as usual, making sure that `this` is converted to a formal parameter (Schäfer actually uses a *language extension* for this [37]). The last step (up arrow) converts the classes back to Java with synchronized methods; note that, since the body of `m(A)` is synchronized on a different object than `this`, conversion is possible only for `n()`. Were class `B` a subclass of `A`, `this` and `a` would always point to the same object, so that both methods could use the `synchronized` keyword.

As it turns out, name binding preservation can also be framed in terms of a restricted language [38]. For this, all names used in references and declarations of a program are replaced with unique names, or labels. Because each declaration now uses a different label, the binding rules of the language become extremely simple: Each reference binds to the sole declaration carrying the same label.[7] In particular, no hiding, shadowing, obscuring, or overloading may get in the way of a refactoring. After a refactoring, the declared entities can adopt their original names, and the inverted lookup function can be used to compute the names of the references.

However, as already noted in Sect. 6.1, there are refactoring problems that exceed the capabilities of dependency preservation and language extensions or restrictions. For instance, in the course of the REPLACE INHERITANCE WITH DELEGATION refactoring access modifiers may need to be adapted at the declaration site to keep a program well-formed (see Fig. 1a). In addition, if qualifiers (as part of the name computed by inverting the lookup function) must be introduced at the reference site, the names used for qualification may refer to inaccessible entities, requiring additional access modifier adjustments to avoid malformedness [38]. However, adjusting access modifiers can itself lead to a change of name binding, not only making binding preservation a recursive problem, but also intermingling well-formedness preservation with dependency preservation. The same applies to refactorings that may make a program ill-typed: For instance, when the subtype relationship is removed (again as with REPLACE INHERITANCE WITH DELEGATION), assignments (as in Fig. 1c) or member accesses (as in Fig. 1d) may become ill-typed. For dealing with these kinds of problems, another principled approach to implementing refactoring tools seems better suited: *constraint-based refactoring.*

6.3 Constraint-Based Refactoring

Constraint-based refactoring was pioneered by Frank Tip et al., who adopted Jens Palsberg and Michael Schwartzbach's constraint-based capture of object-oriented type systems [32] for the implementation of type generalization refactorings such as GENERALIZE DECLARED TYPE or USE SUPERTYPE WHERE POSSIBLE [48,49]. However, rather than following the historic trail of this seminal work, I present constraint-based refactoring as a way of *preserving refactoring invariants* in the spirit of Sect. 3.3 here.

6.3.1 Preserving Dependencies with Constraints

To show how refactoring invariants can be expressed in terms of constraints, I will start with preserving dependencies, as this allows me to draw some parallels to Schäfer's work. Below, I will address how well-formedness can be preserved, using the examples of accessibilities and types.

[7] Note the relationship to projectional editing, which uses references, or pointers, rather than names.

For the first part, we return to the name capture problem of Sect. 6.1:

```
class A {
  int x;
  A(int newX) {
    x = newX;
  }
}
```

A binding invariant of this snippet is expressed by the two simple constraints

$$ref_x.name = decl_x.name \tag{1}$$

$$ref_{newX}.name = decl_{newX}.name \tag{2}$$

Here, $decl_x$ and $decl_{newX}$ represent the *declared entities* of the program (currently named “x” and “newX”, resp.), and $decl_x.name$ and $decl_{newX}.name$ represent *constraint variables* holding the names of these entities. Analogously, ref_x and ref_{newX} represent *references* to the declared entities, and $ref_x.name$ and $ref_{newX}.name$ their names.

As invariants, (1) and (2) enforce that the names of the references must always equal those of the declared entities they bind to, where the binding has been determined prior to the constraint generation (e.g., by querying the compiler; but see Sect. 6.3.2 for how binding can be computed using constraints). A RENAME refactoring is hence expressed as changing the value of one of the constraint variables; the violation of constraints that this immediately causes flags the loss of a dependency. For instance, if $decl_{newX}.name$ is changed to “x”, (2) is immediately violated, since $ref_{newX}.name$ still holds the value “newX”. However, the lost binding can easily be restored, simply by letting a constraint solver assign the other constraint variable ($ref_{newX}.name$ in the above example) the same name (representing a corresponding name change in the program), hence curing the violation. Thus, using a single set of constraints, we cannot only check dependency preservation, but also compute the corrective changes required to preserve dependencies actively (as in Sect. 6.1).

In constraint-based refactoring the name capture caused by renaming the formal parameter `newX` to “x” is avoided by adding a third constraint

$$decl_{newX}.name \neq decl_x.name \tag{3}$$

The generation of this constraint is justified by the fact that `newX` is declared in a scope in which it would shadow the declaration of `x`, if their names were the same. While not necessary for the program as is, it helps preserve the name binding under renaming either `newX` or `x`, by requiring that their names are always different. In fact, a constraint-based implementation of the RENAME refactoring would not need to reject the renaming of the formal parameter `newX` to “x”; rather, it would rename the field `x` to a different name and, observing (1), the reference to `x` with it. While this measure of actively achieving dependency preservation differs from Schäfer’s (which worked by introducing qualifiers; see Sect. 6.1), it is equally successful. In fact, it works even in cases in which name qualification is impossible.

6.3.2 Aside: Implicit Specification of Name Lookup and Its Reversal Using Constraints

It is instructive to observe that, despite their technological differences, Schäfer's computation of the inverse of the binding function presented in Sect. 6.1 and the constraint-based capture of active dependency preservation presented above are closely related. This is revealed by the following slight modification of (1) and (2):

$$ref_x.name = ref_x.binding.name \tag{4}$$

$$ref_{newX}.name = ref_{newX}.binding.name \tag{5}$$

Here, the variable declarations to which the references ref_x and ref_{newX} bind have been replaced by the constraint variables $ref_x.binding$ and $ref_{newX}.binding$, resp. Assuming that all names in a program are fixated (so that the *name* variables do not change their values), a constraint solver will determine the bindings of ref_x and ref_{newX} by finding values for $ref_x.binding$ and $ref_{newX}.binding$ such that the constraints are satisfied. This corresponds to *computing the lookup function*.

Using the same constraints (4) and (5), that a binding must not change under refactoring (the binding invariance) is expressed by fixating the values of the constraint variables $ref_x.binding$ and $ref_{newX}.binding$ (the values just computed by the solver). Renaming declarations (by assigning $decl_x.name$ or $decl_{newX}.name$ new values) and making the values of the $ref_x.name$ and $ref_{newX}.name$ variable, then corresponds to *inverting the lookup function* as proposed by Schäfer, in that it propagates a changed name of a declared entity to its references. However, unlike for Schäfer's procedural approach, which needs to provide related, but still independent implementations for name lookup and name computation, constraint-based refactoring exploits that constraints are generally undirected ("n-way"), and makes do with a single specification. In fact, a single constraint-based specification can be used to

1. extract dependencies before the refactoring (corresponding to Step 2 in the generic procedure of Sect. 3.3),
2. check dependencies after the refactoring (Step 5), and
3. compute required corrective changes (part of Step 3).

As we will see next, constraints can also be used to

4. check well-formedness before and after a refactoring (Steps 1 and 4 in the generic procedure of Sect. 3.3), and to
5. compute corrective changes required to preserve well-formedness (again part of Step 3).

Note that Item 4 is also done by the compiler, which needs to check the same constraints. Elements of Item 1 must also be implemented by the compiler (for instance when resolving names or when creating tables for dynamic method dispatch), even though most compilers will not use constraints and constraint solving for this purpose.[8] Items 3 and 5 are actually part of the mechanics of a refactoring; I will return to this at the end of this section.

6.3.3 Accessibility Constraints

One of the refactoring problems classified in Sect. 6.1 as not being amenable to dependency preservation is that of adapting access modifiers. To get an impression of the problem, we adapt the code snippet of Fig. 1(a), adding `C` as a second subclass of `A` defining an overriding method `m()`:

```
package a;
class A {
  protected void m() {...}
}
package b;
class B extends A {
  void n() { m(); }
}
class C extends A {
  @override protected void m() {...}
}
```

Recall that the problem of applying Replace Inheritance with Delegation on class `B` was that it makes `A.m()` inaccessible from the body of class `B`. This problem appears to be readily fixed by increasing the accessibility of `A.m()` to `public`; however, this makes the program malformed, since Java requires that overriding methods must be declared at least as accessible as the methods they override. In this particular case, this means that accessibility of `C.m()` needs to be adjusted to `public` as well; in other cases, other rules may apply.

A constraint-based solution to this problem is to express the well-formedness rules related to access modifiers in the same style as the name binding rules above. For instance, accessibility of `A.m()` from `B` can be expressed by the constraint

$$decl_{A.m()}.accessibility \geq (B <: A\, ?\, protected : public) \tag{6}$$

where $<:$ denotes the subtype relation and $?\ :$ is the ternary conditional operator (note that access modifiers are totally ordered in Java: $private < package < protected < public$). The constraint says that *protected* accessibility for `A.m()` suffices as long as `B` is a subclass of `A`; otherwise, it must be *public*.[9] Note that this constraint is only justified if `B` (or any other class from a different package) requires access to `A.m()`; in the above example, it is required by the access through `B.n()`.

[8] In fact, for efficiency reasons, it may not be advisable to use standard constraint solving for this purpose. However, efficient one-way computations may be synthesized from n-way constraints [22]. A conventionally implemented lookup function is a good use case for this. See also at the end of Sect. 9 in Part II of this briefing, where this issue is picked up again.

[9] This greatly oversimplifies matters — see [38] for a more thorough account of accessibility in Java.

The fact that accessibility of `C.m()` must be greater or equal than that of `A.m()` is expressed as the *conditional constraint*

$$C <: A \rightarrow (decl_{C.m()}.accessibility \geq decl_{A.m()}.accessibility) \tag{7}$$

which says that if `C` is a subclass of `A`, accessibility of `C.m()` must be equal to or greater than accessibility of `A.m()`. Note that both constraints (6) and (7) use the subtype relation as a condition — this is important, since the Replace Inheritance with Delegation refactoring changes this relation, and access modifiers must adapt to the change.

Conditional constraints are commonplace in constraint-based refactoring (see, e.g., [2]); however, they also impact tractability and hence require special treatment. For instance, for refactorings that move program elements between scopes, constraint (3) of Sect. 6.3.1 would need to be conditioned on both declared entities residing in the same scope, so that the constraint is active when they do, and inactive otherwise. In general, it is not trivial to know which constraints will actually be needed in which form for a given refactoring, and generating all constraints, and each in its most general form, will be too expensive. Therefore, the constraint generation process must "foresee" all changes a refactoring may possibly make [43]. Note how this parallels the problem of extracting all and only the dependencies required for a specific refactoring in Schäfer's work (cf. Sect. 6.1.1).

6.3.4 Type Constraints

Some of the remaining problems of Replace Inheritance with Delegation discussed in Sect. 4 can be framed as typing problems, specifically as the loss of well-typedness. Similar to accessibility above, preserving well-typedness can be expressed as a constraint satisfaction problem.

To see how this works, we return to the example of Fig. 1, specifically the code snippet

```
class A implements I {}
class B extends A {}
I i = new B();
```

Recall that applying Replace Inheritance with Delegation to class `B` makes the assignment ill-typed, since `B` is no longer a subtype of `I`.

The corresponding typing invariant is expressed by the constraint

$$B <: decl_i.type$$

Clearly, this constraint, which is satisfied before the refactoring, is violated after it, since the current type of `i`, `I`, is no longer a supertype of `B`. However, a constraint solver can repair the broken constraint, either by changing the value of $delc_i.type$ to B (corresponding to a change of the declared type of `i` to `B`) or by changing the type hierarchy so that $B <: I$ (corresponding to letting class `B` implement interface `I`). However, in a program larger than the above, we must expect both changes to be subject to further constraints. For instance, class `B` must implement all methods declared in `I`. Conversely, if members are accessed on `b`, these members must be declared in interface `I`.

The latter constraint also plays a role in refactoring the code of Fig. 1(d), again repeated here for ease of access:

```
class A { int i; }
class B extends A {}
B b = new B(); b.i = 0;
```

Here, the access of `i` on receiver `b` requires that `i` is a field of `b`, expressed by the constraint

$$ref_b.type <: decl_i.host$$

Again, removing the subtype relationship between `B` and `A` violates this constraint. A constraint solver can compute a fix, however: by setting $decl_i.host$ to B, the constraint is satisfied again (and the declaration of field `i` is pushed down from class `A` to class `B`). In most real programs, however, this fix will be prevented by other constraints requiring that field `i` remains a member of class `A`.

A detailed presentation of the Java type constraints relevant for type-related refactorings is found in [48].

6.3.5 Generic Constraint-Based Refactoring Tool Implementation

It should be clear from the above that using constraints, we cannot only

– check well-formedness and dependency preservation of a program (and hence whether a refactoring was successful),

but also

– extract dependencies (as in the name binding example) to be preserved, and
– compute the corrective changes required to actively preserve well-formedness *and* dependencies.

What is missing from a completely constraint-based implementation of a refactoring tool is that the refactoring intent (see Sect. 3.1) is also expressed in terms of constraints. However, in as much as the changes constituting the refactoring intent can be expressed in terms of new values for constraint variables (as was the case for most examples presented in this section), this is easy: Simply add constraints forcing the new value (e.g., adding the constraint $decl_{newX}.name$ = "x" forces the renaming of `newX`). The constraint solver is then a generic refactoring engine capable of computing all changes required to realize a given refactoring intent.

7 Refactoring Résumé: Three Competing Camps

The previous sections suggest that three different perceptions of refactoring tools have emerged in the refactoring community:

– *Tool builders* maintain that to implement a refactoring correctly, it suffices
  1. to identify its preconditions and
  2. to specify the mechanics performing the changes that constitute the refactoring.

– *Tool users* suggest that for refactoring tools to be useful,
  1. an implementation of the mechanics and
  2. automated oracles checking the generic postconditions (compiler and test suite)

  are all that is required.
– *Tool researchers* hope that the necessary changes associated with a refactoring can be synthesized from
  1. the invariants of the refactoring and
  2. the specific changes to be seen in the program (the specific postconditions, or refactoring intent).

Reality catches up with:

– *tool builders* when the bug reports from users start coming in, and the struggle against the intricacies of the subject language leads to thoughts of resignation;
– *tool users* when they lose control over their code, because they are trapped in fixing bugs they did not introduce, in places they had not dreamt of; and
– *tool researchers*, when they apply their tools to real programs written in real programming languages and they learn how complex the semantics of these languages are.

Undoubtedly, building correct refactoring tools is hard. With the compiler specifying the semantics of a programming language, it seems that re-using as much of it as possible for the implementation of refactoring tools is *the* key to success.

Part II: Their Kin

Maybe the technical difficulties of producing correct refactoring tools and the expected benefit do not match well. Maybe the investment necessary to get refactoring right pays only if other programming tools profit from it, too. Maybe there is a common basis of a large variety of programming tools, of which refactoring tools are just one offspring.

Figure 2 depicts a bunch of such tools that all depend on a single software artefact, the specification of the static semantics of a programming language. In this bigger picture, it appears that refactoring merely plays a small, if not subordinate, role. However, we have seen that refactoring is also one of the harder problems in this bouquet, requiring some guarantees with respect to well-formedness *and* behaviour. For any specification sufficient for refactoring we may therefore expect that it is sufficient for the other problems as well.

8 Static Checking

Static checking is a central activity of compilers that comes straight after syntactic checking, or parsing. Depending on the language specification, it includes checks that all names are declared before they are used, that all expressions are

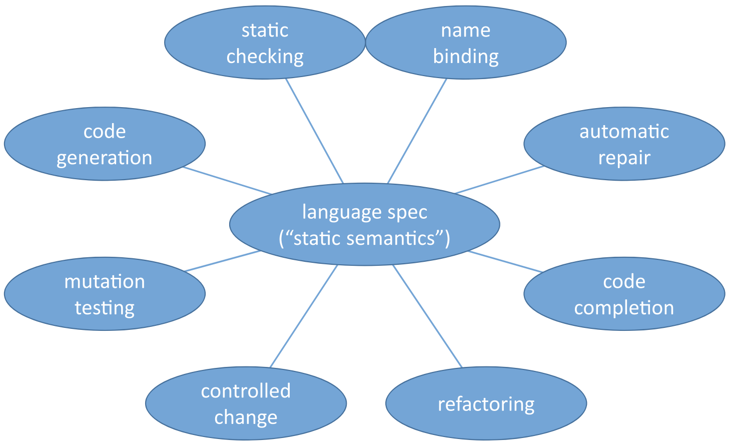

Fig. 2. Single investment, many pay-offs.

well-typed, etc. A program that passes all checks is considered well-formed, and ready for code generation or interpretation.

Static checkers can be implemented in a number of ways, with speed usually being a primary concern. However, there is a growing awareness of the fact that static checking is an option that can be traded for flexibility [6]. In addition, frameworks and even individual users of a programming language may define their own rules of well-formedness ("coding conventions"), which they want to see enforced by a compiler. JavaCOP [1] and the Checker framework [9] are two representatives of this movement in the Java field; for other, especially modelling, languages, the Object Constraint Language (OCL) is in use. Note that JavaCOP and OCL rely on constraints (but not constraint solving!) for static checking; since constraints are also the basis of constraint-based refactoring (Sect. 6.3), they are a hot candidate for our central capture of static semantics.

To see how rules of well-formedness can be expressed with constraints, we use a simple example. For any conventional programming language declaring names, we have a well-formedness condition stating that for all uses of (or references to) names there must exist a declaration introducing that name. More formally, we have that

$$\forall ref\ \exists decl : ref.name = decl.name \tag{8}$$

Applied to the program

```
int i;
bool j;
i = 1;
j = true;
```

this rule gives us the two constraints

$$\exists decl : ref_i.name = decl.name \tag{9}$$
$$\exists decl : ref_j.name = decl.name \tag{10}$$

Note that this rule does not express to which declaration a name *does* bind — it just expresses that for a program to be well-formed, a declaration must exist to which the name *can* bind.

Another well-formedness rule that is straightforwardly expressed using constraints is usually found in statically type-checked programming languages. It requires that the types on both sides of an assignment must equal. Expressing this rule with constraints and applying it to the above program will give us something like

$$ref_i.binding.type = lit_1.type \tag{11}$$
$$ref_j.binding.type = lit_{true}.type \tag{12}$$

As in Sect. 6.3, the constraint variables $ref_i.binding$ and $ref_j.binding$ represent the declarations (declared entities) the references bind to. The above well-formedness constraints (9) and (10) guarantee that a value can be found for these variables (because a declaration carrying the same name as the reference must exist), but for the type checking to be effective, the variables need to have values assigned. While one could argue that since constraints are generally undirected, the type constraints (11) and (12) can be used to compute the bindings of `i` and `j` (taking (11) and (12) as implicit specifications of the binding function; cf. Sect. 6.3.2), generally, type information does not suffice to determine binding unambiguously (what if `i` and `j` had the same type?) — this is what the names are for.

9 Name Binding

In contemporary programming languages like Java, name binding is the Siamese twin of static checking: One cannot live without the other. While a more thorough treatment of the interrelationship can be found in a companion briefing on "Name Binding" by Guido Wachsmuth (in this Volume), I want to emphasize here that name binding can in principle be expressed as a constraint satisfaction problem, and thus thrive on the same (constraint-based) specification of static semantics as all other tools discussed in this part of my briefing. In particular, expressing both static checking and name binding as one constraint satisfaction problem, the two tasks are automatically intertwined by the constraint solver.

Section 6.3, specifically constraints (4) and (5), already provided a brief glimpse of how names can be bound using constraints. Here, we note that for name binding, we need to complement the well-formedness constraints (9) and (10) guaranteeing that the names *can* be bound with a set of rules expressing *how* they are bound. Just like the constraints (4) and (5), the constraints

$$\mathit{ref}_i.\mathit{binding}.\mathit{name} = \mathit{ref}_i.\mathit{name} \tag{13}$$
$$\mathit{ref}_j.\mathit{binding}.\mathit{name} = \mathit{ref}_j.\mathit{name} \tag{14}$$

provide an *implicit specification* of the binding function applied to ref_i and ref_j. Interestingly, the well-formedness constraints (9) and (10) and the binding constraints (13) and (14) are related by Skolemization, with *binding* being the Skolem function.[10] Given that Skolemization makes (9) and (13), as well as (10) and (14), equisatisfiable, and given that (13) and (14) are more useful (they can be used for well-formedness checking and to compute name binding), the well-formedness constraints (9) and (10) appear dispensable. This is the more so since (13) and (14) can be used to detect ambiguity: if the constraint solver finds more than one value for a *binding* variable, binding cannot be uniquely determined. (Note how this amounts to replacing $\exists$ in (9) and (10) with $\exists_1$.)

10 Automatic Repair

If constraints can be used to check the well-formedness of a program, it seems natural that using constraint solving instead of constraint checking, the same constraints can also be used to correct a malformed program, simply by replacing fixated values in the failing constraints with constraint variables for which a constraint solver can compute new values. These new values then represent the fixes that mend the program.[11] However, contemporary IDEs implement auto-fixes (also called quick fixes) imperatively.

The auto-fixing implementations offered by contemporary IDEs are often short-sighted in that they offer fixes that break the program in other places. As for refactoring tools, there is a discussion whether this presents a bug or a feature: While a fix introducing a bug may not be a fix to some users, others may argue that it is still helpful if it saves manual edits. Without delving into this discussion, we note here that the same erratic behaviour can be obtained by solving violated constraints locally; if undesired, this behaviour can be cured by submitting all derivable constraints constraining the variables to the solver [42]. As for constraint-based refactoring (Sect. 6.3), this may turn out to be too expensive; again, as noted at the end of Sect. 6.3.4, much of the art of using constraints lies in deciding precisely which constraints to generate.

To give the reader an impression of how constraint-based auto-fixing works, we look at the following piece of malformed code written in our sample language:

```
int i;
bool j;
i = 1;
k = true;
```

[10] See [41] for some more details on how Skolemization relates well-formedness checking and binding.

[11] Note that we do not consider syntax errors here.

Most programmers would agree that the obvious way to fix the name binding problem is to replace k in the program text with j, but not knowing what the intention of the programmer was, all fixes

```
int i;           int i;           int k;           int i;
bool j;          bool j;          bool j;          bool k;
i = 1;           i = 1;           i = 1;           i = 1;
j = true;        i = true;        k = true;        k = true;
```

are equally conceivable. In fact, all these fixes can be derived from solving the (combined well-formedness and binding) constraint

$$\mathit{ref}_k.\mathit{name} = \mathit{ref}_k.\mathit{binding}.\mathit{name} \tag{15}$$

with $\mathit{ref}_k.\mathit{name}$, $\mathit{ref}_k.\mathit{binding}$, $\mathit{decl}_i.\mathit{name}$, and $\mathit{decl}_j.\mathit{name}$ all being variable. If a tool user has a preference for changing the names of references rather than declarations, this can be expressed by fixating the values of the constraint variables $\mathit{decl}_i.\mathit{name}$ and $\mathit{decl}_j.\mathit{name}$ to their current values; if the preference is on changing the names of declarations, the value of $\mathit{ref}_k.\mathit{name}$ can be fixated. Note that tying all variables to their current values makes the constraint unsolvable; it reflects the malformedness of the program in its present form.

As the alert reader will have noticed, two of the above fixes are short-sighted, in that they produce malformed programs. However, the resulting programs do not suffer from name binding problems — they are ill-typed. The problematic fixes can thus be prevented by adding the relevant type constraints to the constraints to be solved, in the above example

$$\begin{aligned}
\mathit{ref}_k.\mathit{type} &= \mathit{ref}_k.\mathit{binding}.\mathit{type} \\
\mathit{ref}_k.\mathit{type} &= \mathit{lit}_{\mathit{true}}.\mathit{type} \\
\mathit{decl}_i.\mathit{type} &= \mathit{type}_{\mathit{int}}.\mathit{val} \\
\mathit{decl}_j.\mathit{type} &= \mathit{type}_{\mathit{bool}}.\mathit{val}
\end{aligned}$$

The constraint variable connecting the name constraint (15) with the above type constraints is $\mathit{ref}_k.\mathit{binding}$; if its value is changed in the course of solving the name constraint, this change propagates to the type constraints, where it leads to the computation of the (new) type of the reference ref_k. With all constraint variables representing types having fixated values, the only solution of the joint constraint system is setting $\mathit{ref}_k.\mathit{name}$ to "j"; accepting new values for type variables also results in alternative fixes adjusting types in declarations or literals. Whether these additional fixes make sense and should be offered to the user is a different discussion; here it is important to note that (a) all fixes have been computed from the very same constraints used for detecting malformedness and (b) no fix computed by these constraints leaves the program malformed. As can easily be imagined, obtaining the same guarantee from an imperatively implemented auto-fix tool is hard.

A first delineation of constraint-based repair (together with constraint-based code completion) appeared in [46]; a more comprehensive treatise (based on constraint attribute grammars) has only recently been published [42].

11 Code Completion

Using constraints and constraint solving, auto-completion corresponds to computing values for constraint variables that do not have initial values that could be derived from a program as is. For instance, when the current program reads

```
int i;
bool j;
i = ^
```

and the user is about to enter a name at the caret position which represents a new reference, ref_{new}, the constraints

$$\begin{aligned} \mathit{ref}_{new}.\mathit{name} &= \mathit{ref}_{new}.\mathit{binding}.\mathit{name} \\ \mathit{ref}_{new}.\mathit{type} &= \mathit{ref}_{new}.\mathit{binding}.\mathit{type} \\ \mathit{ref}_{i}.\mathit{type} &= \mathit{ref}_{new}.\mathit{type} \end{aligned}$$

known from Sects. 8 and 9, and given that the current value of $\mathit{ref}_i.\mathit{type}$, int, is fixated, suffice to compute $\mathit{ref}_{new}.\mathit{name} :=$ “i” as the only valid completion of the program. Note that, analogously to auto-fixing, not generating all constraints would give us alternative completions (such as $\mathit{ref}_{new}.\mathit{name} :=$ “j”), which would however give us a malformed program.

Auto-completion is the dual to auto-fixing: most, if not all, incomplete programs can be thought of as being malformed (if only by introducing a random program element that renders the program so), so that the repairs represent the completions; and most, if not all, malformed programs can be thought of as being incomplete, so that eliminating the parts that are thought to cause the malformedness, and completing the resulting program, presents a fix. Therefore, it is highly advisable that auto-completion and auto-fixing are based on the same implementation; in current IDEs, however, this does not seem to be the case.

12 Controlled Change

It is safe to claim that programming is an alternating sequence of behaviour-altering change and refactoring. With refactoring being increasingly supported by corresponding tools, the next challenge is to support behaviour-altering changes also. At the very least, such support should guarantee that behaviour-altering changes leave a program well-formed; at a more advanced level, such support would ensure preservation of arbitrary, selected properties the program had before the change. We call a change that is guaranteed to preserve selected properties a *controlled change*. Refactoring is a controlled change in which the selected property is behaviour.

Arguably, the same discussion that is currently being led with regard to refactoring can also be led with regard to controlled change: Given some decision procedure ("oracle") for the desired properties, a tool or the programmer can go ahead and change a program in any way they deem appropriate, and the oracle can report the accidentally introduced errors later. In particular, any property that can be cast into a static check can be preserved this way. For Java programs this includes non-nullness, object confinement [20], etc. (see [1,9] for many more examples).

However, for academics at least, this would seem too modest a solution to be satisfactory. In fact, given how far we got in Part I of this briefing by computing from the same specification used for checking the very changes required to pass a check, why could the same not be achieved for other controlled changes? And indeed, the principle is the same: Identify the relevant invariants and make sure that they are preserved.

There is however a major difference between (arbitrary) controlled changes and refactoring: While refactorings usually follow patterns (refactorings are catalogued!), other controlled changes may not. Indeed, unless we go down to atomic changes (such as the changes used in mutation testing; see below), it is not clear whether we will ever see a compilation of controlled changes that receives the same recognition as Fowler's refactoring catalogue [12]. The source manipulation menus of contemporary IDEs such as Eclipse (containing entries like "surround with try-catch") may provide a starting point for implementing more complex controlled changes, however.

On the other hand, not all refactorings are catalogued: Firstly, a programmer is free to make any changes she pleases manually, and still demand tool support making sure that these changes constitute a refactoring. Secondly, tools can be devised for non-catalogued, ad-hoc refactorings (or "refactorings without names" [44]) also. If these can be made to work, other non-catalogued controlled changes should work also.

13 Mutation Testing

Mutation testing or, as it is sometimes also referred to, mutation analysis, is the technique of changing a program in such a way that it still compiles, but exhibits changed behaviour. Mutation testing is useful for testing the adequacy of test suites: for each behaviourally changed program — called *mutant* — that does not get caught, an additional test case should be added which catches it.

Traditional mutation testing works by applying mutation operators to programs. This suffers from two major problems: (1) The mutation operators may make the mutant malformed, and (2) the mutant may exhibit equivalent behaviour. While the former can be avoided by applying only mutation operators that cannot make a program malformed, or (somewhat expensively) by rejecting generated mutants that do not compile, the latter is a hard problem (undecidable in general) and in any case requires human inspection.

The attentive reader will have noticed that mutation testing (or, more specifically, the generation of non-equivalent mutants) is a special case of controlled change (Sect. 12). In fact, it is a complement of refactoring, one in which well-formedness is to be preserved, but behaviour is to be changed. The great advantages of implementing mutation testing as a controlled change activity are that (a) it does not limit mutations to the application of (single) mutation operators that cannot introduce malformedness, without having to pay the price of time-consuming compiler checks rejecting malformed mutants; and that (b) mutants are more likely to exhibit changed behaviour, namely when behaviour-critical dependencies (Sect. 6.1) have been changed by the mutation.

To give a concrete example of how this may work, we use our simple language again and start with the program

```
int i;
int j;
bool k;
i = 1;
j = 0;
k = true;
i = j;
```

A traditional mutation operator would replace the literal `1` with `0` (or vice versa), or `true` with `false`, which cannot make the program malformed. However, replacing names (identifiers) in the same shallow manner risks making the program ill-typed, as evidenced by replacing `i` with `k` in an assignment. By separating the set of constraints generated for the above program into ones that preserve well-formedness and ones that preserve behaviour-critical dependencies (see Sect. 6.1 for examples of these), and by negating one constraint of the latter kind, we can let the constraint solver compute a change that leaves the program well-formed, but (likely) exhibiting changed behaviour. For instance, for the last line of the above program we get the constraint

$$ref_j.binding.name = ref_j.name$$

which is solved by the assignment $ref_j.binding := decl_j$ (an extracted invariant in the case of refactoring; cf. Sect. 6.3.2). By adding a constraint

$$ref_j.binding \neq decl_j$$

(which negates the extracted invariant) and solving all constraints we get a new program in which `j` in the last line has been replaced with `i` (assuming that the names of declarations have been fixated; however, this is not required for the approach to work). Note that the well-formedness constraints (which have not been touched) prevent that `i` is replaced with `k`, since this would make the program ill-typed. Standard mutation operators do not have this language intimacy, and always apply their changes indiscriminately.

14 Code Generation

Code generation is the big brother of code completion (Sect. 11): it can be seen as the iterative completion of a program, starting with no (or the empty) program. A trivial approach to generating (arbitrary) well-formed code follows

Listing 1. Definite clause grammar generating (and accepting) only well-formed programs of a given length.

```
program(LOCs) -->
  decls([], Tab, LOCs, LOCsR),
  assigns(Tab, LOCsR, 0).
decls(Tab, Tab, LOCs, LOCs) --> [].
decls(In, Out, LOCs, LOCsR) -->
  {LOCs > 0},
  decl(In, Tmp),
  {LOCsD is LOCs - 1},
  decls(Tmp, Out, LOCsD, LOCsR).
decl(Tab, [var(Name, Type)|Tab]) -->
  type(Type),
  var(Name),
  {nonmember(var(Name, _), Tab)}.
type(int) --> [int].
type(bool) --> [bool].
var(i) --> [i].
var(j) --> [j].
assigns(Tab, LOCs, LOCs) --> [].
assigns(Tab, LOCs, LOCsR) -->
  {LOCs > 0},
  assign(Tab),
  {LOCsD is LOCs - 1},
  assigns(Tab, LOCsD, LOCsR).
assign(Tab) -->
  var(Name),
  {member(var(Name, Type), Tab)},
  [=],
  lit(Type).
assign(Tab) -->
  var(Name1),
  {member(var(Name1, Type), Tab)},
  [=],
  var(Name2),
  {member(var(Name2, Type), Tab)}.
lit(int) --> [0] | [1].
lit(bool) --> [true] | [false].
```

the *generate-and-test* paradigm: Syntactically well-formed programs, generated using the language's grammar or constructor invocations on an object-oriented capture of the language's abstract syntax, are subjected to static checking, dismissing all (semantically) malformed programs. However, while straightforward, this approach is usually too expensive for practical use.

A more practical approach to generating (arbitrary) well-formed code is to use attribute grammars enhancing the syntax rules of a target language with the rules of (static) semantics [25]. By replacing the computation of the attribute values of such a grammar with constraint solving, we can make sure that all programs generated by this grammar are well-formed [42]. For instance, the definite clause grammar (DCG) of Listing 1 can be used to generate all well-formed programs of our little sample language having a given number of lines of code (LOCs). Note that it uses unification and backtracking for constraint solving, which are also the main mechanisms of parsing in Prolog.

However, the use cases for generating *arbitrary* well-formed programs are fairly limited (but note that they include model checking language specifications [17] and testing programming tools [8], which both are highly relevant in the context of this briefing). What is needed more often is the generation of programs devised to fulfil some given purpose. In the most general case, a program generator would be an arbitrary program (written in a generator, or meta, language) that produces, from some given input, an output program in a target, or object, language that is well-formed according to the rules of that language.

The problems of and solutions for safe program generation are the topic of the companion briefing on "Structured Code Generation Techniques" provided by Yannis Smaragdakis et al. in this Volume, to which I would like to refer interested readers at this point. However, I will not leave them without noting that any proof of correctness of such program-generating programs via their compiler (the meta-language compiler), i.e., the proof of the fact that a given program-generating program can produce, for any input, only output programs that are well-formed in the target language, requires a full and formal capture of the well-formedness rules of the host language, for instance in first-order logic [23]. The constraints presented throughout this briefing are first order; hence, it seems justified to add structured code generators to the circle of programming tools profiting from the same single specification of a language's static semantics.

Acknowledgements. The work presented in this briefing has been supported by Deutsche Forschungsgemeinschaft (DFG) under grants STE 906/4-1&2 and STE 906/5-1. I thank Andreas Thies, Jens von Pilgrim, Frank Tip, Max Schäfer, and Bastian Ulke for their collaboration.

References

1. Andreae, C., Noble, J., Markstrum, S., Millstein, T.D.: A framework for implementing pluggable type systems. In: Tarr, P.L., Cook, W.R. (eds.) Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, 22–26 October 2006, Portland, Oregon, USA, pp. 57–74. ACM (2006). http://doi.acm.org/10.1145/1167473.1167479
2. Balaban, I., Tip, F., Fuhrer, R.M.: Refactoring support for class library migration. In: Johnson, R.E., Gabriel, R.P. (eds.) Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, 16–20 October 2005, San Diego, CA, USA, pp. 265–279. ACM (2005). http://doi.acm.org/10.1145/1094811.1094832

3. Bannwart, F., Müller, P.: Changing programs correctly: refactoring with specifications. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 492–507. Springer, Heidelberg (2006). doi:10.1007/11813040_33
4. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Longman Publishing Co. Inc., Boston (2000)
5. Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic reasoning for object-oriented programming. Sci. Comput. Program. **52**, 53–100 (2004). http://dx.doi.org/10.1016/j.scico.2004.03.003
6. Bracha, G.: Pluggable type systems. In: OOPSLA Workshop on Revival of Dynamic Languages, vol. 1. Citeseer (2004)
7. Brant, J., Steimann, F.: Refactoring tools are trustworthy enough and trust must be earned. IEEE Softw. **32**(6), 80–83 (2015). http://dx.doi.org/10.1109/MS.2015.145
8. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: Crnkovic, I., Bertolino, A. (eds.) Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, 3–7 September 2007, pp. 185–194. ACM (2007). http://doi.acm.org/10.1145/1287624.1287651
9. Dietl, W., Dietzel, S., Ernst, M.D., Muslu, K., Schiller, T.W.: Building and using pluggable type-checkers. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, 21–28 May 2011, pp. 681–690. ACM (2011). http://doi.acm.org/10.1145/1985793.1985889
10. Dig, D., Marrero, J., Ernst, M.D.: Refactoring sequential Java code for concurrency via concurrent libraries. In: 31st International Conference on Software Engineering, ICSE 2009, 16–24 May 2009, Vancouver, Canada, Proceedings, pp. 397–407. IEEE (2009). http://dx.doi.org/10.1109/ICSE.2009.5070539
11. Foster, S.R., Griswold, W.G., Lerner, S.: Witchdoctor: IDE support for real-time auto-completion of refactorings. In: Glinz et al. [16], pp. 222–232. http://dx.doi.org/10.1109/ICSE.2012.6227191
12. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley Object Technology Series. Addison-Wesley, Boston (1999)
13. Garrido, A., Meseguer, J.: Formal specification and verification of Java refactorings. In: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2006, pp. 165–174 (2006). http://dx.doi.org/10.1109/SCAM.2006.16
14. Ge, X., DuBose, Q.L., Murphy-Hill, E.R.: Reconciling manual and automatic refactoring. In: Glinz et al. [16], pp. 211–221. http://dx.doi.org/10.1109/ICSE.2012.6227192
15. Gligoric, M., Behrang, F., Li, Y., Overbey, J., Hafiz, M., Marinov, D.: Systematic testing of refactoring engines on real software projects. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 629–653. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39038-8_26
16. Glinz, M., Murphy, G.C., Pezzè, M. (eds.): 34th International Conference on Software Engineering, ICSE 2012, 2–9 June 2012, Zurich, Switzerland. IEEE (2012)

17. González, C.A., Büttner, F., Clarisó, R., Cabot, J.: EMFtoCSP: a tool for the lightweight verification of EMF models. In: Gnesi, S., Gruner, S., Plat, N., Rumpe, B. (eds.) Proceedings of the First International Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, 2 June 2012, pp. 44–50. IEEE (2012). http://dx.doi.org/10.1109/FormSERA.2012.6229788
18. Griswold, W.G.: Program restructuring as an aid to software maintenance. Ph.D. thesis, University of Washington (1991)
19. Griswold, W.G., Notkin, D.: Automated assistance for program restructuring. ACM Trans. Softw. Eng. Methodol. **2**(3), 228–269 (1993). http://doi.acm.org/10.1145/152388.152389
20. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. ACM Trans. Program. Lang. Syst. **29**(6) (2007). http://doi.acm.org/10.1145/1286821.1286823
21. Heuzeroth, D., Aßmann, U., Trifu, M., Kuttruff, V.: The COMPOST, COMPASS, Inject/J and RECODER tool suite for invasive software composition: invasive composition with compass aspect-oriented connectors. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 357–377. Springer, Heidelberg (2006). doi:10.1007/11877028_14
22. Hottelier, T., Bodík, R.: Synthesis of layout engines from relational constraints. In: Aldrich, J., Eugster, P. (eds.) Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, 25–30 October 2015, pp. 74–88. ACM (2015). http://doi.acm.org/10.1145/2814270.2814291
23. Huang, S.S., Zook, D., Smaragdakis, Y.: Statically safe program generation with safegen. Sci. Comput. Program. **76**(5), 376–391 (2011). http://dx.doi.org/10.1016/j.scico.2008.09.007
24. Kegel, H., Steimann, F.: Systematically refactoring inheritance to delegation in Java. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, 10–18 May 2008, pp. 431–440. ACM (2008). http://doi.acm.org/10.1145/1368088.1368147
25. Knuth, D.E.: Semantics of context-free languages. Math. Syst. Theory **2**(2), 127–145 (1968). http://dx.doi.org/10.1007/BF01692511
26. Lämmel, R.: Towards generic refactoring. In: Fischer, B., Visser, E. (eds.) Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming, Pittsburgh, Pennsylvania, USA, 2002, pp. 15–28. ACM (2002). http://doi.acm.org/10.1145/570186.570188
27. Ludwig, A., Heuzeroth, D.: Metaprogramming in the large. In: Butler, G., Jarzabek, S. (eds.) GCSE 2000. LNCS, vol. 2177, pp. 179–188. Springer, Heidelberg (2001). doi:10.1007/3-540-44815-2_13
28. Murphy-Hill, E.R., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Trans. Softw. Eng. **38**(1), 5–18 (2012). http://doi.ieeecomputersociety.org/10.1109/TSE.2011.41
29. Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
30. Opdyke, W.F., Johnson, R.E.: Creating abstract superclasses by refactoring. In: Kwasny, S.C., Buck, J.F. (eds.) Proceedings of the ACM 21th Conference on Computer Science, CSC 1993, Indianapolis, IN, USA, 16–18 February 1993, pp. 66–73. ACM (1993). http://doi.acm.org/10.1145/170791.170804

31. Overbye, J.L.: A toolkit for constructing refactoring engines. Ph.D. thesis, University of Illinois at Urbana-Champaign (2011)
32. Palsberg, J., Schwartzbach, M.I.: Object-Oriented Type Systems. Wiley Professional Computing. Wiley, Chichester (1994)
33. Raychev, V., Schäfer, M., Sridharan, M., Vechev, M.T.: Refactoring with synthesis. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (eds.) Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, 26–31 October 2013, pp. 339–354. ACM (2013). http://doi.acm.org/10.1145/2509136.2509544
34. Roberts, D., Brant, J., Johnson, R.E.: A refactoring tool for Smalltalk. TAPOS **3**(4), 253–263 (1997)
35. Roberts, D.B.: Practical analysis for refactoring. Ph.D. thesis, University of Illinois at Urbana-Champaign (1999)
36. Schäfer, M.: Specification, implementation and verification of refactorings. Ph.D. thesis, Oxford University Computing Laboratory (2010)
37. Schäfer, M., de Moor, O.: Of gnats and dragons: sources of complexity in implementing refactorings. In: Workshop on Refactoring Tools (WRT) (2009)
38. Schäfer, M., Thies, A., Steimann, F., Tip, F.: A comprehensive approach to naming and accessibility in refactoring Java programs. IEEE Trans. Softw. Eng. **38**(6), 1233–1257 (2012). http://doi.ieeecomputersociety.org/10.1109/TSE.2012.13
39. Soares, G., Gheyi, R., Massoni, T.: Automated behavioral testing of refactoring engines. IEEE Trans. Softw. Eng. **39**(2), 147–162 (2013). http://dx.doi.org/10.1109/TSE.2012.19
40. Soares, G., Mongiovi, M., Gheyi, R.: Identifying overly strong conditions in refactoring implementations. In: IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, 25–30 September 2011, pp. 173–182. IEEE Computer Society (2011). http://dx.doi.org/10.1109/ICSM.2011.6080784
41. Steimann, F.: From well-formedness to meaning preservation: model refactoring for almost free. Softw. Syst. Model. **14**(1), 307–320 (2015). http://dx.doi.org/10.1007/s10270-013-0314-z
42. Steimann, F., Hagemann, J., Ulke, B.: Computing repair alternatives for malformed programs using constraint attribute grammars. In: Visser, E., Smaragdakis, Y. (eds.) Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, 30 October – 4 November 2016, pp. 711–730. ACM (2016). http://doi.acm.org/10.1145/2983990.2984007
43. Steimann, F., von Pilgrim, J.: Constraint-based refactoring with foresight. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 535–559. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31057-7_24
44. Steimann, F., von Pilgrim, J.: Refactorings without names. In: Goedicke, M., Menzies, T., Saeki, M. (eds.) IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, Essen, Germany, 3–7 September 2012, pp. 290–293. ACM (2012). http://doi.acm.org/10.1145/2351676.2351726
45. Steimann, F., Thies, A.: From public to private to absent: refactoring Java programs under constrained accessibility. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 419–443. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03013-0_19

46. Steimann, F., Ulke, B.: Generic model assist. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 18–34. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41533-3_2
47. Sultana, N., Thompson, S.J.: Mechanical verification of refactorings. In: Glück, R., de Moor, O. (eds.) Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, 7–8 January 2008, pp. 51–60. ACM (2008). http://doi.acm.org/10.1145/1328408.1328417
48. Tip, F., Fuhrer, R.M., Kiezun, A., Ernst, M.D., Balaban, I., Sutter, B.D.: Refactoring using type constraints. ACM Trans. Program. Lang. Syst. **33**(3), 9 (2011). http://doi.acm.org/10.1145/1961204.1961205
49. Tip, F., Kiezun, A., Bäumer, D.: Refactoring for generalization using type constraints. In: Crocker, R., Jr., G.L.S. (eds.) Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, 26–30 October 2003, Anaheim, CA, USA, pp. 13–26. ACM (2003). http://doi.acm.org/10.1145/949305.949308
50. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E.: Use, disuse, and misuse of automated refactorings. In: Glinz et al. [16], pp. 233–243. http://dx.doi.org/10.1109/ICSE.2012.6227190