

Dijkstras Dream

Internal Iterators as Software Theorems

Jan Dockx

Marko van Dooren

Eric Steegmans

Report CW 340, Jun 2002



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Dijkstras Dream

Internal Iterators as Software Theorems

Jan Dockx

Marko van Dooren

Eric Steegmans

Report CW 340, Jun 2002

Department of Computer Science, K.U.Leuven

Abstract

One of the main goals of structured programming is to make it possible for programmers to reason about and prove the correctness of programs. The assertions that describe the algorithm make explicit the complexity of the code. Encapsulation and inheritance remove an important part of that complexity in object-oriented programming. This paper shows that the reduction in complexity we get by replacing iteration structures by internal iterators is so big that it brings us very close to zero-defect software becoming viable.

Keywords : object oriented programming, structured programming, correctness proof, iteration, iterator, sequence, selection, framework, design pattern, jutil.org

CR Subject Classification : D.1.5, D.2.4, D.2.11, D.2.13, D.3.3.

Dijkstra's Dream

Internal Iterators as Software Theorems

Jan Dockx, Marko van Dooren, Eric Steegmans

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A
3001 Leuven, Belgium
{Jan.Dockx, Marko.vanDooren, Eric.Steegmans}@cs.kuleuven.ac.be
<http://www.cs.kuleuven.ac.be/>

Abstract. One of the main goals of structured programming is to make it possible for programmers to reason about and prove the correctness of programs. The assertions that describe the algorithm make explicit the complexity of the code. Encapsulation and inheritance remove an important part of that complexity in object-oriented programming. This paper shows that the reduction in complexity we get by replacing iteration structures by internal iterators is so big that it brings us very close to zero-defect software becoming viable.

Version 1: 26 November 2001

Submission to ECOOP 2002

Version 2: 28 May 2002

CW Report 340

1. Introduction

Structured programming, as introduced by Dijkstra in the late 1960's, forms the basis of modern imperative programming, including imperative object oriented programming. The intention of structured programming was to solve the software crisis by giving programmers a means to reason about and proving the correctness of their code. This should be done by making explicit the knowledge a programmer has about his code through assertions that describe the state of the computer at different points in an algorithm. To make this possible, programmers should restrict themselves to a set of well-known control structures in their implementations. These assertions make visible the intrinsic complexity of code.

The ultimate goal of structured programming is to produce *zero-defect* software. Zero-defect software is proven correct, but the assertions that make up the proof are written during the development of the algorithm, not afterwards. Writing down the assertions that make explicit the knowledge of the programmer about the algorithm he is developing helps against making errors in reasoning from the start. The software development community has never really adopted this way of working, and reverted to testing instead, because writing down the assertions and proving the correctness of code is extremely time intensive and requires highly trained personnel. It is deemed economically impractical. It is debatable whether this is true when the entire life cycle of a program is taken into account, but the short-term cost analysis is more important in the software development industry.

Algorithms in state-of-the-art object oriented programs are less complex than their traditional counterparts. In this paper, we contend that this is so because the carriers of some the complexity are replaced by object oriented techniques that are easier to handle. This is an accepted practice for sequence and selection. State-of-the-art object oriented programs still do feature complex iterations though. This is highly surprising, since iterations are well known to be the most complex control structures in algorithms. It was Dijkstra's dream that a generally accepted set of *software theorems* would emerge for iterations, which would make it possible for programmers to reason about iterations on a more abstract level, thus removing some or all of the complexity of the iteration from the algorithm.

We will show that *internal iterators* are the incarnation of Dijkstra's software theorems for iterations, and thus should get their rightful place in object oriented programming techniques next to encapsulation and inheritance. Internal iterators are the closure in the transition from structured programming to object oriented programming. We believe that with internal iterators, we are very close to zero-defect software becoming a viable alternative for software developers.

In the next chapter, we will refresh the reader's memory about the basic concepts of structured programming. In chapter 3 we will show how sequence and selection have been replaced by other concepts in state-of-the-art object oriented programming. This leaves iteration. The rest of the paper will show how we get very close to the actual goal of structured programming by replacing iteration by internal iterators in object oriented programs. Chapter 4 discusses the internal iterator design pattern and introduces the family of internal iterators the authors contributed to jutil.org. In chapter 5, we show that the reduction of complexity we get by replacing iterations with internal iterators is immense, by comparing different implementations of an example algorithm. In chapter 6, we ask ourselves where the complexity that is removed by object oriented practices from algorithms has gone to.

2. Structured Programming and Dijkstra's Dream

Source code is more complex if it is more difficult for humans to understand it and reason about it. Understanding and reasoning about code is the most important goal of *structured programming* [Dijkstra 1969-1970][Dijkstra 1972]. The most well known part of structured programming is that programs should be restricted to a limited set of well-understood control structures. This position is most famous through the 1968 "*Letter to the editor, Go To Statement Considered Harmful*" [Dijkstra 1968]. Since then, most programming courses teach programming based on 4 basic concepts: *sequence*, *selection*, *iteration* and *procedures*. What seems all but forgotten is that the most important point to Dijkstra is that software should be proven correct, by making explicit the knowledge the programmer has about the state of the computer at each point in the algorithm. This is done by writing logic assertions about the state, and proving them, while developing the algorithm. For each of the 4 basic concepts, proof techniques are presented: *enumeration* for sequence and selection, *mathematical induction* for iteration, and *abstraction* for procedures (see [Dijkstra 1969-1970], page 8).

2.1. Sequence, Selection and Enumeration

To prove the correctness of a sequence, a substitution, applying the contract of the executed step to the pre-state, leads to the post state. Most often, proving the preconditions of the executed step is equally straightforward. To prove the correctness of a selection structure, we need to prove that the desired post state is reached via each possible branch of the selection, given the branching conditions. This is done by choosing a post state for the entire selection structure that is the generalization of the post state of each branch.

Reasoning about and proving the correctness of sequence and selection is relatively simple. Any difficulties that there might be, arise from the sheer length of the sequence or the intricacy of the, possibly deeply nested, selection structure.

Note however that selection structures are the most important source of instability in software under evolution. If the difference between cars, vans and trucks is relevant, different program behavior will be required for a number of functionalities. In traditional programming languages, this results in the selection structure being repeated many times throughout an application. Any change to the context of the program, e.g., a redefinition of the partition with the advent of monovolumes, requires changes in all the occurrences of the selection structure. In traditional software, these occurrences will be very hard to find, and even hard to recognize.

2.2. Iteration and Mathematical Induction

To prove the correctness of an iteration structure, a loop invariant is specified, which holds true at the start and end of each iteration. The post state should be proven from the loop invariant and the stop criterion. The internal correctness of the loop needs to be proven by proving that the loop invariant holds at the end of the loop, given that it holds at the start of the loop and given the negation of the stop criterion. This proves the induction. What is left is the proof that the loop invariant will hold true the first time the loop is entered, from the state immediately before the loop. Proving that the loop ends is done by formulation a variant assertion, an integer function that should evolve to 0.

Reasoning about and proving the correctness of iteration is extremely difficult. In our opinion, iterations are the reason why the idea of developing zero-defect software has been all but abandoned, and the industry reverted to testing. About the correctness proof of a (relatively simple) iteration, Dijkstra says:

"To tell the honest truth: the pomp and length of the above proof infuriates me as well! [...]"

“Of course I would not dare to suggest (at least at present!) that it is the programmer's duty to supply such a proof whenever he writes a simple loop in his program. If so, he could never write a program of any size at all! [...]”

[Dijkstra 1969-1970], p12

2.3. Procedures and Abstraction and Software Theorems

Procedures are, from the standpoint of structured programming, a technique to make reasoning about and proving the correctness of programs easier. Proving the correctness of a program as a whole, with only the semantics of the programming language to work with, is highly impractical.

Subroutines, procedures or methods are equivalent to *lemmas* in mathematical proofs. Part of the overall problem is isolated and reasoned about and proven correct separately. The results of the lemma can then be used in the overall proof, which becomes much more tractable. Abstraction is made from the actual algorithm of the procedure (the *how*), and the programmer can focus on *what* the procedure accomplishes. To this end, procedures are *specified* using a *contract*. The *postcondition* expresses which assertions will hold after execution of the procedure, *preconditions* express under which circumstances the postconditions are proven.

2.4. Dijkstra's Dream: Software Theorems

To Dijkstra, abstraction is the solution for the difficulties in reasoning about and proving the correctness of a program. He compares reasoning about software with plane geometry. Proving the correctness of an algorithm (in casu, an iteration) with only the semantics of the programming language as building blocks ...

“[...] fills me with the same kind of anger as years ago the crazy proofs of the first simple theorems in plane geometry did, proving things of the same degree of "obviousness" as Euclid's axioms themselves. [...] Proving the correctness using only the semantics of the programming language] would be as impractical as reducing each proof in plane geometry explicitly and in extenso to Euclid's axioms.[...]”

[Dijkstra 1969-1970], p12

Dijkstra dreams of a *“set of [software] theorems whose usefulness has been generally accepted”* [Dijkstra 1969-1970].

“[...] when a programmer considers a construction like [the iteration above] as obviously correct, he can do so because he is familiar with the construction. I prefer to regard this behavior as an unconscious appeal to a theorem he knows [...] But we could call our assertions about [the iteration above], say, “the Linear Search Theorem”, and knowing such a name it is much easier (and more natural) to appeal to it consciously.”

[Dijkstra 1969-1970], p12

Although a programming problem could be implemented in numerous ways, a programmer who has knowledge about the generally accepted set of software theorems would use one of the algorithms described by a known theorem. He would be crazy to invent a new algorithm, and do all the reasoning and proving for the new algorithm from scratch. Dijkstra contends that this is already so, on an intuitive level:

“[...] the intuitively competent programmer is probably the one who confines himself, whenever acceptable, to program structures with which he is very familiar, while becoming very alert and careful whenever he constructs something unusual (for him).”

[Dijkstra 1969-1970], p12

Procedures are a nice way to alleviate the complexity of a monolithic program, but in themselves they can merely act as lemmas. Lemmas are different from theorems in that they are specific to a particular problem, in casu a particular application. Software theorems are meant to be more general. They should be generally accepted over the field of programming as a whole.

2.5. Proving is Easy, Programming is Difficult

The complexity of program assertions and their proof reveals the actual, often hidden, complexity of the source code. Proving the correctness of code is difficult, not because of the proof techniques, but because of the code.

“[...] the length of the proof we needed in our last example is a warning that should not be ignored. [...] I am inclined to conclude from this length that programming is more difficult than is

commonly assumed: let us be honestly humble and interpret the length of the proof as an urgent advice to restrict ourselves to simple structures whenever possible and to avoid in all intellectual modesty “clever constructions” like the plague.”

[Dijkstra 1969-1970], p13

Below we will try to show that the algorithms in state-of-the-art imperative object oriented programs are significantly less complex than traditional imperative, structured and procedural programs. It is far more easy to reason about and prove the correctness of object oriented programs than it is for traditional programs.

3. Object Oriented Programs Are Easier

Algorithms in object oriented programs are easier to reason about than algorithms in traditional structured programs, because the basic concepts of traditional structured programming have become less important in state-of-the-art object oriented programming (except for abstraction). Their place is taken by other basic concepts. From this follow 3 *rules of thumb* for good object oriented design.

In this section, we will discuss the demise of sequence and selection, and the increased importance of procedures and abstraction. These shifts are generally accepted. Iteration is still a part of state-of-the-art object oriented programming.

3.1. Procedures and Abstraction

A word more commonly used nowadays that covers abstraction, is *encapsulation*. Abstraction as a basic concept is not replaced in object orientation, but expanded. In structured programming, procedures were the first technique that enabled programmers to isolate pieces of the program, so that they can reason about their implementation in isolation, and so that the complexity of the actual implementation is removed from the main program.

This idea is the basis of state-of-the-art object oriented programming. Another abstraction layer is added: types. The idea to specify these extracted pieces of code using a contract is expanded by the object oriented contract paradigm [Meyer 1991][Meyer 1997]. Type invariants are introduced as generalizations of pre- and postconditions on the level of the types. They serve as an anchor point for reasoning about and proving the correctness of the implementations of the type and algorithms that use the type. The types as a whole can be seen as even more powerful lemmas in an object oriented program.

The first part of Dijkstra’s dream comes true when a set of such types becomes available whose usefulness has been generally accepted. This can be said to be true, e.g., of the standard Java API library, although they are not (yet) proven correct.

3.2. Sequence

In object oriented programs, algorithms are very much shorter than in procedural languages [Churcher, Shepperd 1995]. It is clear that the introduction of subroutines itself in pre-object-oriented times was responsible for a first large reduction in source code complexity. Object orientation takes this further.

For an automated theorem prover, this makes a big difference. One of the biggest tasks for an automated theorem prover is to come up with assertions that apply in consecutive places in an algorithm. Through shorter methods and design by contract, we actually relieved the automated theorem prover of much of that burden. The contracts of the classes and methods take the place of the lemmas the theorem prover would otherwise need to devise itself.

From this follows our first rule of thumb for good object oriented programming:

Any method longer than 10 lines is suspect. Consider splitting it in separate methods.

3.3. Selection

In state-of-the-art object oriented programs, selection is replaced by *dynamic binding*. Foremost, this makes programs more stable under evolution. Further, it removes the complexity of intricate nested selection structures.

Our second rule of thumb of good object oriented programming states:

Any selection structure is suspect. Consider replacing it by a polymorph type structure and dynamic binding.

3.4. Iteration

Iteration is still part of state-of-the-art object oriented programming. Yet, it has been known for over 30 years that iterations are the most complex programming structures. Current programming languages offer no replacement for iterations like they do for selection.

It is interesting to see that recently replacements for explicit iterative data structures, like arrays or linked lists or trees, have found their way into the mainstream. The Java Collections API offers classes that encapsulate complex data structures. Some of the design decisions, such as the ugly `NotSupportedException` that mutators may throw, can be debated, but as a whole, the API is acceptable. Iterative control structures are left out though.

Our third and last rule of thumb of good object oriented programming states:

Any iteration structure is suspect. Consider replacing it by a proven internal iterator.

While the first two rules of thumb cited above are part of the mainstream state-of-the-art object oriented thinking, the last rule is not. We will show in the next 2 sections of this paper that this rule of thumb reduces the complexity of software to such an extent that reasoning about and proving correctness of implementations becomes easy. This is the last changeover to be made from structured programming on the way of making the dream of structured programming a reality.

4. Internal Iterators

Internal iterators are discussed in [Gamma, Helm, et al. 1994]¹ as a side note. Gamma et al. claim that internal iterators are less than perfect because “*creating a subclass for every different traversal is [much] work*”. This is indeed true in the C++ context the book focuses on, but it is not in Java, which offers *anonymous inner classes*.

Internal iterators are not supported by the Collections API offered in Java since JDK 1.2. This is by design:

“[Collection Interface,] 2. Why don't you provide an “*apply*” method in *Collection* to apply a given method (“*upcall*”) to all the elements of the *Collection*?

This is what is referred to as an “Internal Iterator” in the “Design Patterns” book (Gamma et al.). We considered providing it, but decided not to as it seems somewhat redundant to support internal and external iterators, and Java already has a precedent for external iterators (with Enumerations). The “throw weight” of this functionality is increased by the fact that it requires a public interface to describe upcalls.”

[Java Collections API Design FAQ 1997]

We believe that this decision is a major mistake, because internal iterators, or any other incarnation of what is essentially known as the Visitor pattern [Gamma, Helm, et al. 1994], are the prime abstractions to remove iterations, the most difficult code to reason about and prove the correctness of, from algorithms. This is recognized by [Gamma, Helm, et al. 1994]:

“*Note how the client doesn't specify the iteration loop. The entire iteration logic can be reused. This is the primary benefit of an internal iterator.*”

[Gamma, Helm, et al. 1994], page 269

Different incarnations of internal iterators are exactly what we encounter most in the utility libraries of organizations we have contact with, and in our own projects. We believe the internal iterators do carry their weight, and more than that. Internal iterators are the *software theorems* for iterations Dijkstra dreamed about.

The enormous benefits of an approach that annihilates the need for iterations have been recognized since long in, e.g., functional programming and relational databases. In functional programming, meta functions that *filter*, *map* or *accumulate* elements of a collection are part of the functional programming

¹¹ in the sample code of the Iterator pattern, note 6, page 267

memes for decades². The wish to get rid of iterations using *cursors* is the basis for the introduction of relational databases. Instead of iteration over tables, relational databases offer functionality that manipulates tables as a whole, as *sets* of tuples: *selection*, *projection* and *aggregate functionality* are the equivalents of the filter, map and accumulate meta functions.

In the rest of the chapter we present a family of internal iterators for the Java Collection API we submitted to `jutil.org` [Blakeley, Dockx, van Dooren 2001]. Two internal iterators we use in chapter 5 are presented in detail. In that chapter, we compare traditional versions of an algorithm using iterations and a version using our internal iterators, concerning their fitness for reasoning about them and proving their correctness.

4.1. `jutil.org`

`jutil.org` is an Open Source project that offers general Java utility code. The project was started in 2001 by Peter Blakeley and the authors of this paper, and is hosted on SourceForge [Blakeley, Dockx, van Dooren 2001].

Like so many other people, we found that we were writing more or less the same utility code over and over again in different projects. The idea is that we, and other interested parties, submit their organization's utility code to `jutil.org`. The benefit will be a uniform API, with optimized, and correct implementations.

Most often, this utility code is code that *should-have-been-there* in the first place. Because of that, the functionality the contributions offer, would often better be supported by instance methods in, e.g., the Java Standard API. Because that code collection is closed to us in the short term, we, and many developers with us, are forced to place this functionality elsewhere, often as static methods. For our internal iterators, an `apply` instance method which takes a `Visitor` object as argument in the `Collection` interface is not possible. The classes we offer are self-contained. In this particular case, this might be beneficial, since now different kinds of internal iterators do not have to extend a common type, and the `apply`-method can have a return type tuned to the specific needs of the internal iterator. Over time, the Java Standard API might include the functionality that is covered by a `jutil.org` contribution. So much the better. If this happens, the `jutil.org` entry becomes deprecated. This is exactly what is happening with another of our contributions, which gives the program structured access to the stack trace carried by a `Throwable`. Our `StackTrace.asList(Throwable)` method will become deprecated when JDK 1.4 is released.

`jutil.org` does require contributions to be well documented using the contract paradigm [Meyer 1991][Meyer 1997] and verified. Documentation using the contract paradigm can be informal or formal. Documentation in English as we are accustomed from the standard Java API by Sun (which is written with the contract paradigm in mind) is the minimum. Our `jutil.org` contributions are formally specified using JML³ [Leavens]. Contributions should come with verification, either static, i.e., with a proof of correctness, or dynamic, i.e., with tests.

Readers are invited to submit their utility code collections to the project. Utility code collections are exactly the kind of project that benefits the most from an Open Source approach. Large scale use will lead quickly to stable and correct code, and most organizations already have such code collections, so you will not be giving your competitors an advantage by making your collection public. Instead, it will be your source collection that will be optimized and extended by peers.

4.2. `jutil.org` Internal Iterators

The internal iterators we submitted to `jutil.org` are in the package `org.jutil.java.collections`. Amongst other related utility code, there currently are general internal iterators called `Visitor` and `MapVisitor`, respectively for `Collections` and `Maps`, and `RobustVisitor` and `RobustMapVisitor`. The latter are to be used when an exception can occur which cannot be dealt with locally during the visit of an element.

In the footsteps of functional programming culture and relational databases, our internal iterator family offers a `Filter`, `Mapping` and `Accumulator` class. `TypeFilter` is a subclass of `Filter` that uses the type of elements in the collection via reflection as filter criterion. `AndFilter` makes the logic conjunction of other filters (intersection of all the collections). All the other internal iterators present in the family at the time of this writing are special types of accumulators, such as `Transi-`

² Actually, there are no iterations in functional programs, but recursion, and specifically tail recursion, but the difference is irrelevant to our story.

³ with some extensions

tiveClosure. Two boolean accumulators, or quantifications, are shown in detail in the rest of this chapter, because those will be used in chapter 5.4.

The family of internal iterators does not offer classes for set union, intersection or difference, since this functionality is offered by the Java Collections API.

4.3. Internal Iterators for Quantification

In the example, below we will use two internal iterators from the `org.jutil.java.-collections` family: `ForAll` and `Exists`. Here we present their unified contract, the conjunction of the contract of the actual classes with the contract of their supertypes.

4.3.1. Universal Quantifier

The first boolean accumulator is the `ForAll` class which checks whether some boolean criterion is satisfied by all elements in a collection (see **Fig. 1**).

The class has one abstract method `boolean criterion(Object element)` that has to be given an implementation in a subclass. The `boolean in(Collection collection)` method checks whether all elements in `collection` satisfy the criterion. The pure model method `boolean isValidElement()` is used as an abstract precondition ([Meyer 1997], page 576) for the `criterion()` and `in()` methods. It functions as a tunnel for information the client has about the elements of the collection, through the `in()` method, to `criterion()`.

```
public abstract class ForAll {

    /*@
     * @ public behavior
     * @
     * @ (* this method should be consistent with equals;
     * @ subclass can say more *)
     * @ post (\forall object o1;;
     * @ (\forall object o2; o2.equals(o1);
     * @ isValidElement(o1) ==
     * @ isValidElement(o2));
     * @
     * public model boolean isValidElement(Object element);
     */

    /*@
     * @ public behavior
     * @
     * @ pre isValidElement(element);
     * @
     * @ (* criterion should be consistent with equals;
     * @ subclass can say more *)
     * @ post (\forall object o1;;
     * @ (\forall object o2; o2.equals(o1);
     * @ criterion(o1) == criterion(o2));
     */
    abstract public boolean criterion(Object element);

    /*@
     * @ public behavior
     * @
     * @ pre (\forall Object o, collection.contains(o);
     * @ isValidElement(o));
     * @
     * @ post \result ==
     * @ (\forall Object o; collection.contains(o);
     * @ criterion(o));
     */
    final public boolean in(Collection collection) {
        ...
    }
}
```

Fig. 1. Specification of `org.jutil.java.collections.ForAll`

4.3.2. Existential Quantifier

The second boolean accumulator is the `Exists` class. It is used to check whether a collection contains at least one element meeting the criterion defined in the `boolean criterion(Object element)` method. As with the `ForAll` class, the `boolean in(Collection collection)` method returns the result for the given collection. Again, we use `isValidElement()` as an abstract precondition. The specification of the class is shown in **Fig. 2**.

```
public abstract class Exists {

    /*@
    @ public behavior
    @
    @ (* this method should be consistent with equals;
    @   subclass can say more *)
    @ post (\forall object o1;;
    @       (\forall object o2; o2.equals(o1);
    @         isValidElement(o1) ==
    @         isValidElement(o2)));
    @
    public model boolean isValidElement(Object element);
    @*/

    /*@
    @ public behavior
    @
    @ pre isValidElement(element);
    @
    @ (* criterion should be consistent with equals;
    @   subclass can say more *)
    @ post (\forall object o1;;
    @       (\forall object o2; o2.equals(o1);
    @         criterion(o1) == criterion(o2)));
    @*/
    abstract public boolean criterion(Object element);

    /*@
    @ public behavior
    @
    @ pre (\forall Object o, collection.contains(o);
    @     isValidElement(o));
    @
    @ post \result ==
    @       (\exists Object o; collection.contains(o);
    @         criterion(o));
    @*/
    final public boolean in(Collection collection) {
        ...
    }
}
```

Fig. 2. Specification of `org.jutil.java.collections.Exists`

5. Comparison of Different Versions of an Algorithm

To show that code using internal iterators instead of explicit iterations is much less complex, we will elaborate different versions of an example method. As a measure of the complexity of the implementation, we will discuss the correctness proofs of each version. The correctness proof makes explicit the reasoning about the code, and thus can be used as a qualitative measure for the complexity of the code⁴.

We assume that we are not working in a concurrent environment, so the collections cannot be changed by other threads during the method execution, and no `ConcurrentModificationExceptions` will be thrown while working with the collections. For `java.util.Iterator` we

⁴ The implementations shown are not tweaked for performance. More performant versions are even more complex.

use a specification based on JML (see `java.util.Iterator.jml` and `edu.iastate.cs.-jml.models.JMLObjectSequence.jml-refined`, [Leavens]). We extend this specification with an instance model field `previousElements`, which we need for our loop invariants⁵.

5.1. Example

The example method checks whether all persons in one collection are exactly one year older than some person in another collection. The specification for the method is show in listing **Fig. 3**.

```

/*@
 @ public behavior
 @
 @ pre first != null;
 @ pre (\forall Object o; first.contains(o);
 @     o instanceof Person);
 @ pre ! first.contains(null);
 @ pre second != null;
 @ pre (\forall Object o; second.contains(o);
 @     o instanceof Person);
 @ pre ! second.contains(null);
 @
 @ post \result == (\forall Person p1;
 @                 first.contains(p1);
 @                 (\exists Person p2;
 @                 second.contains(p2);
 @                 p1.age() == p2.age() + 1));
 @*/
public boolean allOneYearOlder(
    final Collection first,
    final Collection second);

```

Fig. 3. Specification of the `allOneYearOlder()`

Fig. 4 shows the type invariants of the class `Person`. They state that a person must have an age between 0 and 2000, and that two equal persons objects are the same object. The first is used to prevent overflow in calculations with the age, the second is needed to avoid semantic problems with the `contains` method of Java Collections, which uses `equals`.

```

/*@
 @ public invariant
 @     (\forall Person p1; p1 != null;
 @     (\forall Person p2; p2 != null;
 @     p1.equals(p2) <=> (p1 == p2)));
 @ public invariant age() >= 0;
 @ public invariant age() <= 2000;
 @*/

```

Fig. 4. Type invariants of class `Person`

5.2. Implementation Using Nested Iteration

Listing **Fig. 5** shows an implementation of the example method using iterators in a nested loop.

```

public boolean allOneYearOlder(
    final Collection first,
    final Collection second) {
    Iterator outer = first.iterator();

```

⁵ Our first idea, to derive the `previousElements` from the `UnderlyingCollection` and the `nextElements` collection that are offered by the JML specification of `Iterator`, was not correct. The symmetric difference of the original collection and `nextElements` does not contain elements that actually were visited already, but are repeated in that part of the collection that is not visited yet, a.k.a., `nextElements`. This is not important for `Sets`, which cannot contain duplicates, but it is for `Collections` in general.

```

boolean outerBool = true;
while(outer.hasNext() && outerBool) {
    Person person1 = (Person)outer.next();
    Iterator inner = second.iterator();
    boolean innerBool = false;
    while(inner.hasNext() && (!innerBool)) {
        Person person2 = inner.next();
        innerBool = (innerBool ||
            (person1.age() == person2.age()+1));
    }
    outerBool = outerBool && innerBool;
}
return outerBool;
}

```

Fig. 5. Implementation of `allOneYearOlder()` using iterators in a nested loop

The nested loop is the traditional way of implementing a method like this in structured programming. Nested loops are among the most difficult things to reason about. An even more basic version first extracts the persons from the collections into an array, and iterates over the arrays. This version and the complete correctness proof can be found in [Dockx, van Dooren, Steegmans 2001].

The complete correctness proof of this version takes 40 pages.

5.3. Implementation Using Simple Iteration and a Helper Method

When writing an algorithm using a nested iteration, it is common to introduce an additional method that takes care of the inner loop. This is an application of the first rule of thumb for good object oriented programming (see section 3.2). Both the complexity of the implementation and the proof are reduced. The implementation of the example method of this variant is shown in **Fig. 6**. **Fig. 7** shows the specification and a possible implementation of the helper method.

```

public boolean allOneYearOlder(
    final Collection first,
    final Collection second)
{
    Iterator iter = first.iterator();
    boolean acc = true;
    while(iter.hasNext()) {
        Person person = (Person)iter.next();
        acc = acc && oneYearOlderThanSome(person, second);
    }
    return acc;
}

```

Fig. 6. Implementation of the `allOneYearOlder()` using a helper method

```

/*@
  @ public behavior
  @
  @ pre person != null;
  @ pre collection != null;
  @ pre (\forall Object o; collection.contains(o);
  @     o instanceof Person);
  @ pre ! collection.contains(null);
  @
  @ post \result == (\exists Person p2;
  @         collection.contains(p2);
  @         person.age() == p2.age() + 1));
  @*/
public boolean oneYearOlderThanSome(
    Person person, Collection collection) { //P0
    Iterator iter = collection.iterator();
    boolean acc = false; //P1
    while(iter.hasNext()) { //P2
        Person otherPerson = (Person)iter.next();
        acc = acc ||
            (person.age() == otherPerson.age() + 1); //P3
    }
}

```

```

    }                                     //P4
    return acc;                           //Pn
}

```

Fig. 7. Specification and implementation of `oneYearOlderThanSome()` helper method

The complete correctness proofs can be found in [Dockx, van Dooren, Steegmans 2001]. It shows that this version is easier to reason about than the version with the nested loop. Below we give a schematic proof of the helper method `oneYearOlderThanSome()`.

A program point (`//P#`) denotes the state after the statement on that line has been executed. The proof lists the assertions that describe the relevant part of that state and proves that they are correct. We also need to prove that each step in the algorithm is acceptable, i.e., it does not violate preconditions of called code. Furthermore, we need to take the possibility of exceptions into account. If an exception is thrown, the postcondition will not be reached, but other assertions apply. In this case, we prove that no exceptions will be thrown.

5.3.1. P0

At program point `P0`, we know that the preconditions of the method are satisfied. For `oneYearOlderThanSome()` this means that the following assertions are true.

- `person != null`
- `collection != null`
- `(\forall \text{Object } o; \text{collection.contains}(o);`
`o instanceof Person)`
- `! collection.contains(null)`

5.3.2. P1

At program point `P1` we know the initial values of `iter` and `acc`.

- `iter = collection.iterator();`
 - `acc = false`
- We also know that
- `iter != null`

Actually, we can't prove this last assertion formally since the Java documentation says nothing about this in the documentation of the `iterator()` method, nor does the JML specification. We assume that this information should be in the contract of the method. All assertions of `P0` are still valid.

There is the possibility for a `NullPointerException` if `collection` is `null`, but we know from `P0` that this is not the case.

5.3.3. P2

The loop invariant is:

- `acc == (\exists \text{Person } p;`
`iter.previousElements.contains(p);`
`person.age() == p.age() + 1)`

`iter.previousElements.contains(p)` means that `p` is already visited. The loop variant can be written formally:

- `iter.nextElement().size()`

From the loop invariant, we will prove `P4`. To prove the loop invariant, we use induction. From `P1` and the loop condition `iter.hasNext()` we will prove that the loop invariant holds initially. From `P3` and the loop condition we will prove that the loop invariant holds each iteration step if the previous iteration step upheld it.

The loop invariant is met initially because no elements have been visited yet (`iter.previousElements.isEmpty()`), so the `\exists` predicate results in `false`. In `P1`, `acc` indeed is `false`. `P3` will be a verbatim copy of the loop invariant, so the induction step is proved trivially.

The loop variant will evolve to 0, since in every iteration step the collection `iter.nextElement()` will shrink.

5.3.4. P3

In `P3`, the loop invariant has to be satisfied for the new states of `iter` and `acc`.

- `acc == (\exists Person p;
 iter.previousElements.contains(p);
 person.age() == p.age() + 1)`

As given, we can use the loop invariant in P2 (the `\old` state) and the loop condition.

- `\old(acc == (\exists Person p;
 iter.previousElements.contains(p);
 person.age() == p.age() + 1))`
- `\old(iter.hasNext())`

Again, no exceptions will be thrown. The preconditions forbid objects `person` and `otherPerson` to be null. The iterator is not null as is shown in P1. The addition of 1 to the age of the other person cannot result in an overflow because of the type invariants of `Person`. The call to `next()` could throw a `NoSuchElementException`, but it will not because `iter.hasNext()` (the loop condition) is guaranteed to be true here.

We now have to prove that the loop invariant is still satisfied after traversing the body of the loop. To prove the loop invariant we must consider two cases:

A) `person.age() == otherPerson.age() + 1`

In this case, the person currently being visited (`p`) is one year younger than `person` is. Consequently, `acc` will be true.

- `(acc == \old(acc) || true) ⊢ acc == true`

Since `person.age() == otherPerson.age() + 1`, it is also true that some visited person was exactly one year younger than `person`, namely `otherPerson`, which is now in the new `iter.previousElements`. This means the `\exists` clause is true, and thus equal to `acc`, which is what the loop invariant demands.

B) `person.age() != otherPerson.age() + 1`

In this case, `acc` will be equal to the value it had in the previous iteration.

- `(acc == \old(acc) || false) ⊢ acc == \old(acc)`

In the new state, `otherPerson` is an extra `p` that needs to be covered in the `\exists` clause of the loop invariant (it is now in `iter.previousElements`). It does not change the value of the `\exists` clause. Since `person.age() != otherPerson.age() + 1`, it is certainly not `otherPerson` which would make the existential quantification true.

- `\old((\exists Person p;
 iter.previousElements.contains(p);
 person.age() == p.age() + 1))`

`==`

```
(\exists Person p;  
  iter.previousElements.contains(p);  
  person.age() == p.age() + 1)
```

Since both the left hand side and the right hand side of the equation in the loop invariant remain unchanged, the loop invariant applies.

5.3.5. P4

At P4, we know the loop invariant and the negation of the loop condition hold. The latter implies that `iter.previousElements.equals(collection)`. This means that we can substitute `iter.previousElements` with `collection` where we want.

- `acc == (\exists Person p;
 iter.previousElements.contains(p);
 person.age() == p.age() + 1)`
- `! iter.hasNext()
 ⊢ iter.previousElements.equals(collection)`

The substitution leads us to:

- `acc == (\exists Person p; collection.contains(p);
 person.age() == p.age() + 1)`

5.3.6. Pn

A simple substitution of `acc` by `\result` then brings us to the proof of the postcondition of the method:

- `\result == (\exists Person p; collection.contains(p);
 person.age() == p.age() + 1)`

QED

5.4. Implementation Using Internal Iterators

The implementation and internal specification of a version of the implementation that uses nested internal iterators is shown in **Fig. 8**. The formal arguments are declared `final` so that they can be used in the anonymous inner classes.

```
public boolean allOneYearOlder(
    final Collection first,
    final Collection second) {
    return new ForAll() { //A

        /*@
         * @ also public behavior
         * @
         * @ post \result == (element != null) &&
         * @ (element instanceof Person);
         * @
         * public model boolean isValidElement(
         * @ Object element);
         * */

        /*@
         * @ also public behavior
         * @
         * @ post \result == (\exists Object p2;
         * @ second.contains(p2);
         * @ ((Person)element1).age() ==
         * @ ((Person)p2).age() + 1);
         * */
        public boolean criterion(
            final Object element1) { //P0
            return new Exists() { //E

                /*@
                 * @ also public behavior
                 * @
                 * @ post \result == (element != null) &&
                 * @ (element instanceof Person)
                 * public model boolean isValidElement(
                 * @ Object element);
                 * */

                /*@
                 * @ also public behavior
                 * @
                 * @ post \result ==
                 * @ ((Person)element1).age() ==
                 * @ ((Person)element2).age() + 1);
                 * */
                public boolean criterion(
                    final Object element2) {
                    return ((Person)element1).age() ==
                        ((Person)element2).age() + 1;
                }
            }
        }
    }.in(second); //Pn
}
}.in(first);
}
```

Fig. 8. Implementation of `allOneYearOlder()` using nested internal iterators

A detailed correctness proof is presented in [Dockx, van Dooren, Steegmans 2001]. Here we show a schematic proof of the outer criterion method (`criterion-A()`)⁶, for comparison with the proof

⁶ We have labeled the anonymous inner classes A and E, and add this name as an extension to the method names in the proof for easy reference

given in 5.3.1 to 5.3.6. First, we will prove that the method is correct given the inner iterator that extends `Exists` (i.e., class `E`). Secondly, we will prove that this inner iterator is correct itself.

5.4.1. P0

At program point `P0`, we know that the preconditions of `allOneYearOlder()` and `criterion-A()` are satisfied. The interesting statements are:

- `second != null;`
- `(\forall \text{Object } o; \text{second.contains}(o);`
 `o instanceof Person);`
- `! second.contains(null)`
- `isValidElement-A(element1)`

5.4.2. Pn

The method `Exists.in(Collection)` is called and the result of the method is returned.

The precondition of the method `Exists.in()` requires:

- `(\forall \text{Object } o; \text{second.contains}(o);`
 `isValidElement-E(o)`
 `-| (\forall \text{Object } o; \text{second.contains}(o);`
 `(o != null) \&\& (o instanceof Person))`
 `-| (\forall \text{Object } o; \text{second.contains}(o);`
 `(o instanceof Person)) \&\&`
 `(! second.contains(null))`

This follows immediately from `P0`.

The postcondition of the `in()` method is:

- `\result == (\exists \text{Object } o; \text{collection.contains}(o);`
 `criterion(o));`
 `-| \result ==`
 `(\exists \text{Object } o; \text{collection.contains}(o);`
 `(((Person)element1).age() ==`
 `((Person)o).age() + 1))`

The `in()` method is applied to `second`, so we get the postcondition specified for `criterion-A()`.

5.4.3. Inner Class Exists

The only implementation that is added in the anonymous inner class that derives from `Exists` is the body of the most inner `criterion()` method, a.k.a. `criterion-E()`. Therefore, we need to prove that it is correct. The correctness of the inherited code is also inherited.

The proof of the postcondition of the method is trivial, since the code is verbatim the same as the postcondition.

The last thing we need to check is whether we did not violate any preconditions or triggered any exceptions in the implementation. The addition cannot give an overflow, because of the type invariants of `Person`. What we are left with is the possibility of a `ClassCastException` in the down casts, or a `NullPointerException`. Both are impossible because of the precondition of the method.

- `isValidElement-E(element2)`
 `-| (element2 != null) \&\& (element2 instanceof Person)`

QED

5.5. Comparison

It should be clear from the above example that the reduction in complexity of the code, and of the assertions that make explicit our reasoning about that code, is enormous when internal iterators are used. The reduction in complexity we get from encapsulating the inner loop in a helper method is big, but the reduction in complexity we get by completely annihilating the iteration code is phenomenal.

6. Where Did All My Complexity Go?

In computer science, we have Laws of Conservation just like in physics. The basic law of conservation in computer science is in our experience the Law of *Conservation of Misery*. In the current context, this

can be read as *Conservation of Complexity*. If complexity is reduced in one place, it is sure to rear its ugly head somewhere else. So where did the complexity that we claim disappeared from applications, go?

It is obvious that, if methods get shorter, you need more of them to get the same net effect. So the complexity has moved from intra-method complexity to interaction of different methods and the management of the vast amount of methods that appear in object oriented programs. This complexity however can be managed better, we contend, in object oriented programs, where they are gathered in classes. This reasoning probably already applies to object based programs. The complexity in method interaction is visible in the contract of the interacting methods. Here, the class concept and the instance concept enables us to separate *type invariants* out of the general contract of the methods. The type invariants function as *anchor points* in the reasoning about the methods and instances of the class. This approach, with developers actively trying to design solid anchor points, proves extremely powerful in reducing the overall complexity of the class. The extra layer of abstraction which a class represents makes the methods easier to understand, because all the methods in a class dance to a common tune. Whatever complexity there is left, is well encapsulated, like a lemma.

If selection structures are replaced by dynamic binding, we get much more classes in an application. Again, the complexity has moved from intra-method complexity to interaction of different methods and the management of the larger number of classes. Added is the complexity of the interaction between the same method in superclasses and subclasses. Good design is governed here by the *Liskov Substitution Principle* [Liskov, Wing 1994]. The contract of the superclass functions as an *anchor point* for the diversity of subclass behavior. Whatever complexity there is left is also well encapsulated.

If iteration structures are replaced by internal iterators, again, we get more classes in the application, and the complexity moves from intra-method complexity to object and class interaction and the management of the extra classes. Whatever complexity there is left is again well encapsulated.

In general, we can say that with object oriented programming techniques, the complexity of software has moved from the *time domain*, out of the algorithms, into a more *spatial domain*. In traditional imperative programs, the complexity we as humans seem to have most difficulties with in reasoning about the algorithm, is the temporal complexity of the evolution of the states of different variables. This is even more difficult when the algorithm has selection structures or iterations. In object oriented programming this complexity is replaced by the more spatial complexity of object graphs. It seems that we are more capable to handle this kind of spatial complexity than temporal complexity. One reason might be that spatial problems offer us more possibilities for *anchor points* around which we can focus our reasoning. Another way of saying this, might be that the bulk of the complexity moved from the *implementation* (the algorithms) to the *design* (the class diagram, interaction diagram, and contract design)

In conclusion, we believe that object oriented programming as a whole is not naturally less complex than its traditional structured counterpart, but that in state-of-the-art object oriented programs the complexity is in a form which humans can master far better. The complexity is of a spatial nature, and *can* be better structured and managed through anchor points, and *can* be encapsulated better. The result is that we can create more complex programs with the same resources.

7. Conclusion

In this paper, we reminded the reader that *structured programming* is more than a rigorous self-limitation to well-known control structures. It is about being able to reason about and proving the correctness of algorithms, through making the knowledge the programmer has about the state of the computer at each point in the algorithm explicit. The intricacy of the assertions that do this is a measure for the complexity of programming.

Next, we showed that 2 carriers of this complexity, *sequence* and *selection*, have become less prominent in state-of-the-art object oriented programming. As a result, the complexity that accompanied them has disappeared from algorithms in state-of-the-art object oriented programs. A similar thing has not happened for *iteration*, although iteration is known to be the most complex control structure, and it has been phased out for this reason of, e.g. functional programming and databases.

In the third chapter, we presented *internal iterators* as the means to achieve the same effect in object oriented programs. We made clear that the importance of using internal iterators instead of explicit iterations is highly underrated by comparing the code and the correctness proof of different implementations of an example method.

We believe internal iterators represent a closure in the migration from traditional, imperative procedural programming to imperative object oriented programming. We believe internal iterators are the

last step needed to make Dijkstra's dream of *zero-defect* software through *structured programming* or *clean room programming* practically feasible. Internal iterators represent the software theorems for iterations Dijkstra dreamed about.

Finally, we presented some ideas on where the complexity that these techniques removed from the algorithms went. We believe state-of-the-art object oriented techniques shifts complexity from the *time domain*, out of the algorithms, into a more *spatial domain* of object interaction graphs and type design. We believe this kind of complexity can be handled better by developers because it provides more possibilities for *anchor points* in reasoning about the code.

8. References

- [Blakeley, Dockx, van Dooren 2001]
Peter Blakeley, Jan Dockx & Marko van Dooren: **jutil.org**
SourceForge; Labrador, Leuven; 2001; <http://org-jutil.sourceforge.net/>,
<http://www.sourceforge.net/projects/org-jutil/>, <http://www.jutil.org/>
- [Churcher, Shepperd 1995]
N.I. Churcher & M.J. Shepperd: **Towards a Conceptual Framework for OO-Software Metrics**
ACM SIGSOFT, Vol. 20 1995, p.69-76
- [Dijkstra 1968]
Edsger W. Dijkstra: **Go To statement considered harmful**
Communications of the ACM, Vol. 11, Nr. 3; 1968; p. 147-148
- [Dijkstra 1969-1970]
Edsger W. Dijkstra: **Notes on Structured Programming**
In Pursuit of Simplicity; the manuscripts of Edsger W. Dijkstra, Ed. Texas; 1969-1970;
<http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>
- [Dijkstra 1972]
Edsger W. Dijkstra: **Notes on Structured Programming**
Structured Programming, Ed. O. J. Dahl, Edsger W. Dijkstra, C. A. R. Hoare, F. Genuys; Vol. Academic Press; New York; 1972
- [Dockx, van Dooren, Steegmans 2001]
Jan Dockx, Marko van Dooren & Eric Steegmans: **Different Implementations in Java of a Nested Loop and Their Proofs**
Katholieke Universiteit Leuven, Dept. of Computer Science; Leuven; 2001; CW324;
<http://www.cs.kuleuven.ac.be/publicaties/rapporten/>
- [Gamma, Helm, et al. 1994]
Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides: **Design Patterns: Elements of Reusable Object-Oriented Software**
Addison-Wesley Pub. Co.; Wokingham, England; Reading, Mass.; 1994; ISBN 0-201-63361-2
- [Java Collections API Design FAQ 1997]
Java Collections API Design FAQ:
Sun Microsystems, Inc.; 1997;
<http://java.sun.com/j2se/1.3/docs/guide/collections/designfaq.html>
- [Leavens]
Gary T. Leavens: **JML**
<http://www.cs.iastate.edu/~leavens/JML.html>
- [Liskov, Wing 1994]
Barbara H. Liskov & Jeannette M. Wing: **A Behavioral Notion of Subtyping**
ACM Transactions on Programming Languages and Systems, Vol. 16, Nr. 6; November 1994;
p. 1811-1841
- [Meyer 1991]
Bertrand Meyer: **Design by Contract**
Advances in Object-Oriented Software Engineering, Ed. D. Mandrioli, Bertrand Meyer; Prentice Hall; Englewood Cliffs, N.J.; 1991; p. 1-50
- [Meyer 1997]
Bertrand Meyer: **Object Oriented Software Construction**; 2nd Edition
Prentice Hall; Upper Saddle River, NJ; 1997; 1254 pages; ISBN 0-13-629155-4