Specialization of Java Generic Types

Sam BeVier, Elena Machkasova
University of Minnesota, Morris

July 17, 2006

Contents

Chapter 1

Overview

1.1 Introduction

Generic types are a much-needed recent addition to Java™. They make it convenient for programmers to write polymorphic code and to leave it up to the compiler to check the type safety and to insert all necessary type casts. Java generic types are implemented using type erasure - the compiler replaces the type parameter by the upper bound of the class (either the one specified in the class declaration, or, if none is specified, by Object). This approach provides a convenient uniform representation of a generic type, but leads to inefficiency at run time.

We propose a compilation-time transformation that we call *Specialization of Type Bounds*. This optimization creates a copy of a generic type with the type bound by a more specific one than the upper bound in the original class or method declaration. We show that this optimization results in up to 56% decrease in run time of the relevant segment of the code in the client mode of the HotSpot JVM and in up to 84% time decrease for the server mode of the JVM. Using test examples, we study some factors involved in choosing the specific bound or bounds for specialization.

This is work in progress with the goal to write a source-to-source preprocessor for specializing type bounds of generic types to achieve efficiency while, if possible, minimizing the code duplication.

1.2 Specializing Type Bounds

1.2.1 Java Generics and Type Bounds

Java 1.5 introduced generic types which allow programmers to write classes or methods parameterized over type variables and then instantiate these variables to a concrete type [1]. As an example, consider a code snippet from a definition of a generic Set class. T is the type parameter. T can be used within the

definition of the generic class as a type of a method parameter, a return type of a method, or as a type of variables within the class.

```
public class Set<T> {
    private T [] elements;
    ...
    public T find(element T) { ... }
    ...
    public void add(T element) { ... }
    ...
}
```

Any instance of a generic class must supply a concrete type that would be used to instantiate T. For instance, a program may create a set of strings like this: `Set<String>`. `String` in this case is said to *instantiate* the type parameter T. The compiler checks that the only objects added to this instance of Set are strings.

Often generic classes call a method on objects of the class given by the type parameter, and thus require that all type instances define this method. For instance, one might want to implement a generic sorted list which would require that elements inserted into the list implement Comparable interface since the elements must be compared to each other. In such cases a programmer who declares the generic class specifies a class or an interface as a *bound* on the type parameter to guarantee that only classes that extend the bounding class (or implement the bounding interface) can instantiate the type parameter.

Below, the sorted list declaration specifies the bound for T to be Comparable. Note that despite Comparable being an interface, the keyword is `extends`, not `implements`.

```
public class SortedList<T extends Comparable>
```

The compiler would allow `SortedList<String>` but not `SortedList<Object>` since String implements Comparable but Object does not.

Generic types with no specified type bound (such as the Set example above) are in fact bounded by Object.

Methods can be parameterized even if the class they are in is not. In these cases the method signatures contain the type parameter (bounded or unbounded) in angle brackets much like parameterized classes do.

```
public <T extends Comparable>T min(T t1, T t2) {
    if (t1.compareTo(t2) > 0) {
        return t2;
    } else { return t1;}
}
```

Unlike generic classes which require the programmer to supply an explicit value of the parameter when creating an instance of the class, the actual types of a method's type parameters are inferred by the compiler based on the type of the

parameters in the method call. For instance, in the call `min("apple","banana")` the compiler infers T to be String.

At this point we are not considering several features of generic types, such as inheritance of generic types, wildcards, or bounded wildcards. Studying these features from the point of view of specializing type bounds is a part of our future research.

1.2.2 Type Erasure

Generic types in Java are implemented using *type erasure*: after checking type-correctness of the generic code the compiler replaces all occurrences of type parameters by their type bounds, thus creating one uniform representation of each generic class or method. Type casts are inserted to guarantee that at runtime an attempt to assign an object of a wrong type to a variable (for example, as a return of a method) results in a ClassCastException. The resulting bytecode has no information about specific type parameters used to instantiate generic classes or methods. The type information is said to be *erased*.

Type erasure provides a simple uniform representation of a generic type, avoiding creation of extra classes. However, this implementation lacks efficiency. There are two sources of overhead that result from type erasure: type casts and the inability of JVM to perform type-specific optimizations based on the actual type parameter. For example, the SortedList above will be implemented as a sorted list of Comparable objects, even though it is used with strings. This means that every time an object is returned, say, by a `getNext()` method, it is returned as Comparable, and a type cast is needed to convert it to a string. This also means that the bytecode implementation of SortedList uses Comparables, and therefore every call to `compareTo` is made via `invokeinterface`, even though all of these calls are in fact on strings. Had the precise type information been passed to the JVM, there would have been an opportunity to devirtualize or even inline the `compareTo` method. Devirtualization is an optimization that can be performed by the JVM when it knows which version of a method to use and where on a hierarchy to find its definition so it doesn't have to do a runtime lookup of the method and can just skip straight to the correct definition [3]. Inlining is when the JVM replaces a method call with the actual code of that method, avoiding the overhead of a method call. So if type information is available at runtime, then devirtualization is possible. Since the exact method is known, inlining may be performed as well.

1.2.3 Specializing the Type Bounds

We propose and study an optimization that removes the overhead of type erasure where it is beneficial to do so. We present it as a source to source transformation, although incorporating it as a part of source to bytecode compilation is also possible. However, the optimization must be done on the source code, not on the bytecode, since by the time the program is compiled to bytecode concrete type information is "erased".

The essence of the optimization is to create a copy of a generic type with the type bound replaced by the actual instance of the type used in the program. For instance, in the sorted list example above we would create a specialized copy of the list class with the String bound:

```
public class SortedList_Str<T extends String>
```

We then replace all uses of `SortedList<String>` by the new specialized list: `SortedList_Str<String>`[1]. This eliminates the need to type cast return values of the list methods and allows for the possibility of type-specific dynamic optimizations by the JVM, such as devirtualization and inlining. Note that it is important to create a separate optimized copy of the generic class or method rather than to modify the original implementation since it is possible that the compiled generic code would be used by other programs where the type parameters would be instantiated with different actual types, not the ones for which the specialization was performed.

In the next section we present test examples (mini-benchmarks) that demonstrate the benefits of the optimization and examine the best opportunities for the optimization with a simple class hierarchy. We also study different ways of optimizing code that follows a well-known *visitor design pattern.*

[1]Note that we could, alternatively, remove the type parameter from the SortedList class altogether and just turn it into a sorted list of Strings, but replacing the bound amounts to the same thing because the type erasure would replace T by String throughout the implementation, and it is easier to replace just the type bound, not every occurrence of the type.

Chapter 2

Code Examples: Mini-benchmarks

2.1 Testing Methodology

2.1.1 Test Setup

We tested our optimizations on short simple programs that make a heavy use of generics, with a goal of measuring maximal possible efficiency of the optimization and studying what factors affect this efficiency.

We ran two groups of test cases: those of a generic method and those of a generic class. We used a simple hierarchy of four classes for our tests shown in Figure 2.1.1: class B extends class A and classes C and D extend class B. We also tested cases when A is an interface and B implements A. The method called `method` is defined in the class A. We experimented with overwriting the method in all three (B,C, and D), or in B only.

Note that while the class hierarchy is the same for all our examples, the actual classes (or interfaces in some cases) differ. In the following pages we

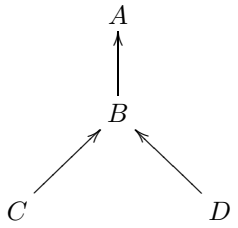

Figure 2.1: The class hierarchy used in examples: B extends A, C and D extend B

provide the actual code that we tested, results, and brief conclusions.

In the test cases for a generic method we used a method

```
public static <T extends A> void m(T [] array) {
    for (int i = 0; i < array.length; i++) {
        array[i].method();
    }
}
```

We then used two arrays of 10000 elements each, `carray` of a type C and `darray` of a type D, and called the method `m` on elements of each of the two arrays 1000 times:

```
    for (int i = 0; i < 1000; i++) {
        m(carray);
        m(darray);
    }
```

For each test case we compared running times for three versions of the generic method: one with the bound A, one with the bound B, and another one with two separate copies of the method with bounds C and D, respectively.

Another series of test examples deals with a generic sorted linked list: a data structure that stores elements in the non-decreasing order. While in real-life such data structures would probably require that the elements implement Comparable interface, we studied this example with the same 4-class hierarchy as in the method examples. The class A defined a `compare` method (to avoid confusion, we named it differently from the `compareTo` method required by the Comparable interface). To keep the list insertion's running time uniform when inserting objects into the list, the method just performs a comparison of two fixed strings.

Similarly to the test cases for a generic method, we created two linked lists, one with the type parameter C and the other one with D:

```
SortedLinkedList<C> c = new SortedLinkedList<C>();
SortedLinkedList<D> d = new SortedLinkedList<D>();
```

Then we inserted 4000 elements into each sorted list. Inserting n elements into a sorted list takes $\frac{n^2}{2}$ comparisons (since we needed to maximize the number of comparisons to measure the running time more accurately, we set the `compare` method in such a way that each new inserted element is compared to each element already in the list). Hence the total number of calls to `compare` is 8,000,000 per list. As for the generic method case, we ran three versions of the code: one with the bound A, one with the bound B, and another one with two separate sorted linked list classes with bounds C and D.

All tests were run on the same machine running Fedora Core, with version 1.5.0_04 of both the Sun's Java HotSpot$^{\text{TM}}$VM and the javac compiler.

There are two modes for running the HotSpot VM, the client and the server modes. Both modes optimize the program at run time using profiling to select

frequently called methods for dynamic optimization and possibly for compilation to native code. The client mode is more memory efficient and is intended to get a program running as quickly as possible without performing time-consuming optimizations. The server mode focuses on better optimizing the program, even if that requires more time and memory. Each of our test programs was run in each of the client and server mode of the HotSpot JVM 20 times for each of the three type bounds.

We recorded the runtime of test programs in several ways. We used the function `System.currentTimeMillis()` to time the loop that calls the generic method or inserts elements into the list (we refer to this time as the *loop time*). We also timed the execution of the JVM with the Unix time command, which includes the *user* CPU time, i.e. the total time that the process spent in the user mode and the *real (total elapsed)* time that the process took.

We found out that the difference between the user time and the loop time for all our test runs seems to depend only on the mode of the JVM but not on a specific program or the number of times the loop runs (for instance, increasing the loop 10 times did not seem to make a difference). This difference is 0.03-0.07 sec. for the client mode and 0.07-0.1 sec.for the server mode. The source of the overhead over the loop time is likely to be class loading and startup, creating the objects before the loop starts, garbage collection, and run-time profiling and management (the latter time seems to be small since the overhead does not increase when the loop executes more times). Since the difference is quite consistent we only show the loop times for our test cases since they allow us to better compare efficiency for different choices of bounds.

The additional overhead of the total (elapsed) time over the user time is quite inconsistent and is anywhere between 0.01 and 0.15 sec for runs of the same program in the same mode. This may be due to other processes running at the same time. We are not including the total time in our results since its variation does not depend on the optimizations we are testing.

The results for mini-benchmarks are summarized in Figure 2.2. The table shows efficiency increase when a bound of a parameterized class or a method is replaced by a more precise bound. "A $\rightarrow$ B" marks the change from A bound to B bound, "B $\rightarrow$ C/D" marks the change from B to two separate bounds for C and D. The table does not include the typecasting tests or any of the additional tests.

2.1.2 Graph Notations

For visual representation of our data we use R, a statistical analysis program. All of the graphs we use are box plots, otherwise known as box and whiskers diagrams. This is a simple way to show many aspects of each data set in a concise manner. The bold line through the middle of the box is the set's median. Each edge of the box(upper and lower quartile) represents the 25 percent of the data above and 25 percent below the median, respectively. The arms of the plot represent the values furthest away from the median while still remaining within a certain range from the lower and upper edges of the box. This range is known

	Client		Server	
Test Cases	A → B	B → C/D	A → B	B → C/D
Parameterized Method; A is a class				
NonOverridden Empty	0.57	-0.04	0.87	-0.18
NonOverridden NonEmpty	0.23	-0.01	0.42	-0.02
Overridden Empty	-0.01	0.56	-0.02	0.84
Overridden NonEmpty	0	0.19	-0.01	0.35
Parameterized Method; A is an interface				
NonOverridden NonEmpty	0.24	0	0.54	-0.03
Overridden NonEmpty	0.12	0.22	0.17	0.34
Parameterized Class; A is a class				
NonOverridden NonEmpty	0.21	-0.01	0.32	-0.11
Overridden NonEmpty	0	0.2	0.05	0.36
Parameterized Class; A is an interface				
NonOverridden NonEmpty	0.26	-0.01	0.49	-0.11
Overridden NonEmpty	0.13	0.19	0.2	0.36

Figure 2.2: Percentage efficiency increase

as the interquartile range (IQR), and is equal to the width of the box (upper quartile - lower quartile). Any points past the arms to either side are considered to be outliers and are represented in R as small circles. Although outliers are still shown in the plot, they are not considered to be indicative of the set, and are all but ignored.

Each box plot in a graph has its own label. That label consists of either "Client" or "Server" and the bound which is used during that run. e.g. "Client-C/D" means that it was the client run of the code using the bounds C and D. In the typecasting example it works a little differently. The JVM mode is shown by "Cl" or "S", then the bound is "A","B",or "C", and the "A" or "NA" at the end denotes whether or not the result of the method was assigned to a variable or not. e.g. "S-B-NA" means that it was a server run of a B bound where the result was not assigned.

2.2 Parameterized Method Examples

These are the examples dealing with parameterized methods. The method `m` in the classes TestA, TEstB, and TestCD is parameterized with different bounds (in TestCD case there are actually two different methods with bounds C and D respectively). This method takes an array of the type given by the type parameter and calls the method `method` on each element of the array. There are two calls to `m` in the testing code: one with an array of C elements, the other one with an array of D elements.

Our test answer the following questions:

- whether the keyword `final` has any effect on the optimization.
- which bound is the best if the method `method` is defined in A and B and is further overridden in C and D?
- which bound is the best if the method `method` is defined in A and B and is `not` overridden in C and D?
- the above two questions for the case when A is an interface.
- we also ran some of the above tests with an empty method. However, only the actual numbers differed, but the qualitative results were the same, so we stopped using the empty methods to cut down on the number of tests.

Note that the code for classes A, B, C, and D changes from one test to another (for instance, when we replace a non-empty method by an empty one), but the testing code does not change.

Below is the code for the three testing classes:

```
public class TestA {
    public static <T extends A> void m(T [] array) {
        for (int i = 0; i < array.length; i++) {
            array[i].method();
```

```
        }
    }
    public static void main(String [] args) {
        int n = 10000;
        int m = 1000;
        C [] carray = new C[n];
        D [] darray = new D[n];

        for (int i = 0; i < n; i++) {
            carray[i] = new C();
            darray[i] = new D();
        }

        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int i = 0; i < m; i++) {
            m(carray);
            m(darray);
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-A\t" + (time2-time1));
    }
}
-----------------------------------------------------------------
public class TestB {
    public static <T extends B> void m(T [] array) {
        for (int i = 0; i < array.length; i++) {
            array[i].method();
        }
    }
    public static void main(String [] args) {
        int n = 10000;
        int m = 1000;
        C [] carray = new C[n];
        D [] darray = new D[n];

        for (int i = 0; i < n; i++) {
            carray[i] = new C();
            darray[i] = new D();
        }

        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int i = 0; i < m; i++) {
```

```
            m(carray);
            m(darray);
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-B\t" + (time2-time1));
    }
}
-------------------------------------------------------------
public class TestCD {
    public static <T extends C> void m(T [] array) {
        for (int i = 0; i < array.length; i++) {
            array[i].method();
        }
    }
    public static <T extends D> void m(T [] array) {
        for (int i = 0; i < array.length; i++) {
            array[i].method();
        }
    }
    public static void main(String [] args) {
        int n = 10000;
        int m = 1000;
        C [] carray = new C[n];
        D [] darray = new D[n];

        for (int i = 0; i < n; i++) {
            carray[i] = new C();
            darray[i] = new D();
        }

        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int i = 0; i < m; i++) {
            m(carray);
            m(darray);
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-C/D\t" + (time2-time1));
    }
}
```

2.2.1 Test Case for Keyword “final”

This is the final test case where we test the effect of the ”final” Java keyword on generic optimizations. Here is the A,B,C, and D code for the MethodEmpty

test case. Code:

```
public class A {
    protected int item;
    public A() {
        item = 5;
    }
    public void method() {
    }
}
-------------------------------------
public class B extends A {
    protected int item2;
    public B() {
        super();
        item2 = 6;
    }
    public void method() {
    }
}
-------------------------------------
public final class C extends B {
    private String Cstring;
    public C() {
        super();
        Cstring = "C";
    }
    public final void method() {
    }
}
-------------------------------------
public final class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
    public final void method() {
    }
}
```

The graph corresponding to this example is Figure 2.3. The results for this, when compared to the results for the "Method overwritten empty" example (Figure 2.4), show that the `final` keyword does not increase effectiveness of optimizations.

The next test case we had was the same case, except the methods were not empty. The A, B, C, and D code for this case is:

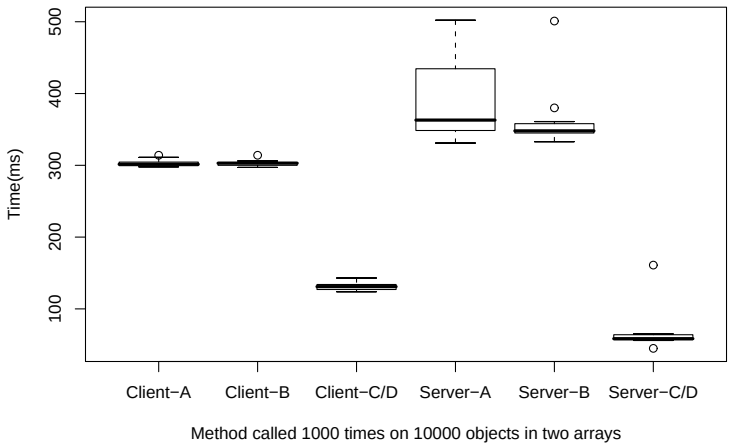

Figure 2.3: Final keyword test case with an empty method

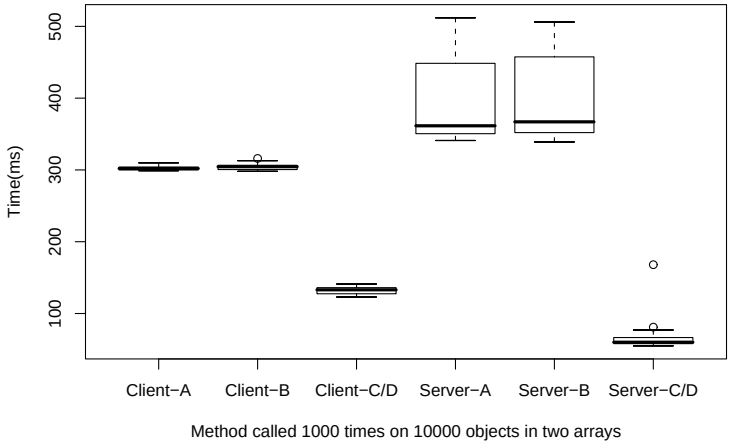

Figure 2.4: Parameterized method overridden in C and D

```
public class A {
    protected int item;
    public A() {
        item = 5;
    }
    public void method() {
        item--;
    }
}
-----------------------------------
public class B extends A {
    protected int item2;
    public B() {
        super();
        item2 = 6;
    }
    public void method() {
        item2 = "hi".compareTo("Bye");
    }
}
-----------------------------------
public final class C extends B {
    private String Cstring;
    public C() {
        super();
        Cstring = "C";
    }
    public final void method() {
        item2 = Cstring.compareTo("Hi");
    }
}
-----------------------------------
public final class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
    public final void method() {
        item2 = Dstring.compareTo("Bye");
    }
}
```

The graph for this case is Figure 2.5.

We discovered that both empty and non-empty methods were optimized in the case of the more precise bound. The difference for empty methods was more

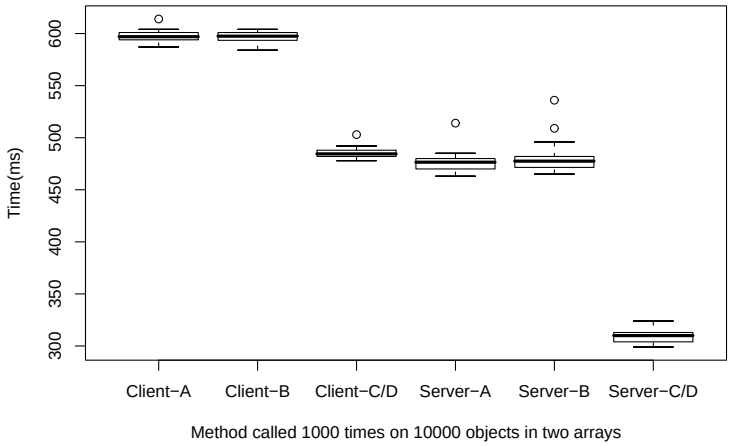

Figure 2.5: Final keyword test case with a non-empty method

noticeable than for non-empty ones because the overhead of the method call was larger proportionally to the total running time of the test.

2.2.2 Method Overridden vs. Not Overridden case

This is where we test the efficiency of the optimization when methods are overridden in the subclasses vs. when they are not. For each of these two cases, we test with a case for an empty method, and one for when the method is not empty. Below is the code for the case when the method is empty and is not overridden in C and D, only in B.

```
public class A {
    protected int item;
    public A() {
        item = 5;
    }
    public void method() {
    }
}
-------------------------------------
public class B extends A {
    protected int item2;
    public B() {
        super();
        item2 = 6;
    }
```

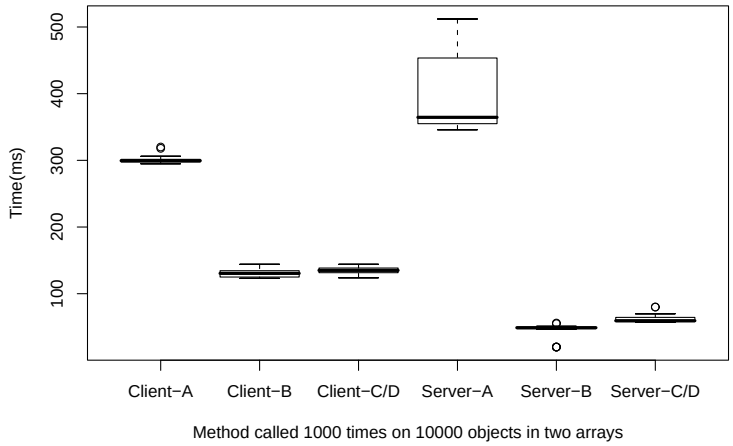

Figure 2.6: Parameterized method; method() not overridden in C and D

```
    public void method() {
    }
}
-------------------------------------
public class C extends B {
    private String Cstring;
    public C() {
        super();
        Cstring = "C";
    }
}
-------------------------------------
public class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
}
```

The corresponding graph is Figure 2.6.

The next test case we have is when the method is not overridden and the methods are not empty. The code for this case is exactly the same as the last, except the methods are not empty. The graph that corresponds to this test case is Figure 2.7. Here is the code:

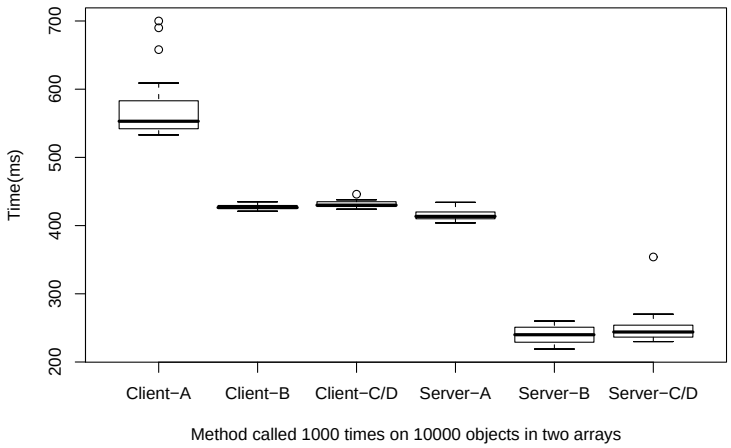

Figure 2.7: Parameterized method; method() not overridden in C and D

```
public class A {
    protected int item;
    public A() {
        item = 5;
    }
    public void method() {
        item--;
    }
}
---------------------------------
public class B extends A {
    protected int item2;
    public B() {
        super();
        item2 = 6;
    }
    public void method() {
        item2 = "hi".compareTo("Bye");
    }
}
---------------------------------
public class C extends B {
    private String Cstring;
    public C() {
        super();
```

```
        Cstring = "C";
    }
}
-----------------------------------
public class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
}
```

The test cases that deal with method overwriting have shown us that the best type bound is the class in which the method is defined. For instance, if the method was not overridden in C and D, any call to that method on an object of C or D is calling the B defined method. So the runs that parameterize the bounds to B run just as fast as the ones bound to C and D, even faster by a little bit because of the overhead for the method to be looked up one level. Therefore it is more beneficial to parameterize the method to B because it runs at least as fast and only uses one instantiation to hold all the elements, not two.

The next two cases are the same as the last two, except we are testing empty and non-empty methods when they are overridden in C and D. The test case where the method is empty and overridden is represented by Figure 2.4 and the code is here:

```
public class A {
    protected int item;
    public A() {
        item = 5;
    }
    public void method() {
    }
}
-----------------------------------
public class B extends A {
    protected int item2;
    public B() {
        super();
        item2 = 6;
    }
    public void method() {
    }
}
-----------------------------------
public class C extends B {
    private String Cstring;
    public C() {
```

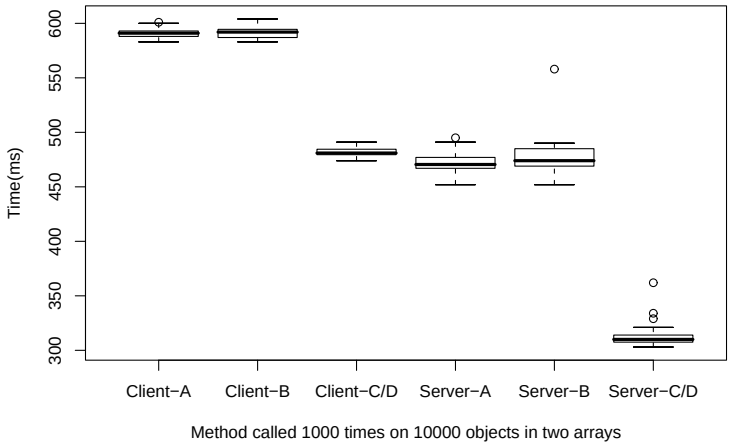

Figure 2.8: Parameterized method overridden in C and D

```
        super();
        Cstring = "C";
    }
    public void method() {
    }
}
---------------------------------
public class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
    public void method() {
    }
}
```

The case with a nonempty method is shown in Figure 2.8 and its code is here:

```
public class A {
    protected int item;
    public A() {
        item = 5;
    }
    public void method() {
        item--;
```

```
    }
}
---------------------------------
public class B extends A {
    protected int item2;
    public B() {
        super();
        item2 = 6;
    }
    public void method() {
        item2 = "hi".compareTo("Bye");
    }
}
---------------------------------
public class C extends B {
    private String Cstring;
    public C() {
        super();
        Cstring = "C";
    }
    public void method() {
        item2 = Cstring.compareTo("Hi");
    }
}
---------------------------------
public class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
    public void method() {
        item2 = Dstring.compareTo("Bye");
    }
}
```

The overridden cases show that in this case, the efficiency is highest when the bounds are set to C and D (because that is where the actual method that gets called is defined), but the B bound time is about the same as the A bound time. That means that the method lookup from A to C/D is about the same as from B to C/D, so in this case it is definitely worth it to create two implementations for the lowest bound (C and D) than to only create one (bound B)[1].

[1]However, this may change if the method returns; in this case typecasting may play a role. See section 2.4 for related tests

2.2.3 Method interfaces example

In this test case we test whether changing A to be an interface affects the optimizations happening when the method is overridden or not in C and D. Here is the code for the Not Overridden test case and the graph is Figure 2.10:

```
public interface AIF {
    public void method();
}
-------------------------------------------------------
public class B implements AIF {
    protected int item;
    protected int item2;
    public B() {
        item = 5;
        item2 = 6;
    }
    public void method() {
        item2 = "Hi".compareTo("Bye");
    }
}
-------------------------------------------------------
public class C extends B {
    private String Cstring;
    public C() {
        super();
        Cstring = "C";
    }
}
-------------------------------------------------------
public class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
}
```

The graph for the Overridden test case is Figure 2.9 and the code is here:

```
public interface AIF {
    public void method();
}
-------------------------------------------------------
public class B implements AIF {
    protected int item;
    protected int item2;
```

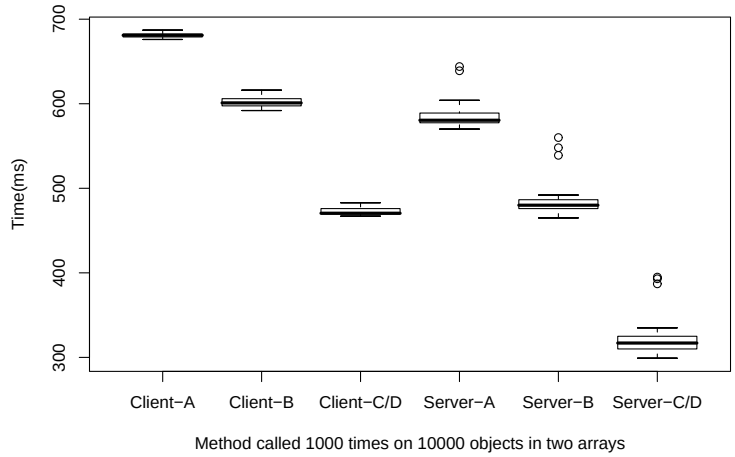

Figure 2.9: Parameterized method; method() overridden in C and D. A is an interface.

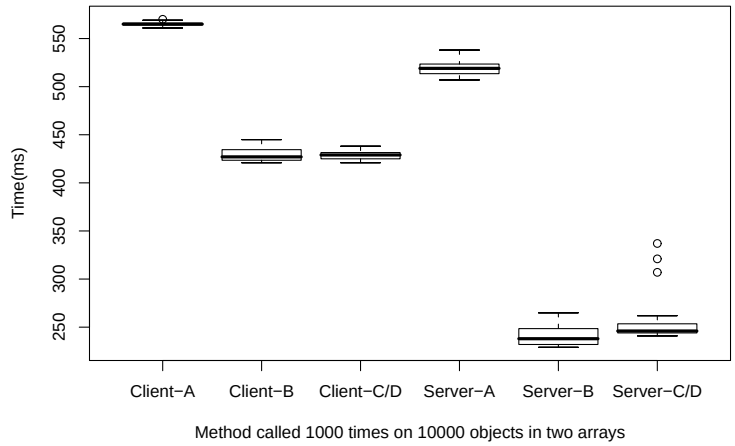

Figure 2.10: Parameterized method; method() is not overridden in C and D. A is an interface.

```
    public B() {
        item = 5;
        item2 = 6;
    }
    public void method() {
        item2 = "Hi".compareTo("Bye");
    }
}
-------------------------------------------------------
public class C extends B {
    private String Cstring;
    public C() {
        super();
        Cstring = "C";
    }
    public void method() {
        item2 = Cstring.compareTo("Hi");
    }
}
-------------------------------------------------------
public class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
    public void method() {
        item2 = Dstring.compareTo("Bye");
    }
}
```

The interfaces example showed that the interface bound creates a substantial overhead over a class bound. Notice that the graphs look the same as the ones for the case of an overridden method (Figure 2.8) and the nonoverridden method (Figure 2.7) except that there is an extra runtime difference between the A and B bound when the method is not overridden in C/D.

2.3 Parameterized Class Examples

The test cases in this section are very similar to those of the method examples section, except we are dealing with parameterized classes, not just methods. We use a data structure (a sorted linked list) which stores elements in an increasing order. A new inserted element is compared to elements already in the list until the right place for it is found.

We have 4 test cases in this section:

- the method is overridden vs not overridden if A is a class,
- the method is overridden vs not overridden if A is an interface.

The only code that changes between these examples is in the A,B,C,D classes.

Note that we are not testing empty methods at this point since they seem to be optimized similarly to non-empty ones.

Below is the code for the sorted linked list and the testing code that creates the list and inserts and removes the elements. This code is the same for all test cases.

```
public class SortedLinkedListA<T extends A>
{
    Node root;
    SortedLinkedListA(){}

    public void insert(T value)
    {
        if(root==null){
            root = new Node(value, null);
            return;
        }

        if(value.compare(root.getValue()) <= 0){
            root = new Node(value, root);
            return;
        }

        Node node = root;
        while(node != null){
            Node nextNode = node.getNext();
            if(nextNode == null || value.compare(nextNode.getValue()) <= 0){
                node.setNext(new Node(value, nextNode));
                return;
            }
            node = nextNode;
        }
    }

    public void remove(T value)
    {
        if(root==null) return;

        if((root.getValue()).compare(value) == 0){
            root = root.getNext();
            return;
        }
```

```
    Node node = root;
    while(node != null){
        Node nextNode = node.getNext();
        if(nextNode!=null && (nextNode.getValue()).compare(value) == 0){
            node.setNext(nextNode.getNext());
            return;
        }
        node = nextNode;
    }
}

public String toString()
{
    String str = "[ ";
    for(Node node = root; node!=null;
        node = node.getNext()){
        str += node.getValue() + " ";
    }

    return str + "]";
}

private class Node
{
    T value;
    Node next;

    Node(T value, Node next)
    {
        this.value = value;
        this.next = next;
    }

    public T getValue()
    {
        return value;
    }

    public Node getNext()
    {
         return next;
    }

    public void setNext(Node node)
    {
```

```
            next = node;
        }
    }
}
-------------------------------------------------------------------------
public class SortedLinkedListB<T extends B>
{
    Node root;
    SortedLinkedListB(){}

    public void insert(T value)
    {
        if(root==null){
            root = new Node(value, null);
            return;
        }

        if(value.compare(root.getValue()) <= 0){
            root = new Node(value, root);
            return;
        }

        Node node = root;
        while(node != null){
            Node nextNode = node.getNext();
            if(nextNode == null || value.compare(nextNode.getValue()) <= 0){
                node.setNext(new Node(value, nextNode));
                return;
            }
            node = nextNode;
        }
    }

    public void remove(T value)
    {
        if(root==null) return;

        if((root.getValue()).compare(value) == 0){
            root = root.getNext();
            return;
        }

        Node node = root;
        while(node != null){
            Node nextNode = node.getNext();
            if(nextNode!=null && (nextNode.getValue()).compare(value) == 0){
```

```
                node.setNext(nextNode.getNext());
                return;
            }
            node = nextNode;
        }
    }

    public String toString()
    {
        String str = "[ ";
        for(Node node = root; node!=null;
            node = node.getNext()){
            str += node.getValue() + " ";
        }

        return str + "]";
    }

    private class Node
    {
        T value;
        Node next;

        Node(T value, Node next)
        {
            this.value = value;
            this.next = next;
        }

        public T getValue()
        {
            return value;
        }

        public Node getNext()
        {
            return next;
        }

        public void setNext(Node node)
        {
            next = node;
        }
    }
}
```

```
public class SortedLinkedListC<T extends C>
{
    Node root;
    SortedLinkedListC(){}

    public void insert(T value)
    {
        if(root==null){
            root = new Node(value, null);
            return;
        }

        if(value.compare(root.getValue()) <= 0){
            root = new Node(value, root);
            return;
        }

        Node node = root;
        while(node != null){
            Node nextNode = node.getNext();
            if(nextNode == null || value.compare(nextNode.getValue()) <= 0){
                node.setNext(new Node(value, nextNode));
                return;
            }
            node = nextNode;
        }
    }

    public void remove(T value)
    {
        if(root==null) return;

        if((root.getValue()).compare(value) == 0){
            root = root.getNext();
            return;
        }

        Node node = root;
        while(node != null){
            Node nextNode = node.getNext();
            if(nextNode!=null && (nextNode.getValue()).compare(value) == 0){
                node.setNext(nextNode.getNext());
                return;
            }
            node = nextNode;
        }
```

```
    }

    public String toString()
    {
        String str = "[ ";
        for(Node node = root; node!=null;
            node = node.getNext()){
            str += node.getValue() + " ";
        }

        return str + "]";
    }

    private class Node
    {
        T value;
        Node next;

        Node(T value, Node next)
        {
            this.value = value;
            this.next = next;
        }

        public T getValue()
        {
            return value;
        }

        public Node getNext()
        {
            return next;
        }

        public void setNext(Node node)
        {
            next = node;
        }
    }
}
```

```
public class SortedLinkedListD<T extends D>
{
    Node root;
    SortedLinkedListD(){}
```

```
public void insert(T value)
{
    if(root==null){
        root = new Node(value, null);
        return;
    }

    if(value.compare(root.getValue()) <= 0){
        root = new Node(value, root);
        return;
    }

    Node node = root;
    while(node != null){
        Node nextNode = node.getNext();
        if(nextNode == null || value.compare(nextNode.getValue()) <= 0){
            node.setNext(new Node(value, nextNode));
            return;
        }
        node = nextNode;
    }
}

public void remove(T value)
{
    if(root==null) return;

    if((root.getValue()).compare(value) == 0){
        root = root.getNext();
        return;
    }

    Node node = root;
    while(node != null){
        Node nextNode = node.getNext();
        if(nextNode!=null && (nextNode.getValue()).compare(value) == 0){
            node.setNext(nextNode.getNext());
            return;
        }
        node = nextNode;
    }
}

public String toString()
{
    String str = "[ ";
```

```
        for(Node node = root; node!=null;
            node = node.getNext()){
            str += node.getValue() + " ";
        }

        return str + "]";
    }

    private class Node
    {
        T value;
        Node next;

        Node(T value, Node next)
        {
            this.value = value;
            this.next = next;
        }

        public T getValue()
        {
            return value;
        }

        public Node getNext()
        {
            return next;
        }

        public void setNext(Node node)
        {
            next = node;
        }
    }
}
-----------------------------------------------------------------------
public class TestA {
    public static void main(String [] args) {
        int n = 4000;
        SortedLinkedListA<C> Clist = new SortedLinkedListA<C>();
        SortedLinkedListA<D> Dlist = new SortedLinkedListA<D>();
        C[] carray = new C[n];
        D[] darray = new D[n];
        for (int i = 0; i < n; i++) {
            carray[i] = new C();
            darray[i] = new D();
```

```
        }
        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int i = 0; i < n; i++) {
            Clist.insert(carray[i]);
            Dlist.insert(darray[i]);
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-A\t" + (time2-time1));
    }
}
-----------------------------------------------------------------------
public class TestB {
    public static void main(String [] args) {
        int n = 4000;
        SortedLinkedListB<C> Clist = new SortedLinkedListB<C>();
        SortedLinkedListB<D> Dlist = new SortedLinkedListB<D>();
        C[] carray = new C[n];
        D[] darray = new D[n];
        for (int i = 0; i < n; i++) {
            carray[i] = new C();
            darray[i] = new D();
        }
        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int i = 0; i < n; i++) {
            Clist.insert(carray[i]);
            Dlist.insert(darray[i]);
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-B\t" + (time2-time1));
    }
}
-----------------------------------------------------------------------
public class TestCD {
    public static void main(String [] args) {
        int n = 4000;
        SortedLinkedListC<C> Clist = new SortedLinkedListC<C>();
        SortedLinkedListD<D> Dlist = new SortedLinkedListD<D>();
        C[] carray = new C[n];
        D[] darray = new D[n];
        for (int i = 0; i < n; i++) {
            carray[i] = new C();
            darray[i] = new D();
```

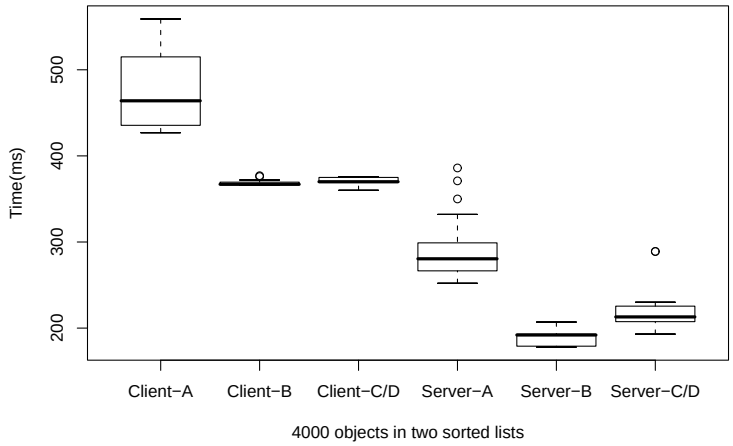

Figure 2.11: Parameterized class; method compare() not overridden in C and D

```
        }
        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int i = 0; i < n; i++) {
            Clist.insert(carray[i]);
            Dlist.insert(darray[i]);
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-C/D\t" + (time2-time1));
    }
}
```

2.3.1 A is a Class

These test cases are to test results when using a parameterized class, if A is a class, not an interface, and with the method overridden in C/D or not. The first case is when the method `compare()` is not overridden. The graph is Figure 2.11 and the code is below:

```
public class A {
    protected int item;
    public A() {
        item = 5;
    }
    public int compare(A obj) {
```

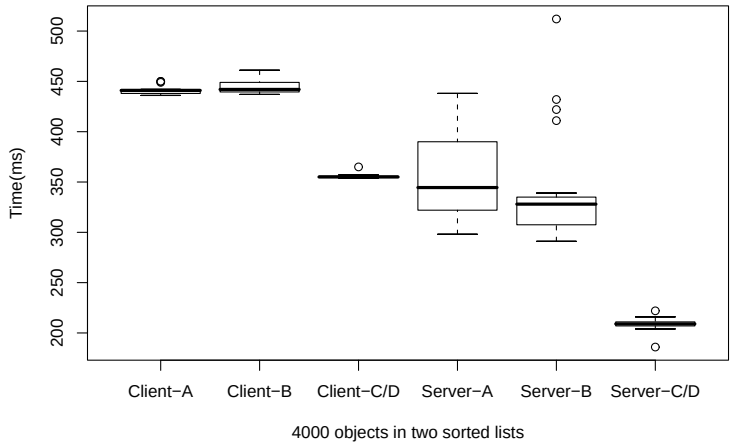

Figure 2.12: Parameterized class; method compare() overridden in C and D

```
        return "Hi".compareTo("Bye");
    }
}
-----------------------------------
public class B extends A {
    protected int item2;
    public B() {
        super();
        item2 = 6;
    }
    public int compare(A obj) {
        return "Hi".compareTo("Bye");
    }
}
-----------------------------------
public class C extends B {
    private String Cstring;
    public C() {
        super();
        Cstring = "C";
    }
}
-----------------------------------
public class D extends B {
    private String Dstring;
```

```
    public D() {
        super();
        Dstring = "D";
    }
}
```

The graph for when A is a class and the method is overridden is Figure 2.12. The code is below:

```
public class A {
    protected int item;
    public A() {
        item = 5;
    }
    public int compare(A obj) {
        return "Hi".compareTo("Bye");
    }
}
-----------------------------------
public class B extends A {
    protected int item2;
    public B() {
        super();
        item2 = 6;
    }
    public int compare(A obj) {
        return "Hi".compareTo("Bye");
    }
}
-----------------------------------
public class C extends B {
    private String Cstring;
    public C() {
        super();
        Cstring = "C";
    }
    public int compare(A obj) {
        return Cstring.compareTo("A");
    }
}
-----------------------------------
public class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
```

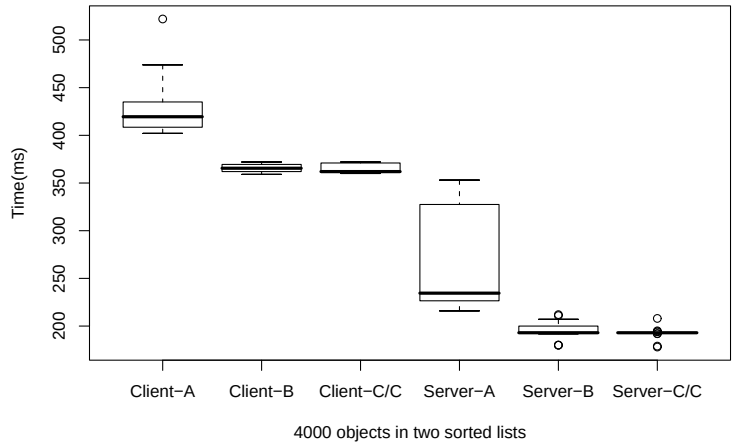

Figure 2.13: Parameterized class; compare() is not overridden in C and D

```
    public int compare(A obj) {
        return Dstring.compareTo("A");
    }
}
```

The results for these examples are very similar to those of the method cases. If the method is overridden in C and D, then the fastest time is going to be C and D and B is going to run about the same speed as A. If the method is not overridden in C and D, then A runs the slowest, and B and C/D bounds run at the same speed, with B running even a hair faster than C/D.

2.3.2 Additional Test Case: One Subclass vs. Two

We had one side test case that we had to check. We wanted to know if there was a speed difference between lookups down two children (C and D) and if there was just one (C). We created a test case where, in the testing code, every D was replaced with a C, so that there were twice as many Cs in that case. Then we ran code to see if there was a difference between BCD or BCC. The code that we replaced all the Ds in was a copy of the parameterized class example where A was a class and the method was not overridden. The graph is Figure 2.13 and the only difference in the code is in the TestCD.java file:

```
public class TestCD {
    public static void main(String [] args) {
        int n = 4000;
        SortedLinkedListC<C> Clist1 = new SortedLinkedListC<C>();
```

```
        SortedLinkedListC<C> Clist2 = new SortedLinkedListC<C>();
        C[] carray1 = new C[n];
        C[] carray2 = new C[n];
        for (int i = 0; i < n; i++) {
            carray1[i] = new C();
            carray2[i] = new C();
        }
        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int i = 0; i < n; i++) {
            Clist1.insert(carray1[i]);
            Clist2.insert(carray2[i]);
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-C/C\t" + (time2-time1));
    }
}
```

The results for this example and the SortedLinkedList test with the non-overridden method are about the same except that the C/C bound in this one is slightly faster than the C/D bound in the example with two classes C and D. It looks like it runs at the same speed as the B bound, instead of the slightly slower than B bound that it runs otherwise. This seems to indicate that the little bit of overhead of C/D bounds over B bounds in the example with C/D bounds was due to the lookup of having two children being used. With only one, B bound and C bound run the same.

2.3.3 A is an Interface

These test cases are the same as the nonoverridden/overridden examples with a parameterized class, except that A is an interface. The first example is when the methods are not overridden in C/D. The graph for this is Figure 2.14 and the code is here:

```
public interface AIF {
    public int compare(AIF obj);
}
----------------------------------
public class B implements AIF {
    protected int item;
    protected int item2;
    public B() {
        item = 5;
        item2 = 6;
    }
    public int compare(AIF obj) {
```

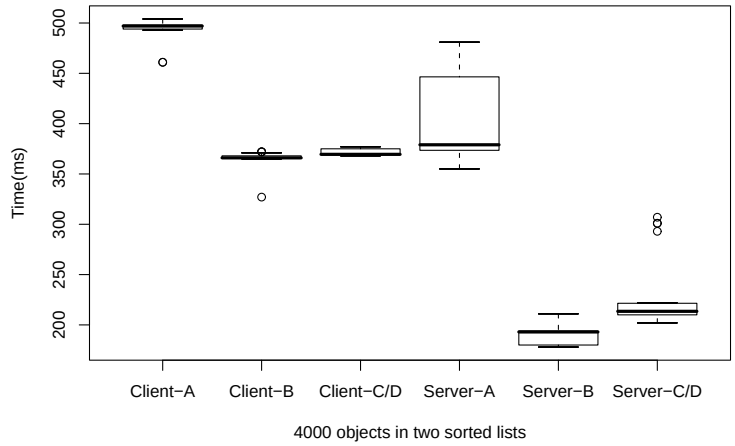

Figure 2.14: Parameterized class; compare() is not overridden in C and D. A is an interface

```
        return "Hi".compareTo("Bye");
    }
}
-----------------------------------
public class C extends B {
    private String Cstring;
    public C() {
        super();
        Cstring = "C";
    }
}
-----------------------------------
public class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
}
```

The next case is when the class is overridden in C and D and A is still an interface. The graph is Figure 2.15 and the code is below:

```
public interface AIF {
    public int compare(AIF obj);
```

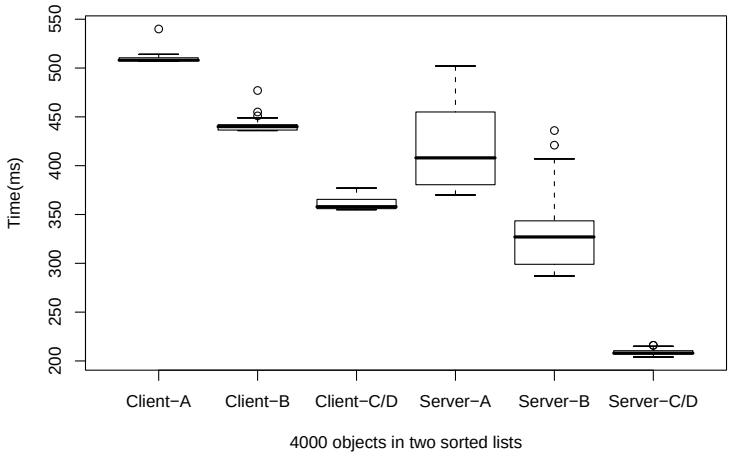

Figure 2.15: Parameterized class; compare() is overridden in C and D. A is an interface

```
}
-----------------------------------
public class B implements AIF {
    protected int item;
    protected int item2;
    public B() {
        item = 5;
        item2 = 6;
    }
    public int compare(AIF obj) {
        return "Hi".compareTo("Bye");
    }
}
-----------------------------------
public class C extends B {
    private String Cstring;
    public C() {
        super();
        Cstring = "C";
    }
    public int compare(AIF obj) {
        return Cstring.compareTo("A");
    }
}
```

```
---------------------------------
public class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
    public int compare(AIF obj) {
        return Dstring.compareTo("A");
    }
}
```

The results from these examples confirm our findings for interface bounds with parameterized methods: interface bounds create an overhead over concrete class bounds. The graph shows that A is always the slowest run, and then B's time depends on whether or not the method is overridden in C and D, just like the parameterized method examples.

2.4 Typecasting examples

These test cases were developed in attempt to measure the overhead of typecasting for bounds that are not the most precise.

The testing code is quite different from that of the parameterized methods and classes examples, but still uses the A,B,C,D hierarchy we developed (i.e. B extends A, C and D extend B). We created several copies of a class with one method that was parameterized. It was parameterized to one of the bounds and accepted one parameter of that type and just returned it. The main testing classes then assigned it to a variable of the type C or D, respectively. The testing classes creates a large number of C objects and an equal number of D objects and calls this method on them in a loop.

This code is run with the methods parameterized to the three different type bounds: A,B, and C/D, just like the rest of our test cases. Now we compare these results to an identical run, except that the return values from the methods are not assigned to variables. We want to see what amount of time the typecast from the method return to the variable takes. There are two graphs, one for the client run of all six cases, and one for the server run. The client graph is Figure 2.16 and the server is Figure 2.17.

Below is the code:

```
public class TestNoTypeCastA {
    public static void main(String [] args) {
        int n = 10000;
        int m = 10000;
        C elem1;
        D elem2;
        C [] carray = new C[n];
```

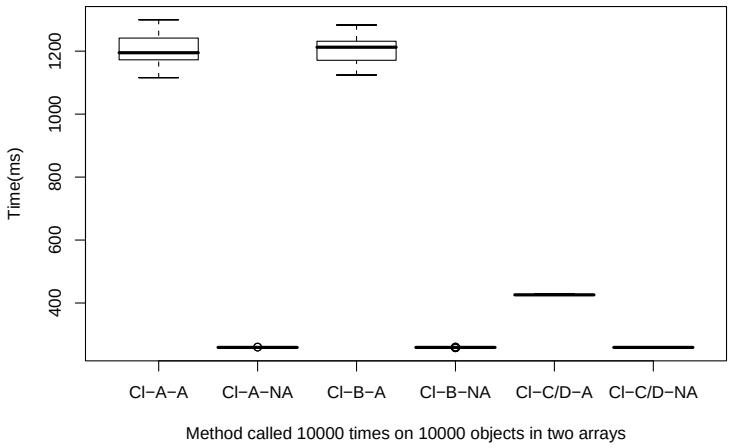

Figure 2.16: Client runs with/without variable assignment

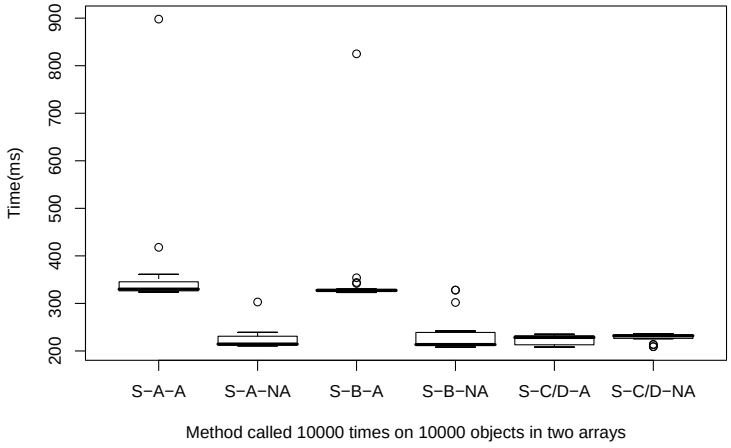

Figure 2.17: Server runs with/without variable assignment

```
        D [] darray = new D[n];
        for (int i = 0; i < n; i++) {
            carray[i] = new C();
            darray[i] = new D();
        }

        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int j = 0; j < m; j++) {
            for (int i = 0; i < n; i++) {
                TypeCastA.returnMethod(carray[i]);
                TypeCastA.returnMethod(darray[i]);
            }
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-A-NA\t" + (time2-time1));
    }
}
-----------------------------------------------------------------------
public class TestNoTypeCastB {
    public static void main(String [] args) {
        int n = 10000;
        int m = 10000;
        C elem1;
        D elem2;
        C [] carray = new C[n];
        D [] darray = new D[n];
        for (int i = 0; i < n; i++) {
            carray[i] = new C();
            darray[i] = new D();
        }

        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int j = 0; j < m; j++) {
            for (int i = 0; i < n; i++) {
                TypeCastB.returnMethod(carray[i]);
                TypeCastB.returnMethod(darray[i]);
            }
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-B-NA\t" + (time2-time1));
    }
}
```

```
-----------------------------------------------------------------------
public class TestNoTypeCastCD {
    public static void main(String [] args) {
        int n = 10000;
        int m = 10000;
        C elem1;
        D elem2;
        C [] carray = new C[n];
        D [] darray = new D[n];
        for (int i = 0; i < n; i++) {
            carray[i] = new C();
            darray[i] = new D();
        }

        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int j = 0; j < m; j++) {
            for (int i = 0; i < n; i++) {
                TypeCastCD.returnMethodC(carray[i]);
                TypeCastCD.returnMethodD(darray[i]);
            }
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-C/D-NA\t" + (time2-time1));
    }
}
-----------------------------------------------------------------------
public class TestTypeCastA {
    public static void main(String [] args) {
        int n = 10000;
        int m = 10000;
        C elem1;
        D elem2;
        C [] carray = new C[n];
        D [] darray = new D[n];
        for (int i = 0; i < n; i++) {
            carray[i] = new C();
            darray[i] = new D();
        }

        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int j = 0; j < m; j++) {
            for (int i = 0; i < n; i++) {
```

```
                elem1 = TypeCastA.returnMethod(carray[i]);
                elem2 = TypeCastA.returnMethod(darray[i]);
            }
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-A-A\t" + (time2-time1));
    }
}
-----------------------------------------------------------------------
public class TestTypeCastB {
    public static void main(String [] args) {
        int n = 10000;
        int m = 10000;
        C elem1;
        D elem2;
        C [] carray = new C[n];
        D [] darray = new D[n];
        for (int i = 0; i < n; i++) {
            carray[i] = new C();
            darray[i] = new D();
        }

        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int j = 0; j < m; j++) {
            for (int i = 0; i < n; i++) {
                elem1 = TypeCastB.returnMethod(carray[i]);
                elem2 = TypeCastB.returnMethod(darray[i]);
            }
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-B-A\t" + (time2-time1));
    }
}
-----------------------------------------------------------------------
public class TestTypeCastCD {
    public static void main(String [] args) {
        int n = 10000;
        int m = 10000;
        C elem1;
        D elem2;
        C [] carray = new C[n];
        D [] darray = new D[n];
        for (int i = 0; i < n; i++) {
            carray[i] = new C();
```

```
            darray[i] = new D();
        }

        long time1;
        long time2;
        time1 = System.currentTimeMillis();
        for (int j = 0; j < m; j++) {
            for (int i = 0; i < n; i++) {
                elem1 = TypeCastCD.returnMethodC(carray[i]);
                elem2 = TypeCastCD.returnMethodD(darray[i]);
            }
        }
        time2 = System.currentTimeMillis();
        System.out.println(args[0] + "-C/D-A\t" + (time2-time1));
    }
}
-----------------------------------------------------------------------
public class TypeCastA {
    public static <T extends A> T returnMethod(T arg) {
        return arg;
    }
}
-----------------------------------------------------------------------
public class TypeCastB {
    public static <T extends B> T returnMethod(T arg) {
        return arg;
    }
}
-----------------------------------------------------------------------
public class TypeCastCD {
    public static <T extends C> T returnMethodC(T arg) {
        return arg;
    }
    public static <T extends D> T returnMethodD(T arg) {
        return arg;
    }
}
-----------------------------------------------------------------------
public class A {
    protected int item;
    public A() {
        item = 5;
    }
    public int compare(A obj) {
        return "Hi".compareTo("Bye");
    }
```

```
}
-------------------------------------------------------------------------
public class B extends A {
    protected int item2;
    public B() {
        super();
        item2 = 6;
    }
    public int compare(A obj) {
        return "Hi".compareTo("Bye");
    }
}
-------------------------------------------------------------------------
public class C extends B {
    private String Cstring;
    public C() {
        super();
        Cstring = "C";
    }
    public int compare(A obj) {
        return Cstring.compareTo("A");
    }
}
-------------------------------------------------------------------------
public class D extends B {
    private String Dstring;
    public D() {
        super();
        Dstring = "D";
    }
    public int compare(A obj) {
        return Dstring.compareTo("A");
    }
}
```

These results for the case when the method `compare` is not overridden in C/D show us that even though B is a better choice as a bound in the case when no assignment is performed, C/D may be better if the result from a parameterized method is going to be assigned to a variable of the actual type of the result. Depending on the case, typecasting can decide which bound is better to choose.

Note that the server run of the lowest bound (C/D) with an assignment and typecast runs at the same speed as the server run of C/D without the assignment. One possible explanation is that the server JIT compiler is smart enough to eliminate the assignment altogether because the result is never used. However, this needs a further investigation.

Chapter 3

Visitor Pattern

3.1 Notations and Summary of Results

Visitor design pattern. This group of tests deals with a somewhat larger and more practical code example: we study different ways of optimizing a well-known visitor design pattern [2]. Visitor pattern is a common pattern in which a data structure (a binary tree in our example) has a method `accept` that takes a *visitor* object specified as an interface. Each visitor performs a specific task (e.g. computing the size of the tree) as it traverses the data structure. The pattern provides the convenience of having only one method in the data structure that can be used for any task if supplied an appropriate visitor. However, this approach creates a substantial overhead since neither the visitor is specialized for the actual type of the structure (assuming that the structure is itself generic), nor is the data structure specialized for a specific visitor.

About the test example. The example that we used follows the structure of the visitor pattern. However, the specific implementation does not claim to be particularly meaningful or efficient. For instance, for simplicity we used pre-existing code for a binary search tree and inserted elements according to the binary search order, but the `FindVisitor` does not use this information when it is looking for a specific element (and in any case our tree class is equipped with a parameterized search method that performs the task more efficiently).

The purpose of the `FindVisitor` in our example was to have a visitor that traverses the entire tree, calling a method on each element it visits. While in this particular case the task could have been accomplished more efficiently, the point of the example was to study this type of a visitor. Similarly, from the point of view of our study `SizeVisitor` represents any visitor that traverses the tree but does not call any methods on the elements. Both kinds of visitors are found in practice. Likewise, the need to return a wrapped type instead of a primitive type seems awkward, but comes up in real-life code because a type parameter in a generic type cannot be a primitive type.

The program structure. The program consists of parameterized `TreeNode`

and `Tree` classes, a `TreeVisitor` interface with two parameters: the type of the tree and the return type of the visitor's method `visit` that takes a `TreeNode`, and two visitors: a `SizeVisitor` that computes the size (i.e. the number of nodes) of a tree and a `FindVisitor` that looks for a specific element in the tree and returns `Boolean.TRUE` if the element is found and `Boolean.FALSE` otherwise. The return type of the `visit` method of the `SizeVisitor` is `Integer`. The test classes are `TestSizeVisitor` for the `SizeVisitor` and `TestFindVisitor` for `FindVisitor`.

Test setup. In tests for both visitors `TestSizeVisitor` and `TestFindVisitor` the program created two trees, one of Integers and one of Strings, with 70000 elements each, and passed the respective visitor to the `accept` method of both trees. In the case of the FindVisitor we repeated the task 10 times to better compare the results. Figure 3.5 summarizes the results. Figures 3.1 and 3.2 give the graphs of the results for the `FindVisitor`, and Figures 3.3 and 3.4 are graphs for the `SizeVisitor` results. The general testing methodology and the general graph notations are explained in section 2.1. All tests were run 20 times in the client and the server mode of the HotSpot JVM. The only important difference is that for the visitor pattern we base our comparison on the **user CPU time**, not the loop times, since this is a more complex program. The user CPU time takes into account all the factors, including the class loading and the overhead of performing run-time optimization.

Optimizations performed. The graphs compare results for different ways of specializing the code. The optimizations and their notations on graphs are as follows:

- `Non-Opt` stands for the non-optimized code
- `Return` specializes the type bound of the return type of the `visit` method of the two visitors
- `Interface` extends `Return` by additionally creating two copies of the interface - one for `Integer` return value and one for `Boolean`
- `Tree` creates specialized classes for the two types of trees (`Integer` and `String`) and correspondingly two copies of the visitor classes and interfaces, also specialized for the type of tree elements (but not the return value)
- `TreeReturn` combines the `Tree` and the `Return` specialization.

Since the `Interface` didn't have any improvement over `Return`, we didn't try it in combination with `Tree`.

Results. The table 3.5 shows the percent increase in efficiency or medians of runtimes for the different optimizations tested in the visitor pattern test cases. All optimization times are compared to that of the original non-optimized code.

We observe that `Return` and `Interface` do not lead to increase in efficiency (in fact, `Interface` shows a slight slowdown). `Tree` shows an impressive efficiency increase of around 17% and 18% for client mode, and 12% for server

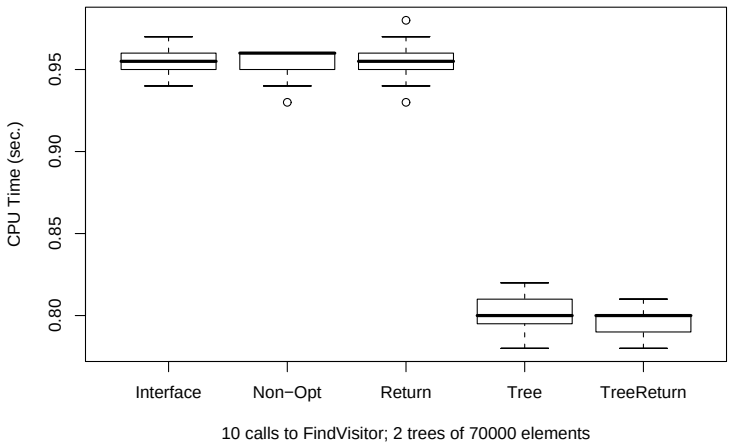

Figure 3.1: FindVisitor optimization in the client mode

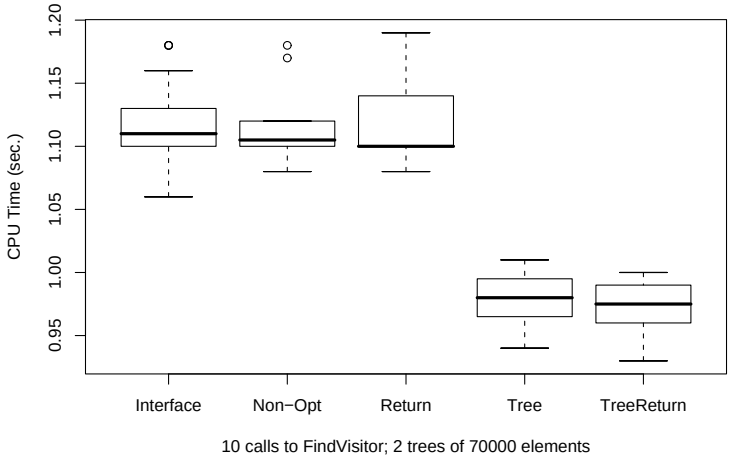

Figure 3.2: FindVisitor optimization in the server mode

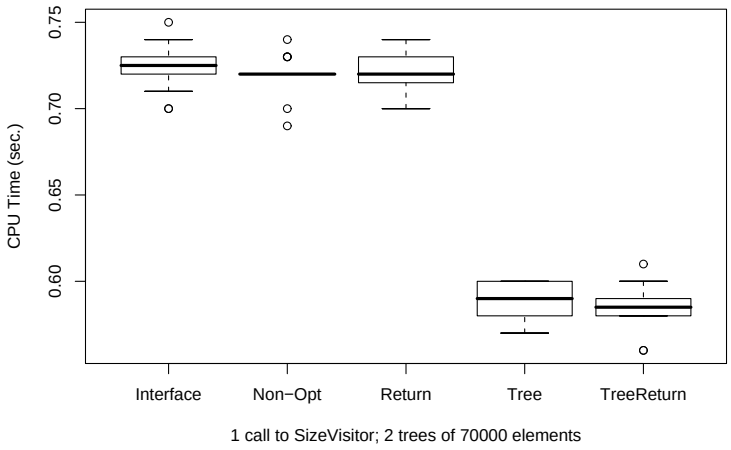

Figure 3.3: SizeVisitor optimization in the client mode

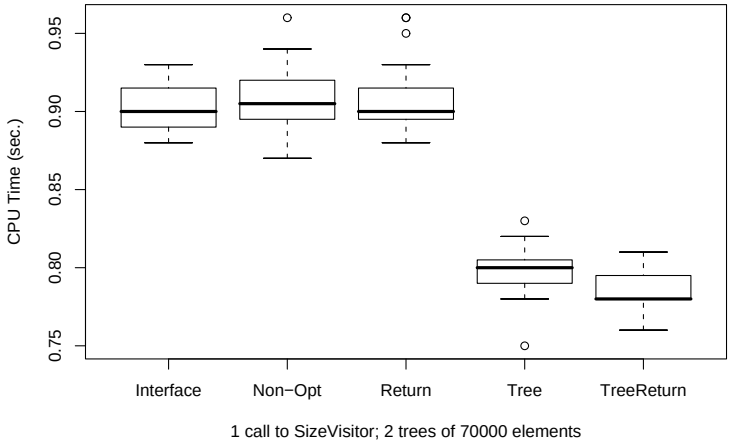

Figure 3.4: SizeVisitor optimization in the server mode

	Client				Server			
	Find		Size		Find		Size	
	Time	%	Time	%	Time	%	Time	%
Non-Opt	0.96	–	0.72	–	1.11	–	0.91	–
Interface	0.96	0	0.73	-0.01	1.11	0	0.90	0.01
Return	0.96	0	0.72	0	1.10	0.01	0.90	0.01
Tree	0.80	0.17	0.59	0.18	0.98	0.12	0.80	0.12
TreeReturn	0.80	0.17	0.59	0.18	0.98	0.12	0.78	0.14

Figure 3.5: Percentage efficiency increase

mode, likely due to method devirtualization. It is interesting that adding `Return` to `Tree` (see `TreeReturn`) clearly added efficiency to `SizeVisitor` test for the server mode, raising the total increase to 14%. The graphs seem to indicate that in the other cases `TreeReturn` was also more efficient, even though not so noticeably.

The rest of the section contains code for all of the test examples. We included a complete set for each case of an optimization even if some of the files did not change.

3.2 Code for the Non-Optimized case (`Non-Opt`)

Here is the Non-Opt code:

```
public class Tree<T extends Comparable<T>> {
    private TreeNode<T> root;

    public Tree() {
    }

    public boolean isEmpty() {
        return (root == null);
    }

    public TreeNode<T> getRoot(){
        return root;
    }

    public <S> S accept(TreeVisitor<T,S> tv) {
        if (root != null) return tv.visit(root);
        return null;
    }

    public TreeNode search(T data ) {
        TreeNode<T> node = root;
```

```
    while (node != null && data != node.getData()){
        if (data.compareTo(node.getData()) > 0){
            node = node.getRight();
        } else {
            node = node.getLeft();
        }
    }
    return node;
}

public TreeNode<T> insert(T data) {
    if(!isEmpty()){
        TreeNode<T> parent = getRoot();
        TreeNode<T> child;
        if (data.compareTo(parent.getData()) == 0) {
            return null;
        }
        if(data.compareTo(parent.getData()) < 0){
            child = parent.getLeft();
        } else {
            child = parent.getRight();
        }

        while(child != null){
            if(data.compareTo(child.getData()) < 0){
                parent = child;
                child = child.getLeft();
            }
            else if (data.compareTo(child.getData()) > 0) {
                parent = child;
                child = child.getRight();
            }
            else return null;
        }
        TreeNode<T> node = new TreeNode<T>(data);
        if(parent.getData().compareTo(data) < 0){
            parent.setRight(node);
        }
        else{
            parent.setLeft(node);
        }
        return node;
    }

    TreeNode<T> node = new TreeNode<T>(data);
    root = node;
```

```
        return node;
    }

}
-----------------------------------------------------------------
public class TreeNode<T extends Comparable<T>> {
    private TreeNode<T> left, right;
    private T data;

    public TreeNode(T d){
        data = d;
    }

    public TreeNode<T> getLeft(){
        return left;
    }

    public TreeNode<T> getRight(){
        return right;
    }
    public void setLeft(TreeNode<T> in){
        left = in;
    }
    public void setRight(TreeNode<T> in){
        right = in;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}
-----------------------------------------------------------------
public interface TreeVisitor<T extends Comparable<T>, S> {
    public S visit(TreeNode<T> node);
}
-----------------------------------------------------------------
public class FindVisitor<T extends Comparable<T>, S> implements TreeVisitor<T,S> {
    private T _toFind;
    public FindVisitor(T toFind) {
        _toFind = toFind;
    }
```

```
    public S visit(TreeNode<T> node) {
        if (_toFind.compareTo(node.getData()) == 0) {
            return (S) Boolean.TRUE;
        }
        if (node.getLeft() != null && visit(node.getLeft()) == Boolean.TRUE) {
            return (S) Boolean.TRUE;
        }
        if (node.getRight() != null) {
            return (S) visit(node.getRight());
        }
        return (S) Boolean.FALSE;
    }
}
-----------------------------------------------------------------
public class SizeVisitor<T extends Comparable<T>, S> implements TreeVisitor<T, S> {
    public S visit(TreeNode<T> node) {
        int size = 0;
        if (node.getLeft() != null) size += ((Integer)
            (visit(node.getLeft()))).intValue();
        if (node.getRight() != null) size += ((Integer)
            (visit(node.getRight()))).intValue();
        return (S) new Integer(size + 1);
    }
}
-----------------------------------------------------------------
import java.util.*;

public class TestFindVisitor {
    public static void main(String [] args) {
        Random r = new Random(10); // a random number generator with a fixed seed

        Tree<Integer> integers = new Tree<Integer>();
        Tree<String> strings = new Tree<String>();

        for (int i = 0; i < 70000; ++i) {
            int n = r.nextInt(10000000);
            integers.insert(new Integer(n));
            strings.insert(new String("" + n));
        }

        long time1, time2, time3;

        time1 = System.currentTimeMillis();

        FindVisitor<Integer,Boolean> fv1 = new FindVisitor<Integer,Boolean>
             (new Integer(5555));
```

```
        // visiting the tree of integers
        for (int i = 0; i < 10; ++i) {
            boolean found1 = ((Boolean) integers.accept(fv1)).booleanValue();
        }

        time2 = System.currentTimeMillis();

        FindVisitor<String, Boolean> fv2 = new FindVisitor<String, Boolean>("5555");
        // visiting the tree of strings
        for (int i = 0; i < 10; ++i) {
            boolean found2 = ((Boolean) strings.accept(fv2)).booleanValue();
        }

        time3 = System.currentTimeMillis();

        System.out.println("Time for integers " + (time2 - time1));
        System.out.println("Time for strings " + (time3 - time2));
    }
}
-----------------------------------------------------------------
import java.util.*;

public class TestSizeVisitor {
    public static void main(String [] args) {
        Random r = new Random(10); // a random number generator with a fixed seed

        Tree<Integer> integers = new Tree<Integer>();
        Tree<String> strings = new Tree<String>();

        for (int i = 0; i < 70000; ++i) {
            int n = r.nextInt(10000000);
            integers.insert(new Integer(n));
            strings.insert(new String("" + n));
        }

        long time1, time2, time3;
        time1 = System.currentTimeMillis();

        // visiting the tree of integers
        SizeVisitor<Integer,Integer> sv1 = new SizeVisitor<Integer,Integer>();
        int size1 = ((Integer) integers.accept(sv1)).intValue();

        time2 = System.currentTimeMillis();

        // visiting the tree of strings
        SizeVisitor<String, Integer> sv2 = new SizeVisitor<String, Integer>();
```

```
        int size2 = ((Integer) strings.accept(sv2)).intValue();

        time3 = System.currentTimeMillis();

        System.out.println("Time for integers " + (time2 - time1));
        System.out.println("Time for strings " + (time3 - time2));
    }
}
```

3.3 Visitor Classes Parameterized by the Return Type (Return)

```
-------------------------------------------------------------
public class Tree<T extends Comparable<T>> {
    private TreeNode<T> root;

    public Tree() {
    }

    public boolean isEmpty() {
        return (root == null);
    }

    public TreeNode<T> getRoot(){
        return root;
    }

    public <S> S accept(TreeVisitor<T,S> tv) {
        if (root != null) return tv.visit(root);
        return null;
    }

    public TreeNode search(T data ) {
        TreeNode<T> node = root;

        while (node != null && data != node.getData()){
            if (data.compareTo(node.getData()) > 0){
                node = node.getRight();
            } else {
                node = node.getLeft();
            }
        }
        return node;
    }
```

```
    public TreeNode<T> insert(T data) {
        if(!isEmpty()){
            TreeNode<T> parent = getRoot();
            TreeNode<T> child;
            if (data.compareTo(parent.getData()) == 0) {
                return null;
            }
            if(data.compareTo(parent.getData()) < 0){
                child = parent.getLeft();
            } else {
                child = parent.getRight();
            }

            while(child != null){
                if(data.compareTo(child.getData()) < 0){
                    parent = child;
                    child = child.getLeft();
                }
                else if (data.compareTo(child.getData()) > 0) {
                    parent = child;
                    child = child.getRight();
                }
                else return null;
            }
            TreeNode<T> node = new TreeNode<T>(data);
            if(parent.getData().compareTo(data) < 0){
                parent.setRight(node);
            }
            else{
                parent.setLeft(node);
            }
            return node;
        }

        TreeNode<T> node = new TreeNode<T>(data);
        root = node;
        return node;
    }

}
-----------------------------------------------------------------
public class TreeNode<T extends Comparable<T>> {
    private TreeNode<T> left, right;
    private T data;
```

```
    public TreeNode(T d){
        data = d;
    }

    public TreeNode<T> getLeft(){
        return left;
    }

    public TreeNode<T> getRight(){
        return right;
    }
    public void setLeft(TreeNode<T> in){
        left = in;
    }
    public void setRight(TreeNode<T> in){
        right = in;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }

}
-------------------------------------------------------------
public interface TreeVisitor<T extends Comparable<T>, S> {
    public S visit(TreeNode<T> node);
}
-------------------------------------------------------------
public class FindVisitor<T extends Comparable<T>, S extends Boolean>
            implements TreeVisitor<T,S> {
    private T _toFind;
    public FindVisitor(T toFind) {
        _toFind = toFind;
    }

    public S visit(TreeNode<T> node) {
        if (_toFind.compareTo(node.getData()) == 0) {
            return (S) Boolean.TRUE;
        }
        if (node.getLeft() != null && visit(node.getLeft()) ==
        Boolean.TRUE) {
            return (S) Boolean.TRUE;
```

```
        }
        if (node.getRight() != null) {
            return (S) visit(node.getRight());
        }
        return (S) Boolean.FALSE;
    }
}

-------------------------------------------------------------
public class SizeVisitor<T extends Comparable<T>, S extends Integer>
            implements TreeVisitor<T, S> {
    public S visit(TreeNode<T> node) {
        int size = 0;
        if (node.getLeft() != null) size += ((Integer)
            (visit(node.getLeft()))).intValue();
        if (node.getRight() != null) size += ((Integer)
            (visit(node.getRight()))).intValue();
        return (S) new Integer(size + 1);
    }
}

-------------------------------------------------------------
import java.util.*;

public class TestFindVisitor {
    public static void main(String [] args) {
        Random r = new Random(10); // a random number generator with a fixed seed

        Tree<Integer> integers = new Tree<Integer>();
        Tree<String> strings = new Tree<String>();

        for (int i = 0; i < 70000; ++i) {
            int n = r.nextInt(10000000);
            integers.insert(new Integer(n));
            strings.insert(new String("" + n));
        }

        long time1, time2, time3;

        time1 = System.currentTimeMillis();

        FindVisitor<Integer,Boolean> fv1 = new FindVisitor<Integer,Boolean>
            (new Integer(5555));
        // visiting the tree of integers
        for (int i = 0; i < 10; ++i) {
            boolean found1 = ((Boolean) integers.accept(fv1)).booleanValue();
```

```
        }

        time2 = System.currentTimeMillis();

        FindVisitor<String, Boolean> fv2 = new FindVisitor<String, Boolean>("5555");
        // visiting the tree of strings
        for (int i = 0; i < 10; ++i) {
            boolean found2 = ((Boolean) strings.accept(fv2)).booleanValue();
        }

        time3 = System.currentTimeMillis();

        System.out.println("Time for integers " + (time2 - time1));
        System.out.println("Time for strings " + (time3 - time2));
    }
}
```

```
import java.util.*;

public class TestSizeVisitor {
    public static void main(String [] args) {
        Random r = new Random(10); // a random number generator with a fixed seed

        Tree<Integer> integers = new Tree<Integer>();
        Tree<String> strings = new Tree<String>();

        for (int i = 0; i < 70000; ++i) {
            int n = r.nextInt(10000000);
            integers.insert(new Integer(n));
            strings.insert(new String("" + n));
        }

        long time1, time2, time3;
        time1 = System.currentTimeMillis();

        // visiting the tree of integers
        SizeVisitor<Integer,Integer> sv1 = new SizeVisitor<Integer,Integer>();
        int size1 = ((Integer) integers.accept(sv1)).intValue();

        time2 = System.currentTimeMillis();

        // visiting the tree of strings
        SizeVisitor<String, Integer> sv2 = new SizeVisitor<String, Integer>();
        int size2 = ((Integer) strings.accept(sv2)).intValue();
```

```
        time3 = System.currentTimeMillis();

        System.out.println("Time for integers " + (time2 - time1));
        System.out.println("Time for strings " + (time3 - time2));
    }
}
```

3.4 Interface Parameterized by Return Type (`Interface`)

Below is the code for the specialized interface optimization `Interface`. The optimization creates two copies of the `TreeVisitor` interface parameterized by the return type of the method `visit`. Each of the two visitors implements the appropriate interface.

```
public class Tree<T extends Comparable<T>> {
    private TreeNode<T> root;

    public Tree() {
    }

    public boolean isEmpty() {
        return (root == null);
    }

    public TreeNode<T> getRoot(){
        return root;
    }

    public <S> S accept(TreeVisitorInt<T,Integer> tv) {
        if (root != null) return (S) tv.visit(root);
        return null;
    }

    public <S> S accept(TreeVisitorBool<T,Boolean> tv) {
        if (root != null) return (S) tv.visit(root);
        return null;
    }

    public TreeNode search(T data ) {
        TreeNode<T> node = root;

        while (node != null && data != node.getData()){
            if (data.compareTo(node.getData()) > 0){
                node = node.getRight();
            } else {
```

```
                node = node.getLeft();
            }
        }
        return node;
    }

    public TreeNode<T> insert(T data) {
        if(!isEmpty()){
            TreeNode<T> parent = getRoot();
            TreeNode<T> child;
            if (data.compareTo(parent.getData()) == 0) {
                return null;
            }
            if(data.compareTo(parent.getData()) < 0){
                child = parent.getLeft();
            } else {
                child = parent.getRight();
            }

            while(child != null){
                if(data.compareTo(child.getData()) < 0){
                    parent = child;
                    child = child.getLeft();
                }
                else if (data.compareTo(child.getData()) > 0) {
                    parent = child;
                    child = child.getRight();
                }
                else return null;
            }
            TreeNode<T> node = new TreeNode<T>(data);
            if(parent.getData().compareTo(data) < 0){
                parent.setRight(node);
            }
            else{
                parent.setLeft(node);
            }
            return node;
        }

        TreeNode<T> node = new TreeNode<T>(data);
        root = node;
        return node;
    }
}
```

```
public class TreeNode<T extends Comparable<T>> {
    private TreeNode<T> left, right;
    private T data;

    public TreeNode(T d){
        data = d;
    }

    public TreeNode<T> getLeft(){
        return left;
    }

    public TreeNode<T> getRight(){
        return right;
    }
    public void setLeft(TreeNode<T> in){
        left = in;
    }
    public void setRight(TreeNode<T> in){
        right = in;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}
-----------------------------------------------------------------
public interface TreeVisitorBool<T extends Comparable<T>, S extends Boolean> {
    public S visit(TreeNode<T> node);
}
-----------------------------------------------------------------
public interface TreeVisitorInt<T extends Comparable<T>, S extends Integer> {
    public S visit(TreeNode<T> node);
}
-----------------------------------------------------------------
public class FindVisitor<T extends Comparable<T>, S extends Boolean>
            implements TreeVisitorBool<T,S> {
    private T _toFind;
    public FindVisitor(T toFind) {
        _toFind = toFind;
    }
```

```
    public S visit(TreeNode<T> node) {
        if (_toFind.compareTo(node.getData()) == 0) {
            return (S) Boolean.TRUE;
        }
        if (node.getLeft() != null && visit(node.getLeft()) == Boolean.TRUE) {
            return (S) Boolean.TRUE;
        }
        if (node.getRight() != null) {
            return (S) visit(node.getRight());
        }
        return (S) Boolean.FALSE;
    }
}
-----------------------------------------------------------------
public class SizeVisitor<T extends Comparable<T>, S extends Integer>
            implements TreeVisitorInt<T, S> {
    public S visit(TreeNode<T> node) {
        int size = 0;
        if (node.getLeft() != null) size += ((Integer)
            (visit(node.getLeft()))).intValue();
        if (node.getRight() != null) size += ((Integer)
            (visit(node.getRight()))).intValue();
        return (S) new Integer(size + 1);
    }
}
-----------------------------------------------------------------
import java.util.*;

public class TestFindVisitor {
    public static void main(String [] args) {
    Random r = new Random(10); // a random number generator with a fixed seed

    Tree<Integer> integers = new Tree<Integer>();
    Tree<String> strings = new Tree<String>();

    for (int i = 0; i < 70000; ++i) {
        int n = r.nextInt(10000000);
        integers.insert(new Integer(n));
        strings.insert(new String("" + n));
    }

    long time1, time2, time3;

    time1 = System.currentTimeMillis();

    FindVisitor<Integer,Boolean> fv1 = new FindVisitor<Integer,Boolean>
```

```
        (new Integer(5555));
    // visiting the tree of integers
    for (int i = 0; i < 10; ++i) {
        boolean found1 = ((Boolean) integers.accept(fv1)).booleanValue();
    }

    time2 = System.currentTimeMillis();

    FindVisitor<String, Boolean> fv2 = new FindVisitor<String, Boolean>("5555");
    // visiting the tree of strings
    for (int i = 0; i < 10; ++i) {
        boolean found2 = ((Boolean) strings.accept(fv2)).booleanValue();
    }

    time3 = System.currentTimeMillis();

    System.out.println("Time for integers " + (time2 - time1));
    System.out.println("Time for strings " + (time3 - time2));
    }
}
-------------------------------------------------------------
import java.util.*;

public class TestSizeVisitor {
    public static void main(String [] args) {
    Random r = new Random(10); // a random number generator with a fixed seed

    Tree<Integer> integers = new Tree<Integer>();
    Tree<String> strings = new Tree<String>();

    for (int i = 0; i < 70000; ++i) {
        int n = r.nextInt(10000000);
        integers.insert(new Integer(n));
        strings.insert(new String("" + n));
    }

    long time1, time2, time3;
    time1 = System.currentTimeMillis();

    // visiting the tree of integers
    SizeVisitor<Integer,Integer> sv1 = new SizeVisitor<Integer,Integer>();
    int size1 = ((Integer) integers.accept(sv1)).intValue();

    time2 = System.currentTimeMillis();

    // visiting the tree of strings
```

```
    SizeVisitor<String, Integer> sv2 = new SizeVisitor<String, Integer>();
    int size2 = ((Integer) strings.accept(sv2)).intValue();

    time3 = System.currentTimeMillis();

    System.out.println("Time for integers " + (time2 - time1));
    System.out.println("Time for strings " + (time3 - time2));
    }
}
```

3.5 Tree Classes Specialized by the Type of Tree Data (Tree)

Two copies of the Tree class (and TreeNode class) are created: one for Integer and one for String. Consequently, two copies TreeVisitor interface are specialized by the type of the tree elements and two copies of each tree visitor are created. This optimization gives the most benefit.

```
-----------------------------------------------------------------
public class TreeInt<T extends Integer> {
    private TreeNodeInt<T> root;

    public TreeInt() {
    }

    public boolean isEmpty() {
         return (root == null);
    }

    public TreeNodeInt<T> getRoot(){
        return root;
    }

    public <S> S accept(TreeVisitorInt<T,S> tv) {
        if (root != null) return tv.visit(root);
        return null;
    }

    public TreeNodeInt<T> search(T data ) {
        TreeNodeInt<T> node = root;

        while (node != null && data != node.getData()){
            if (data.compareTo(node.getData()) > 0){
                node = node.getRight();
            } else {
```

```
                node = node.getLeft();
            }
        }
        return node;
    }

    public TreeNodeInt<T> insert(T data) {
        if(!isEmpty()){
            TreeNodeInt<T> parent = getRoot();
            TreeNodeInt<T> child;
            if (data.compareTo(parent.getData()) == 0) {
                return null;
            }
            if(data.compareTo(parent.getData()) < 0){
                child = parent.getLeft();
            } else {
                child = parent.getRight();
            }

            while(child != null){
                if(data.compareTo(child.getData()) < 0){
                    parent = child;
                    child = child.getLeft();
                }
                else if (data.compareTo(child.getData()) > 0) {
                    parent = child;
                    child = child.getRight();
                }
                else return null;
            }
            TreeNodeInt<T> node = new TreeNodeInt<T>(data);
            if(parent.getData().compareTo(data) < 0){
                parent.setRight(node);
            }
            else{
                parent.setLeft(node);
            }
            return node;
        }

        TreeNodeInt<T> node = new TreeNodeInt<T>(data);
        root = node;
        return node;
    }

}
```

```
-------------------------------------------------------------

public class TreeStr<T extends String> {
    private TreeNodeStr<T> root;

    public TreeStr() {
    }

    public boolean isEmpty() {
        return (root == null);
    }

    public TreeNodeStr<T> getRoot(){
        return root;
    }

    public <S> S accept(TreeVisitorStr<T,S> tv) {
        if (root != null) return tv.visit(root);
        return null;
    }

    public TreeNodeStr<T> search(T data ) {
        TreeNodeStr<T> node = root;

        while (node != null && data != node.getData()){
            if (data.compareTo(node.getData()) > 0){
                node = node.getRight();
            } else {
                node = node.getLeft();
            }
        }
        return node;
    }

    public TreeNodeStr<T> insert(T data) {
        if(!isEmpty()){
            TreeNodeStr<T> parent = getRoot();
            TreeNodeStr<T> child;
            if (data.compareTo(parent.getData()) == 0) {
                return null;
            }
            if(data.compareTo(parent.getData()) < 0){
                child = parent.getLeft();
            } else {
                child = parent.getRight();
            }
```

```
            while(child != null){
                if(data.compareTo(child.getData()) < 0){
                    parent = child;
                    child = child.getLeft();
                }
                else if (data.compareTo(child.getData()) > 0) {
                    parent = child;
                    child = child.getRight();
                }
                else return null;
            }
            TreeNodeStr<T> node = new TreeNodeStr<T>(data);
            if(parent.getData().compareTo(data) < 0){
                parent.setRight(node);
            }
            else{
                parent.setLeft(node);
            }
            return node;
        }

        TreeNodeStr<T> node = new TreeNodeStr<T>(data);
        root = node;
        return node;
    }

}
-----------------------------------------------------------------

public class TreeNodeInt<T extends Integer> {
    private TreeNodeInt<T> left, right;
    private T data;

    public TreeNodeInt(T d){
        data = d;
    }

    public TreeNodeInt<T> getLeft(){
        return left;
    }

    public TreeNodeInt<T> getRight(){
        return right;
    }
    public void setLeft(TreeNodeInt<T> in){
```

```
        left = in;
    }
    public void setRight(TreeNodeInt<T> in){
        right = in;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}
```

```
public class TreeNodeStr<T extends String> {
    private TreeNodeStr<T> left, right;
    private T data;

    public TreeNodeStr(T d){
        data = d;
    }

    public TreeNodeStr<T> getLeft(){
        return left;
    }

    public TreeNodeStr<T> getRight(){
        return right;
    }
    public void setLeft(TreeNodeStr<T> in){
        left = in;
    }
    public void setRight(TreeNodeStr<T> in){
        right = in;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}
```

```
-------------------------------------------------------------
public interface TreeVisitorInt<T extends Integer, S> {
    public S visit(TreeNodeInt<T> node);
}

-------------------------------------------------------------
public interface TreeVisitorStr<T extends String, S> {
    public S visit(TreeNodeStr<T> node);
}
-------------------------------------------------------------
public class FindVisitorInt<T extends Integer, S>
        implements TreeVisitorInt<T,S> {
    private T _toFind;
    public FindVisitorInt(T toFind) {
        _toFind = toFind;
    }

    public S visit(TreeNodeInt<T> node) {
    if (_toFind.compareTo(node.getData()) == 0) {
        return (S) Boolean.TRUE;
    }
    if (node.getLeft() != null && visit(node.getLeft()) == Boolean.TRUE) {
        return (S) Boolean.TRUE;
    }
    if (node.getRight() != null) {
        return (S) visit(node.getRight());
    }
        return (S) Boolean.FALSE;
    }

}
-------------------------------------------------------------
public class FindVisitorStr<T extends String, S>
        implements TreeVisitorStr<T,S> {
    private T _toFind;
    public FindVisitorStr(T toFind) {
        _toFind = toFind;
    }

    public S visit(TreeNodeStr<T> node) {
    if (_toFind.compareTo(node.getData()) == 0) {
        return (S) Boolean.TRUE;
    }
    if (node.getLeft() != null && visit(node.getLeft()) == Boolean.TRUE) {
        return (S) Boolean.TRUE;
    }
```

```
    if (node.getRight() != null) {
        return (S) visit(node.getRight());
    }
        return (S) Boolean.FALSE;
    }
}

-------------------------------------------------------------
public class SizeVisitorInt<T extends Integer, S>
        implements TreeVisitorInt<T,S> {

    public S visit(TreeNodeInt<T> node) {
        int size = 0;
        if (node.getLeft() != null) size += ((Integer)
            (visit(node.getLeft()))).intValue();
        if (node.getRight() != null) size += ((Integer)
            (visit(node.getRight()))).intValue();
        return (S) new Integer(size + 1);
    }

}
-------------------------------------------------------------
public class SizeVisitorStr<T extends String, S>
        implements TreeVisitorStr<T,S> {

    public S visit(TreeNodeStr<T> node) {
        int size = 0;
        if (node.getLeft() != null) size += ((Integer)
            (visit(node.getLeft()))).intValue();
        if (node.getRight() != null) size += ((Integer)
            (visit(node.getRight()))).intValue();
        return (S) new Integer(size + 1);
    }

}
-------------------------------------------------------------
import java.util.*;

public class TestFindVisitor {
    public static void main(String [] args) {
        Random r = new Random(10); // a random number generator with a fixed seed

        TreeInt<Integer> integers = new TreeInt<Integer>();
        TreeStr<String> strings = new TreeStr<String>();

        for (int i = 0; i < 70000; ++i) {
```

```
            int n = r.nextInt(10000000);
            integers.insert(new Integer(n));
            strings.insert(new String("" + n));
}

        long time1, time2, time3;

        time1 = System.currentTimeMillis();

        FindVisitorInt<Integer,Boolean> fv1 = new FindVisitorInt<Integer,Boolean>
                        (new Integer(5555));
        // visiting the tree of integers
        for (int i = 0; i < 10; ++i) {
            boolean found1 = ((Boolean) integers.accept(fv1)).booleanValue();
        }

        time2 = System.currentTimeMillis();

        FindVisitorStr<String, Boolean> fv2 = new FindVisitorStr<String, Boolean>("5555");
        // visiting the tree of strings
        for (int i = 0; i < 10; ++i) {
            boolean found2 = ((Boolean) strings.accept(fv2)).booleanValue();
        }

        time3 = System.currentTimeMillis();

        System.out.println("Time for integers " + (time2 - time1));
        System.out.println("Time for strings " + (time3 - time2));
    }
}
```

```
import java.util.*;

public class TestSizeVisitor {
    public static void main(String [] args) {
        Random r = new Random(10); // a random number generator with a fixed seed

        TreeInt<Integer> integers = new TreeInt<Integer>();
        TreeStr<String> strings = new TreeStr<String>();

        for (int i = 0; i < 70000; ++i) {
            int n = r.nextInt(10000000);
            integers.insert(new Integer(n));
            strings.insert(new String("" + n));
        }
```

```
        long time1, time2, time3;
        time1 = System.currentTimeMillis();

        // visiting the tree of integers
        SizeVisitorInt<Integer,Integer> sv1 = new SizeVisitorInt<Integer,Integer>();
        int size1 = ((Integer) integers.accept(sv1)).intValue();

        time2 = System.currentTimeMillis();

        // visiting the tree of strings
        SizeVisitorStr<String, Integer> sv2 = new SizeVisitorStr<String, Integer>();
        int size2 = ((Integer) strings.accept(sv2)).intValue();

        time3 = System.currentTimeMillis();

        System.out.println("Time for integers " + (time2 - time1));
        System.out.println("Time for strings " + (time3 - time2));
    }
}
```

3.6 Code Specialized by the Tree Element Type and by Return Type (`TreeReturn`)

This optimization combines the `Tree` optimization and the `Return` optimization.

```
-----------------------------------------------------------------
public class TreeInt<T extends Integer> {
    private TreeNodeInt<T> root;

    public TreeInt() {
    }

    public boolean isEmpty() {
        return (root == null);
    }

    public TreeNodeInt<T> getRoot(){
        return root;
    }

    public <S> S accept(TreeVisitorInt<T,S> tv) {
        if (root != null) return tv.visit(root);
        return null;
```

```
}

public TreeNodeInt<T> search(T data ) {
    TreeNodeInt<T> node = root;

    while (node != null && data != node.getData()){
        if (data.compareTo(node.getData()) > 0){
            node = node.getRight();
        } else {
            node = node.getLeft();
        }
    }
    return node;
}

public TreeNodeInt<T> insert(T data) {
    if(!isEmpty()){
        TreeNodeInt<T> parent = getRoot();
        TreeNodeInt<T> child;
        if (data.compareTo(parent.getData()) == 0) {
            return null;
        }
        if(data.compareTo(parent.getData()) < 0){
            child = parent.getLeft();
        } else {
            child = parent.getRight();
        }

        while(child != null){
            if(data.compareTo(child.getData()) < 0){
                parent = child;
                child = child.getLeft();
            }
            else if (data.compareTo(child.getData()) > 0) {
                parent = child;
                child = child.getRight();
            }
            else return null;
        }
        TreeNodeInt<T> node = new TreeNodeInt<T>(data);
        if(parent.getData().compareTo(data) < 0){
            parent.setRight(node);
        }
        else{
            parent.setLeft(node);
        }
```

```
            return node;
        }

        TreeNodeInt<T> node = new TreeNodeInt<T>(data);
        root = node;
        return node;
    }

}

-----------------------------------------------------------------
public class TreeStr<T extends String> {
    private TreeNodeStr<T> root;

    public TreeStr() {
    }

    public boolean isEmpty() {
        return (root == null);
    }

    public TreeNodeStr<T> getRoot(){
        return root;
    }

    public <S> S accept(TreeVisitorStr<T,S> tv) {
        if (root != null) return tv.visit(root);
        return null;
    }

    public TreeNodeStr<T> search(T data ) {
        TreeNodeStr<T> node = root;

        while (node != null && data != node.getData()){
            if (data.compareTo(node.getData()) > 0){
                node = node.getRight();
            } else {
                node = node.getLeft();
            }
        }
        return node;
    }

    public TreeNodeStr<T> insert(T data) {
        if(!isEmpty()){
            TreeNodeStr<T> parent = getRoot();
```

```
            TreeNodeStr<T> child;
            if (data.compareTo(parent.getData()) == 0) {
                return null;
            }
            if(data.compareTo(parent.getData()) < 0){
                child = parent.getLeft();
            } else {
                child = parent.getRight();
            }

            while(child != null){
                if(data.compareTo(child.getData()) < 0){
                    parent = child;
                    child = child.getLeft();
                }
                else if (data.compareTo(child.getData()) > 0) {
                    parent = child;
                    child = child.getRight();
                }
                else return null;
            }
            TreeNodeStr<T> node = new TreeNodeStr<T>(data);
            if(parent.getData().compareTo(data) < 0){
                parent.setRight(node);
            }
            else{
                parent.setLeft(node);
            }
            return node;
        }

        TreeNodeStr<T> node = new TreeNodeStr<T>(data);
        root = node;
        return node;
    }

}


------------------------------------------------------------
public class TreeNodeInt<T extends Integer> {
    private TreeNodeInt<T> left, right;
    private T data;

    public TreeNodeInt(T d){
        data = d;
```

```
    }

    public TreeNodeInt<T> getLeft(){
        return left;
    }

    public TreeNodeInt<T> getRight(){
        return right;
    }
    public void setLeft(TreeNodeInt<T> in){
        left = in;
    }
    public void setRight(TreeNodeInt<T> in){
        right = in;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}



------------------------------------------------------------
public class TreeNodeStr<T extends String> {
    private TreeNodeStr<T> left, right;
    private T data;

    public TreeNodeStr(T d){
        data = d;
    }

    public TreeNodeStr<T> getLeft(){
        return left;
    }

    public TreeNodeStr<T> getRight(){
        return right;
    }
    public void setLeft(TreeNodeStr<T> in){
        left = in;
    }
```

```
    public void setRight(TreeNodeStr<T> in){
        right = in;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}

-----------------------------------------------------------------
public interface TreeVisitorInt<T extends Integer, S> {
    public S visit(TreeNodeInt<T> node);
}

-----------------------------------------------------------------
public interface TreeVisitorStr<T extends String, S> {
    public S visit(TreeNodeStr<T> node);
}

-----------------------------------------------------------------
public class FindVisitorInt<T extends Integer, S extends Boolean>
            implements TreeVisitorInt<T,S> {
    private T _toFind;
    public FindVisitorInt(T toFind) {
        _toFind = toFind;
    }

    public S visit(TreeNodeInt<T> node) {
    if (_toFind.compareTo(node.getData()) == 0) {
        return (S) Boolean.TRUE;
    }
    if (node.getLeft() != null && visit(node.getLeft()) == Boolean.TRUE) {
        return (S) Boolean.TRUE;
    }
    if (node.getRight() != null) {
        return (S) visit(node.getRight());
    }
        return (S) Boolean.FALSE;
    }

}
-----------------------------------------------------------------
```

```
public class FindVisitorStr<T extends String, S extends Boolean>
            implements TreeVisitorStr<T,S> {
    private T _toFind;
    public FindVisitorStr(T toFind) {
        _toFind = toFind;
    }

    public S visit(TreeNodeStr<T> node) {
    if (_toFind.compareTo(node.getData()) == 0) {
        return (S) Boolean.TRUE;
    }
    if (node.getLeft() != null && visit(node.getLeft()) == Boolean.TRUE) {
        return (S) Boolean.TRUE;
    }
    if (node.getRight() != null) {
        return (S) visit(node.getRight());
    }
        return (S) Boolean.FALSE;
    }
}


-----------------------------------------------------------------
public class SizeVisitorInt<T extends Integer, S extends Integer>
            implements TreeVisitorInt<T,S> {

    public S visit(TreeNodeInt<T> node) {
        int size = 0;
        if (node.getLeft() != null) size += ((Integer)
            (visit(node.getLeft()))).intValue();
        if (node.getRight() != null) size += ((Integer)
            (visit(node.getRight()))).intValue();
        return (S) new Integer(size + 1);
    }

}
-----------------------------------------------------------------
public class SizeVisitorStr<T extends String, S extends Integer>
            implements TreeVisitorStr<T,S> {

    public S visit(TreeNodeStr<T> node) {
        int size = 0;
        if (node.getLeft() != null) size += ((Integer)
            (visit(node.getLeft()))).intValue();
        if (node.getRight() != null) size += ((Integer)
            (visit(node.getRight()))).intValue();
```

```
        return (S) new Integer(size + 1);
    }

}

-------------------------------------------------------------
import java.util.*;

public class TestFindVisitor {
    public static void main(String [] args) {
        Random r = new Random(10); // a random number generator with a fixed seed

        TreeInt<Integer> integers = new TreeInt<Integer>();
        TreeStr<String> strings = new TreeStr<String>();

        for (int i = 0; i < 70000; ++i) {
            int n = r.nextInt(10000000);
            integers.insert(new Integer(n));
            strings.insert(new String("" + n));
        }

        long time1, time2, time3;

        time1 = System.currentTimeMillis();

        FindVisitorInt<Integer,Boolean> fv1 = new FindVisitorInt<Integer,Boolean>
                (new Integer(5555));
        // visiting the tree of integers
        for (int i = 0; i < 10; ++i) {
            boolean found1 = ((Boolean) integers.accept(fv1)).booleanValue();
        }

        time2 = System.currentTimeMillis();

        FindVisitorStr<String, Boolean> fv2 = new FindVisitorStr<String, Boolean>("5555");
        // visiting the tree of strings
        for (int i = 0; i < 10; ++i) {
            boolean found2 = ((Boolean) strings.accept(fv2)).booleanValue();
        }

        time3 = System.currentTimeMillis();

        System.out.println("Time for integers " + (time2 - time1));
        System.out.println("Time for strings " + (time3 - time2));
    }
}
```

```
-------------------------------------------------------------
import java.util.*;

public class TestSizeVisitor {
    public static void main(String [] args) {
        Random r = new Random(10); // a random number generator with a fixed seed

        TreeInt<Integer> integers = new TreeInt<Integer>();
        TreeStr<String> strings = new TreeStr<String>();

        for (int i = 0; i < 70000; ++i) {
            int n = r.nextInt(10000000);
            integers.insert(new Integer(n));
            strings.insert(new String("" + n));
        }

        long time1, time2, time3;
        time1 = System.currentTimeMillis();

        // visiting the tree of integers
        SizeVisitorInt<Integer,Integer> sv1 = new SizeVisitorInt<Integer,Integer>();
        int size1 = ((Integer) integers.accept(sv1)).intValue();

        time2 = System.currentTimeMillis();

        // visiting the tree of strings
        SizeVisitorStr<String, Integer> sv2 = new SizeVisitorStr<String, Integer>();
        int size2 = ((Integer) strings.accept(sv2)).intValue();

        time3 = System.currentTimeMillis();

        System.out.println("Time for integers " + (time2 - time1));
        System.out.println("Time for strings " + (time3 - time2));
    }
}
```

Bibliography

[1] Gilad Bracha. Generics in the Java Programming Language. Sun Microsystems, java.sun.com, 2004.

[2] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. Design patterns: elements of reusable object-oriented software *Addison-Wesley Publishing Co., Inc.*, 1995.

[3] Kazuaki Ishizaki and Motohiro Kawahito and Toshiaki Yasue and Hideaki Komatsu and Toshio Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '00)*, 2000.