

Advanced Type Systems for Java in Real-World Programs

Sebastian Kanthak^{*}

ABSTRACT

I show that a high ratio of type-casts can be removed from a typical Java project by applying different type-system extensions. Especially the f-bounded parametric polymorphism and covariant return-types (both included in the upcoming Java 1.5 release) can be applied very successfully, but there remain typing-problems that can be better solved using different approaches.

1. INTRODUCTION

There have been a lot of proposals for extensions to Java's type system to make it more powerful and expressive allowing programmers to write more typeable and generic programs. However, this increase in flexibility does not come for free most of the time. It has to be traded off against runtime-safety (by allowing constructs that are not type-safe) or language simplicity (by making the type-system more complex) or both.

In order to be able to decide which type-system extensions are worth this cost it is important to look at real-world programs and analyze the potential benefit of each proposal in this programs. It is especially important to know how often a new technique can be applied in a sensible way. If a proposed extension increases the complexity of the type-system considerably but the problem it tries to solve does not appear frequently enough in most real-world programs it is questionable whether this extension would be worth the cost.

To identify all places in the source-code where an improved type-system could be applied I looked at all type-casts. A type-cast is an assertion to the type-checker that he can assume an expression to be of a given type. As this judgment might be incorrect and rely on informal assumptions,

^{*}This work was partly supported by the employees of the *Java Express* cafe with their great cappuccino where parts of this paper were written.

runtime-safety is reduced. Thus if this type-check could be removed by an improved type-system runtime safety would be increased as the type-checker would now guarantee this part to be free of type-errors. Additionally the added types could make the intent of the programmer more clear serving as an implicit form of documentation.

Advanced type-systems can also make the executing model of the language more powerful, allowing developers to write more generic code. As a complete redesign of a project would have been required to measure this, I did not include this in my considerations.

In section 2, I will develop a classification of type-casts used for further analysis that I will apply to a well-known Java project in section 3. In section 4, I will analyze the categories, present typical examples of type-casts encountered in these categories and discuss applicable type-system extensions. In section 5, I will present the results of this analysis and compare the trade-offs of the presented extensions. Finally, I will conclude and discuss related and future work.

2. CLASSIFICATION OF TYPE-CASTS

Most type-casts found in real-world programs can be classified into a small number of categories. In order to find these categories I asked the question what context the type-cast is in, why the programmer is sure that it will not fail at run-time and why it was necessary to use a type-cast to override the type-checker. Using this scheme, all type-casts of a given category can usually be handled by applying the same type-system extension.

However, the classification is not meant to cover all possible type-casts. It was developed by looking at type-casts in real-world programs, thus reflecting only the cases found in those programs.

Collections Java has a fairly complete collection api providing different kinds of containers. These containers are designed to store objects of type `Object` the superclass of all types. Thus putting objects into these containers does not require a type-cast but retrieving objects from these containers or iterating over objects requires type-casts as all these methods return `Objects`.

Heterogenous Collections When a container is used to store a heterogenous collection of objects, the objects have to be casted to their type when read from the

collection. Normally the programmer is sure about the type because of the position (when using `Lists`) or because of the key associated with an object (when using `Maps`).

Pattern A lot of patterns use generic *roles* to describe the different types of objects cooperating in the pattern. When designing a generic implementations of these patterns a class or interface is often used for each of these roles that is specialized when implementing a concrete instantiation of this pattern. In such an instantiation the programmer often has to cast object having a type of the generic role to the specific class used for that role in this pattern instantiation. Type-casts of this category also occur frequently in white-box frameworks that are used by subclassing some of the classes, as these frameworks use a lot of patterns.

Numbers When converting numbers (which are not treated as objects but as so called primitives in Java) to a type where a loss of precision might occur (e.g. `float` to `int`) this must be marked with a cast in Java. One can argue that this is not a real type-cast as it does not only give a hint to the type-checker but also includes some implicit conversion, for example when converting from IEEE 754 floating point representation to dual-complement integer representation.

Instance-Of Checks When dealing with heterogenous collections of objects one often wants to treat them in a different way depending on their runtime type. This can lead to a series of `instanceof` checks followed by corresponding type casts.

Cloning Java's `Object` class includes a `clone` method with return type `Object`. When cloning objects of a subclass of `Object` or delegating to parent `clone`-Methods one often has to cast the result to the static type of the receiver of the `clone` message.

Equals The `Object` class also includes an `equals` method to compare the object to another object. As it often only makes sense to compare objects of the same type one often has to cast the parameter in the implementation of the `equals` method.

Serialization Java contains classes to serialize and deserialize object graphs to and from byte-streams. When reading objects from byte-streams one usually has to cast them to their real type.

Reflection When using Java's reflection and introspection mechanisms (for example to create an object given its classname as a string) a cast is needed to cast the result to its type.

Unnecessary Some type-casts are simply not necessary because they are *identity type-casts* or are casting an expressing to a super-type of the statically determined type¹. These unnecessary type-casts often were required once and were not removed during a refactoring.

¹This sometimes is necessary to chose the correct method to call in the presence of method overloading

Category	Count	Percentage
Collections	50	23.36%
Heterogenous Collections	29	13.55 %
Pattern	53	24.77 %
Numbers	25	11.68 %
Instance-Of	15	7.01%
Cloning	8	3.74%
Equals	1	0.47%
Serialization	21	9.81%
Reflection	1	0.47%
Unnecessary	4	1.87%
Unsafe	6	2.80%
Other	1	0.47%
Total	214	100.00%

Figure 1: type-casts in JHotDraw

Unsafe Sometimes it is not clear why the programmer considered a type-cast to be safe. In this cases the safety depends on the fact that the code is only used in a specific way which, however, is not documented clearly.

3. ANALYSIS OF JHOTDRAW

JHotDraw is a "Java GUI framework for technical and structured Graphics" [6]. A simple drawing editor and an editor for petri nets are included as sample applications. The framework provides a lot of support to developers on the UI level (toolbars, handles for figure editing) as well as on the problem domain (different kind of figures, geometric primitives, animation).

It is considered as a very well designed piece of software, written by some highly regarded pattern experts and is often used for teaching and research purposes. I examined the source code of JHotDraw (version 5.2), identifying all type-casts and classifying them into different categories. The source code consists of 7,369 statements on 17,807 lines and there occur 214 type casts giving an average of 1.2 type-casts every 100 lines or 2.9 type-casts per 100 statements. The frequency of type-casts in different categories can be seen in figure 1.

4. TYPE-SYSTEM EXTENSIONS

Looking at the type-casts of the different categories I have described above reveals that most type-casts of one category can be treated uniformly. In the following I will give typical code-examples and show solutions using different type-system extensions. If possible, I always tried to implement the solutions to see if they really type-check and work. However, this was only possible if there was a working implementation of the extension for Java and my solution did not involve modifying the source of the Java library.

Most of the examples are drawn from the JHotDraw source code, but have been reduced to only contain the crucial details for brevity. Package names have been ignored as well and have to be inferred from the context by the reader.

4.1 Collections

A typical example for this category can be seen in listing 1. In the absence of type-system extensions the collections classes use `Object` as the type for their elements to allow developers to store all kinds of objects in them. While this

```

public abstract class CompositeFigure extends
    AbstractFigure /* ... */ {
    protected Vector fFigures;

    public Figure add(Figure figure) {
        if (!fFigures.contains(figure)) {
            fFigures.addElement(figure);
            figure.addToContainer(this);
        }
        return figure;
    }

    public void addAll(Vector newFigures) {
        Enumeration k = newFigures.elements();
        while (k.hasMoreElements())
            add((Figure) k.nextElement());
    }

    public Figure figureAt(int i) {
        return (Figure)fFigures.elementAt(i);
    }

    public boolean includes(Figure figure) {
        /* ... */
        FigureEnumeration k = figures();
        while (k.hasMoreElements()) {
            Figure f = k.nextFigure();
            if (f.includes(figure))
                return true;
        }
        return false;
    }
}

public final class FigureEnumerator implements
    FigureEnumeration {
    Enumeration fEnumeration;

    public Figure nextFigure() {
        return (Figure)fEnumeration.nextElement();
    }
}

```

Listing 1: Collections in Java 1.4

```

public abstract class CompositeFigure extends
    AbstractFigure /*...*/ {
    protected Vector<Figure> fFigures;

    public Figure add(Figure figure) {
        if (!fFigures.contains(figure)) {
            fFigures.addElement(figure);
            figure.addToContainer(this);
        }
        return figure;
    }

    public void addAll(Vector<Figure> newFigures) {
        Enumeration<Figure> k = newFigures.elements();
        while (k.hasMoreElements())
            add(k.nextElement());
    }

    public Figure figureAt(int i) {
        return fFigures.elementAt(i);
    }

    public boolean includes(Figure figure) {
        /* ... */
        for (Figure f : fFigures) {
            if (f.includes(figure))
                return true;
        }
        return false;
    }
}

```

Listing 2: Collections with *Generics* in Java 1.5

allows easy reuse it has a lot of disadvantages when only objects of one type are stored in the collection as in the `Vector` in the example, which only stores `Figure` objects. Storing objects into the collection is no problem, because in Java all classes inherit from `Object`. However, when reading objects as in the `figureAt` method, a type-cast to `Figure` is required. The reason why this cast is safe is, that only `Figures` have been stored in the vector, but this is neither captured nor enforced by the type-system.

The implementation of the `includes` method shows an interesting work-around used by the `JHotDraw` developers. Although this method iterates over all elements stored in the collection (thereby reading them) no type-cast is required in this method. The reason is, that the type-cast has been factored out into the `nextFigure` method of the `FigureEnumerator` class. This shows, that these type-casts were so common, that the developers felt the need to factor them out to make the code more readable. Another point illustrated by this example is, that the measure of type-casts introduced in section 3 is not entirely accurate and can easily be altered by these kind of refactorings.

A well-understood solution for this problem is the use of (F-bounded) parametric polymorphism as described in [1]. This extension will be included in the upcoming Java 1.5 release under the name *Generics*. It allows classes to be parameterized with type-parameters. Together with a modified collection library that makes use of these new possibilities the example can be rewritten as seen in listing 2².

²The syntax uses a new form of the `for`-statement to iterate

```

public abstract class AttributeFigure extends
    AbstractFigure {
    // uses HashMap for implementation
    private FigureAttributes fAttributes;

    public Object getAttribute(String name) {
        if (fAttributes != null) {
            if (fAttributes.hasDefined(name))
                return fAttributes.get(name);
        }
        return getDefaultAttribute(name);
    }

    public void setAttribute(String name, Object value) {
        /* ... */
        fAttributes.set(name, value);
        changed();
    }
}

public class TextFigure extends AttributeFigure /* ... */ {
    public void drawFrame(Graphics g) {
        /* ... */
        g.setColor((Color) getAttribute("TextColor"));
        /* ... */
    }
}

```

Listing 3: Type of attributes depends on String used as Key

Furthermore the compiler now enforces that only Figures can be stored in the vector and this fact is documented by the `Vector<Figure>` type.

I was able to eliminate all 50 type-casts of this category using a prototype of the new generics compiler that will be included in Java 1.5.

4.2 Heterogenous Collections

As explained in section 4.1 Java's collection classes are design to store instances of `Object` as their elements. This allows developers to store objects of different types in the same collection. A usage of this can be seen in listing 3. `AttributeFigure` uses a `HashMap` to store a mapping of `String` keys to objects. The actual type of the object depends on the key used for storing it. For example, an attribute stored under the key "TextColor" is assumed to be an instance of `Color`. There are other examples, where objects of different kinds are stored in a `List` and the actual type of the object depends on the position in the list.

As the common super-type of these objects is often `Object`, parametric polymorphism does not provide a solution for this. It seems, that this problem lies out of the scope of type-systems as the dependence on keys used to store the object or the position in a list is not expressible in a formal way.

In fact, it seems, that the type-casts were consciously accepted to gain the flexibility of being able to store an open-ended set of attributes of different kinds. Even without any over collections, which is syntactic sugar for a loop based on an `Iterator<Figure>`. Note that no type-casts are necessary and that the `FigureEnumerator` class is not needed anymore object

```

public class Geom {
    static public long length(int x1,int y1,int x2,int y2) {
        return (long)Math.sqrt(length2(x1, y1, x2, y2));
    }
}

```

Listing 4: Conversion between number representations

extensions to the type-system this could be solved in a type-safe way by adding the attributes as normal members to the appropriate classes. This would have the advantage of making it more apparent, which attributes are available. Currently, this can only be seen by inspecting the source code as there is only little documentation on this topic.

Another solution would be to have a separate collection per attribute type and then use parametric polymorphism to make this type-safe. While this would require the set of types used as attributes to be fixed it would still allow for an open-ended set of attributes.

This category seems to illustrate more of a design-problem than a type-problem. As the designers did not want to define a set of attributes at design-time they used a dynamic structure to store attributes, thereby moving the decision which attributes to store to run-time. This gives them more flexibility and makes incremental modifications to the figure-hierarchy easier, but essentially moves the problem out of the possibilities a type-checker has.

4.3 Numbers

A lot of type-casts deal with the conversion between different number representations during which a loss of precision might occur. The Java Language Specification [8] forces developer to indicate so called *narrowing conversions* with a type-cast in order to avoid unintended effects while conversions to types with a superset of values (*widening conversions*) are performed implicitly.

A lot of example of this can be seen in the `Geom` utility class of which a small part is shown in listing 4. This class performs a number of primitive geometric operations like computations with angles, distance between points and line segment intersection. In the `length` method, an expression of type `double` is casted to `long` thereby truncating the value to an integer.

I claim that this is not really a type-cast (or at least does not indicate a type-problem) because it does not only give the type-checker the information, that this expression can be assumed to be of type `long`, but also forces the compiler to insert operations to convert from IEEE754 floating point representation to a signed-integer 2-complement representation.

One could argue, that this difference should be reflected by the syntax as well, for example by choosing a function-like syntax `long(Math.sqrt(...))` like in Python [13]. This would make it more visible that a real conversion operation is performed here.

4.4 Instance-Of

```

public class FigureAttributes implements /* ...*/
    Serializable {
    public void write(StorableOutput dw) {
        /* ... */
        if (v instanceof String) {
            dw.writeString("String");
            dw.writeString((String) v);
        } else if (v instanceof Color) {
            writeColor(dw, "Color", (Color)v);
        } else if (v instanceof Boolean) {
            dw.writeString("Boolean");
            if (((Boolean)v).booleanValue())
                dw.writeString("TRUE");
            else
                dw.writeString("FALSE");
        } /* ... */ else {
            System.out.println("Unknown_attribute:_" + v
                );
            dw.writeString("UNKNOWN");
        }
    }
}

```

Listing 5: Multiple instanceof tests followed by casts

Java includes the `instanceof` operator that can be used to test whether an object is an instance of a given type (at runtime). Often such a runtime test is directly followed by a type-cast to the corresponding type as can be seen in listing 5. It is obvious that this cast will not fail at runtime, because of the preceding `instanceof`-check.

This gives rise to some language construct that allows this fact to be captured by the type-system by providing a *test-and-bind* mechanism. JMatch[9] extends the `switch`-statement built into Java in a way that makes it possible to have *case*-statements for different types that automatically bind the object to a new variable of the type, as can be seen in listing 6. This proposal has the disadvantage of inheriting Java's fall-through-behavior for switch statements which does not integrate nicely with the scope of the new variable. From my experiments with JMatch it seems, that the scope of a variable defined in a *case*-statement is extended until the next `break`-statement. However, a nicer `typeswitch` construct is easily imaginable that uses curly braces for the scope of bindings and could look like the one in listing 7

Such an extension is particularly useful for code that has to handle objects in a different way depending on their dynamic type. Readers familiar with design patterns[5] will immediately think of the Visitor-Pattern that is designed exactly for this purpose. I claim, however, that there are legitimate reasons to use an *if*- or *typeswitch*-based approach. One such reasons might be the added complexity and reduced flexibility of the Visitor-Pattern, another the fact, that in order to apply the Visitor-Pattern in the above example one would have to modify the `Integer` and `String` classes³ which are part of the standard library and therefore immutable.

In theory one could replace all type-casts with such a type-safe `instanceof`-variation, thereby getting rid of all type-

³to include an `accept` method into a common superclass for the double-dispatch emulation

```

public class FigureAttributes implements /* ...*/
    Serializable {
    public void write(StorableOutput dw) {
        /* ... */
        switch (v) {
        case String s: // s has static type String
            dw.writeString("String");
            dw.writeString(s); break;
        case Color c: // c has static type Color
            writeColor(dw, "Color", c); break;
        case Boolean b: // b has static type Boolean
            dw.writeString("Boolean");
            if (b.booleanValue())
                dw.writeString("TRUE");
            else
                dw.writeString("FALSE");
            break;
        /* ... */
        default:
            System.out.println("Unknown_attribute:_" + v
                );
            dw.writeString("UNKNOWN");
        }
    }
}

```

Listing 6: Case-statements with types

```

typeswitch (v) {
    case (Integer b): {
        // b has type Integer in this block
        /* ... */
    }
    case (String b): {
        // b has type String in this block
        /* ... */
    }
}

```

Listing 7: Typeswitch mechanism with clearly defined scope for bindings

```

public class CreationTool extends AbstractTool {
    private Figure fPrototype;

    protected Figure createFigure() {
        /* ... */
        return (Figure) fPrototype.clone();
    }
}

```

Listing 8: Using `clone` method in client code

```

public class FigureAttributes /* ... */ {
    private Hashtable fMap;

    public Object clone() {
        /* ... */
        FigureAttributes a = (FigureAttributes) super.clone();
        a.fMap = (Hashtable) fMap.clone();
        return a;
    }
}

```

Listing 9: Using `super.clone()` in implementation of `clone` method

casts. Obviously this would increase the safety of the program by no means. Even worse, the opposite is true: By not throwing an appropriate `ClassCastException` when an `instanceof`-check fails, runtime-errors are hidden. The problem with this mechanism is that in contrast to functional languages the set of types is open, which means that the compiler can never enforce that all possible cases have been listed in a `typeswitch` statement. One therefore has to consider very carefully, if this mechanism or a standard type-cast is the right choice.

It turns out, that the possibility to use types in `switch` statements is only a minor addition of the `JMatch` proposal. In fact, it adds a full-featured pattern-matching mechanism similar to the one found in functional languages like Haskell or ML. However, I did not find a use-case for this in `JHot-Draw`.

4.5 Cloning

Java's `Object` class which is the superclass of all objects contains a method with signature `Object clone()`. The default behavior of this method is to create a shallow copy of the object it is called on. This means the actual type of the returned object is the same as the dynamic type of the object it is called on. As the method's declared return type, however, is only `Object` and return types cannot be specialized in overridden methods in Java 1.4, type-casts become necessary when calling the `clone` method. Upon closer inspection, they can be divided into two categories:

1. Type-casts of clients using the `clone` method to cast the result to the static type of the receiver of the message. This is safe, as the dynamic type of the receiver (which will be the type of the result) is always a subtype of the static type.

An example of this can be seen in listing 8, where the Prototype-Pattern is used to create figures with a

```

public interface Figure /* ... */ {
    public Figure clone();
}

public class CreationTool /* ... */ {
    private Figure fPrototype;

    protected Figure createFigure() {
        /* ... */
        return fPrototype.clone();
    }
}

```

Listing 10: Refining return type of `clone` in a covariant way

creation tool. After cloning the prototype, a cast to `Figure` is necessary.

2. Type-casts in implementations of the `clone` methods. When the desired behavior of the cloning operation is a shallow copy, there is no need to override the `clone` method. If, however, something else is required — for example a deep copy or a copy of some members — one has to override it. The typical implementation of such an overridden `clone` method looks like in listing 9. First `super.clone()` is called and the result has to be casted to the type of the class the overridden `clone` method is in. Then, some additional operations on the result of the `super`-call can be performed. Note that in the example, this involves another cloning-operation, this time as a client as described in case 1.

A partial solution for the usage of the `clone` method in client code is possible with Java 1.5 as it allows return-types of methods to be refined in a covariant way which is type-safe. Using this feature the signature of the `clone` method can be overridden in the `Figure` interface as shown in listing 10. The cloning operation in `CreationTool` does not require a type-cast anymore because the prototype is known to be of type `Figure`.

Of course, this requires every implementation of the `Figure` interface to override the `clone` method to match the refined signature, although one normally would not need to do so because the inherited implementation of this method works fine. If subclasses do not refine the method further the benefits are lost again, e.g. when cloning a `PolygonFigure` a cast is required as the result is only known to be of type `Figure`. Even worse, the implementation of the `clone` method still requires a type-cast as the result of the call to `super.clone()` only has the static type of the superclass. Thus one can see this solution as a factoring out of type-casts very much like the `FigureEnumeration` example from section 4.1, which is useful if there are a lot of clients cloning objects.

A better solution is possible if one uses a `ThisType`-extension as proposed by Bruce [2]. It exploits the fact that the type-casts described above always cast the return value to the static type of the receiver. This can be expressed in the extended type-system with a modified method signature as shown in listing 11. `ThisType` refers to the dynamic type of the object it is used in and is replaced by the static type of the receiver when type-checking method invocations.

```

public class Object /* ... */ {
    public ThisType clone();
}

public class CreationTool /* ... */ {
    private Figure fPrototype;

    protected Figure createFigure() {
        /* ... */
        return fPrototype.clone();
    }
}

public class FigureAttributes /* ... */ {
    private Hashtable fMap;

    public ThisType clone() {
        /* ... */
        ThisType a = super.clone();
        a.fMap = fMap.clone();
        return a;
    }
}

```

Listing 11: Type-safe clone method with ThisType

Thus when typechecking `fPrototype.clone()` the type of the return value is determined to be the static type of `fPrototype` which is `Figure` as desired. Note that for this solution it is not necessary to override the `clone` method.

This also solves the problem with type-casts in the implementation of `clone` methods because when typechecking the method call `super.clone()`, `super` can be assumed to be of the `ThisType` type. Furthermore, as `ThisType` is a sub-type of the type currently defined, the assignment to the attribute `fMap` is safe.

Note, that *exact types* as proposed in the extensions are not necessary because `ThisType` only appears as the return-type of methods. If one wants to call methods that take parameters of type `ThisType` (so called *binary* methods), however, one has to introduce further concepts (like exact types) because method-parameters can now vary in a covariant way.

4.6 Equals

The `equals` method is sometimes called a prototypical example for binary methods, because the argument should be of the same type as the receiver, resulting in the following method signature, using Bruce's proposal again:

```
boolean equals(ThisType that);
```

In order for method calls to this method to be type-safe, they can only be performed on objects when the exact type of the receiver is known statically, i.e. it is confirmed that the object is not actually a subtype of its static type. An example of this can be seen in listing 12.

However, there are good arguments against this method signature for `equals`. Depending on the desired semantics for equality and the problem domain it can make sense to compare objects of different kind, for example when dealing with the Proxy-Pattern or with numbers represented as either fixed-decimal numbers or fractions. This could be solved by

```

public class A {
    int a;
    public boolean equals(ThisType that) {
        return this.a == that.a;
    }
}

public class B {
    int b;
    public boolean equals(ThisType that) {
        return this.b == that.b && this.a == that.a;
    }
}

/* ... */
@A a1 = new A(42);
A a2 = new A(42);
@B b1 = new B(42,23);
B b2 = new B(42,23);
a1.equals(a2); // true
b1.equals(b2); // true
a1.equals(b1); // static error
b1.equals(a1); // static error
a2.equals(a1); // static error

```

Listing 12: Type-safe binary equals method

```

public interface Comparable<A> {
    public boolean equals(A that);
}

public class A implements Comparable<A> {
    int a;
    public boolean equals(A that) {
        return this.a == that.a;
    }
}

public class B implements Comparable<B> {
    int b;
    public boolean equals(B that) {
        return this.b == that.b && this.a == that.a;
    }
}

/* ... */
A a1 = new B(42,1);
A a2 = new B(42,9);
a1.equals(a2); // returns true

```

Listing 13: equals method with parameterized interface

```

public class StandardStorageFormat implements
    StorageFormat {
    public boolean equals(Object compareObject) {
        if (compareObject instanceof StandardStorageFormat
            ) {
            return getFileExtension().
                equals(((StandardStorageFormat)
                    compareObject).getFileExtension());
        }
        else {
            return false;
        }
    }
}

```

Listing 14: Implementation of `equals` in `JHotDraw`

letting objects implement the parameterized `Comparable<A>` interface of listing 13⁴. One disadvantage is that one has to state for a lot of classes that they implement an instantiation of this (template) interface. What is even worse is, that the `equals` methods are all different and are not treated polymorphically. This is illustrated in the listing by the call to `equals` in the listing that returns `true` where one would expect `false`.

Another implementation strategy for the `equals` method is to test whether the argument is of a type comparable to this object and then perform either a type-cast or return `false`. In fact, only one method in `JHotDraw` overrides the `equals` method at all and it uses this strategy. This is shown in listing 14. A type-safe `instanceof` as describe in section 4.4 would probably be the best solution here.

4.7 Serialization

Java has a built-in mechanism to serialize object-graphs into streams and later deserialize them. This mechanism is used in `JHotDraw` to store drawings into files and to perform clipboard operations. When deserializing object-graphs a type-cast is needed. I argue that this type-cast is unavoidable as the source of the objects is not analyzable by the type-checker as it may change at run-time. It is for example possible to instruct `JHotDraw` to load a file that does not hold a serialized `Drawing` object, but a simple `String` and the type-checker has now way to prevent this. Thus, a run-time check (as performed by a type-cast) is the best we can do in this situation.

4.8 Reflection

A similar argument as in section 4.7 can be made about the type-casts needed when using Java's reflection mechanism. For example, when creating an object via `Class.forName(String)` the type depends on the argument string and the type-checker has no possibility to prove anything about the contents of this string in general. Thus, I consider these type-casts as unavoidable as well.

4.9 Patterns

⁴Because of technical difficulties with the type-erasure implementation strategy in Java 1.5 this listing is not compilable there at all. After type-erasure, the two `equals` methods have the same signature, resulting in a compile-time error

```

public interface Figure /* ... */ {
    public Vector<Handle> handles();
}

public interface Handle {
    public Point locate();
    public Figure owner();
}

public abstract class AbstractHandle implements Handle {
    private Figure fOwner;

    public Figure owner() {
        return fOwner;
    }
}

public class ElbowHandle extends AbstractHandle {
    public ElbowHandle(LineConnection owner /* ... */) {
        super(owner);
        /* ... */
    }

    public Point locate() {
        LineConnection line = ownerConnection();
        /* ... */
    }

    private LineConnection ownerConnection() {
        return (LineConnection)owner();
    }
}

```

Listing 15: Adapter pattern for handles

`JHotDraw` uses a lot of design patterns [5] to gain flexibility in the sense, that behavior can be changed incrementally or at run-time. Design patterns often use reference- and subtype-relations simultaneously to reach this goal, resulting in families of collaborating classes. A closer inspection of the reasons for type-casts in this category revealed two different sub-categories for which different solutions were needed:

1. Generic implementations of (parts) of design patterns that are specialized by sub-classing. Type-casts are necessary because references to other objects participating in the pattern have a general type from the generic implementation but the more specific type is required.
2. Specific relationships between the classes collaborating in the design pattern that can not be expressed in the type-system. This often manifests in the covariance problem: Method arguments that vary in a covariant way but cannot be refined because they have already been specified in a super-type.

A good example of the first category can be seen in the implementation of the Adapter-Pattern in the `Handle` class. Handles are little boxes that can be used to modify a figure in the graphical editor. As shown in listing 15 a figure can return a collection of handles and handles have a method to locate themselves so that the framework knows, where to draw a handle. Note that a handle knows its owning figure because of the `owner()` method in the interface.

```

public abstract class AbstractHandle<FigureType extends
    Figure>
    implements Handle {
    private FigureType fOwner;

    public AbstractHandle(FigureType owner) {
        fOwner = owner;
    }

    public FigureType owner() {
        return fOwner;
    }
}

public class ElbowHandle extends AbstractHandle<
    LineConnection> {
    public ElbowHandle(LineConnection owner /* ... */) {
        super(owner);
        /* ... */
    }

    public Point locate() {
        LineConnection line = owner();
        /* ... */
    }
}

```

Listing 16: Type-safe handle implementation by parameterizing type of owner

In the class `AbstractHandle` the common code to store the owner is implemented so that it does not have to be repeated in all handle implementations. However, this makes a type-cast in a concrete handle implementation necessary, that needs access to special attributes of its associated figure in order to locate itself on this figure. In `ElbowHandle` the familiar strategy from section 4.1 to factor out a common type-cast is used, but the type-cast is still there. Note, that in the constructor of `ElbowHandle` the correct type of the owner is still known, but this information gets lost when using the inherited `owner` method. If `AbstractHandle` had not been used, but the handling of the owner had been implemented by every handle again, this problem would not have been arisen.

This suggests a solution with parametric types where the class `AbstractHandle` is parameterized with the type of the owner. In this solution in listing 16 the return-type of the `owner` method is refined in a covariant way, but it still implements the `Handle` interface because of the bound on the type parameter. With this solution different handle implementations that have different kinds of owners are no longer subtypes of them common supertype `AbstractHandle`, because the instantiations of the type-parameters differ. Nevertheless this is not a problem in this case because handles do not inherit from `AbstractHandle` because they want to be a subtype of it, but because they want to reuse and extend its implementation. The important subtype-relation is that all handles are a subtype of `Handle` which is preserved by this solution.

This implies that the above approach would not have worked if `Handle` had been a parameterized abstract class that implements the owner-handling like `AbstractHandle` does. In this case, there would be no suitable return-type for `Figure's`

```

public abstract class PaletteButton /* ... */ {
    private PaletteListener fListener;

    public PaletteButton(PaletteListener listener) {
        fListener = listener;
        /* ... */
    }

    public void mouseMoved(MouseEvent e) {
        fListener.paletteUserOver(this, true);
    }
}

public interface PaletteListener {
    void paletteUserOver(PaletteButton button, boolean
        inside);
}

public class ToolButton extends PaletteButton {
    public ToolButton(PaletteListener listener /* ... */) {
        super(listener);
        /* ... */
    }
}

public class DrawApplication /* ... */ implements
    PaletteListener {
    protected ToolButton createToolButton(/* ... */) {
        return new ToolButton(this /* ... */);
    }

    public void paletteUserOver(PaletteButton button,
        boolean inside) {
        ToolButton toolButton = (ToolButton) button;
        /* ... */
    }
}

```

Listing 17: Covariance problem using listeners

`handles` method. Especially `Handle<Figure>` would not work because `Handle<LineConnection>` is not a subtype of it. A solution to this is given by a proposal to add variance to parametric types by Igarashi and Viroli [7]. Using their notation the appropriate return type would be `Handle<+Figure>`, indicating that the type parameter can be a subtype of `Figure`. To make this type-safe the type-checker would prohibit all method calls on objects of this type that contain the type-parameter as an argument. As there are no such method signatures in `Handle` this would not be a restriction.

A more difficult problem that arises in the context of listeners (also called Subject-Observer-Pattern) can be seen in listing 17. `PaletteButton` is an abstract class that uses a listener object passed in at construction time to inform interested parties about events, passing `this` as an argument to allow for object that are listeners for several buttons. `DrawApplication` implements the listener interfaces and creates some buttons that are instances of a subclass of `PaletteButton`. In the implementation of the methods from the listener interface, a type-cast is required. This cast is safe, because the class only creates instances of `ToolButton` and no other classes register `DrawApplication` as a listener for other buttons. Intuitively an interface `ToolButtonListener` would be needed, where the method argument is refined. However, this interface would not be a subtype of the original `PaletteButtonListener` interface, which would make pass-

```

public interface PaletteListener<ButtonType> {
    void paletteUserOver(ButtonType button, boolean inside)
    ;
}

public abstract class PaletteButton<ButtonType> /* ... */
{
    private PaletteListener<ButtonType> fListener;
    public PaletteButton(PaletteListener<ButtonType>
        listener) {
        fListener = listener ;
        /* ... */
    }

    public void mouseMoved(MouseEvent e) {
        fListener.paletteUserOver(mythis(), true);
    }

    public abstract ButtonType mythis();
}

public class ToolButton extends
    PaletteButton<ToolButton> {
    public ToolButton(PaletteListener<ToolButton> listener
        /* ... */) {
        super(listener);
        /* ... */
    }

    public ToolButton mythis() {
        return this;
    }
}

public class DrawApplication /* ... */
    implements PaletteListener<ToolButton> {

    protected ToolButton createToolButton(/* ... */) {
        return new ToolButton(this /* ... */);
    }

    public void paletteUserSelected(ToolButton toolButton) {
        /* ... */
    }
}

```

Listing 18: Encoding class families in Generics

ing it to the constructor of `PaletteButton` impossible.

A solution with parametric polymorphism is still possible and shown in listing 18. The listener interface is parameterized with the type of the button. As the `PaletteButton` class takes an interface to such a listener, it has to be parameterized as well. Now, the problem is that in the invocation of the listener method `this` does not have the correct type `ButtonType`. This is solved by introducing an abstract method `mythis` that has to be overridden in subclasses and provides a `this` reference with the desired type.

This solution seems very unnatural and complex (note the instantiation of type parameters in the definition of the `ToolButton` with itself), so I consider it more as a “hack” than as a real solution. Additionally it has the disadvantage of loosing subtype relations. One cannot make subclasses of `ToolButton` if a refined listener type is needed and the different button classes are no longer in a subtype relation. Although this is not needed in `JHotDraw` one can imagine

```

public class ButtonFamily {
    public interface ButtonListener (ListenerType) {
        public void paletteUserOver(ButtonType button,
            boolean inside);
    }
    public class Button (ButtonType) {
        private @ListenerType fListener;
        public Button(@ListenerType listener) {
            fListener = listern ;
            /* ... */
        }
        public void mouseMoved(MouseEvent e) {
            fListener.paletteUserOver(this, true);
        }
    }
}

public class ToolButtonFamily extends ButtonFamily {
    public class ToolButton (ButtonType) extends Button {
        /* ... */
    }
}

```

Listing 19: Virtual classes and exact types

code that has to handle different button classes in a generic way. This would not be possible with this solution.

The underlying problem is that of a family of classes (button and listener in this case) that have to be extended in parallel. Virtual classes as proposed for Java in [10] have proven to naturally express this. There are currently two proposals to make virtual types statically safe: Bruce’s generalization of his `ThisType` mechanism [4] and virtual classes with dependant types [12]. I will show how both can be used to type the above problem.

Virtual classes are nested inside classes and can then be refined in subclasses. As shown in listing 19 this relation is expressed in Bruce’s proposal by appending a *type variable name* in parentheses. In `ToolButtonFamily` the argument type of the listener method is `ToolButton` as this is the most specific definition of the `ButtonType` type variable. Note the usage of exact types (indicated by a `@` in front of the type) that is necessary to successfully type-check invocations of the listener method in `Button`. As the proposal is only informally introduced in [4] and currently no implementation exists, it remains somewhat unclear where a concrete listener implementation would have to be defined. As the `ListenerType` identifier is only valid in the scope of the family class it might be necessary to define it in this scope. Generally the scope of these type variable identifiers and the restrictions of exact types are disadvantages of this proposal.

When adding *dependent types* these problems can be solved, leading to a more expressive type system and cleaner syntax and scoping. Dependant types are types that depend on the identity of an object, capturing that the actual class of a virtual class depends on the dynamic type of the object it is contained in. Using a syntax adopted from `FamilyJ` [14] the example is shown in listing 20.

This example uses nested virtual classes as before, but now they do not get two different names, but simply are defined

```

public class ButtonFamily {
    public virtual class Button {
        private this(ButtonFamily).Listener fListener;

        public Button(this(ButtonFamily).Listener listener) {
            fListener = listener;
            /* ... */
        }

        public void mouseMoved(MouseEvent e) {
            fListener.paletteUserOver(this, true);
        }
    }

    public abstract virtual class Listener {
        public void paletteUserOver(this(ButtonFamily).
            Button button,
                                   boolean inside);
    }
}

public class ToolButtonFamily {
    override class Button {
        // ToolButton stuff
    }
}

/* ... */
public class DrawApplication {
    final ToolButtonFamily tbf = new ToolButtonFamily();

    class ToolButtonListener extends tbf.Listener {
        public void paletteUserOver(tbf.Button button,
                                    boolean inside) {
            /* ... */
        }
    }

    tbf.Listener listener = new ToolButtonListener();

    protected tbf.ToolButton createToolButton(/* ... */) {
        return new ToolButton(listener /* ... */);
    }
}

```

Listing 20: Virtual classes with dependent types

```

public interface Locator /*... */ {
    public Point locate(Figure owner);
}

public class PolygonHandle extends AbstractHandle {
    private Locator fLocator;

    public PolygonHandle(PolygonFigure owner, Locator l, int
        index) {
        super(owner);
        /* ... */
    }

    public Point locate() {
        return fLocator.locate(owner());
    }
}

public class PolygonFigure extends AttributeFigure {
    public Vector handles() {
        Vector handles = new Vector(fPoly.npoints);
        for (int i = 0; i < fPoly.npoints; i++)
            handles.addElement(new PolygonHandle(this, locator(i)
                , i));
        /* ... */
    }

    public static Locator locator(final int pointIndex) {
        return new AbstractLocator() {
            public Point locate(Figure owner) {
                PolygonFigure plf = (PolygonFigure)owner;
                /* ... */
            }
        };
    }
}

```

Listing 21: Strategy pattern with class families

with a name and are overridden in subclasses. To illustrate that these types depend on the object they are contained in they are always prefixed with the corresponding object⁵. Note that the field `tbf` is declared `final` so that the type-checker can infer statically that `tbf.Listener` matches the type declared in the constructor of `Button`. If `tbf` was not a constant reference nothing could be inferred about it.

Note that in both solutions the tool button family is a subtype of the standard button family. Thus it is possible to write code that works on any kind of button, although the exact types make this more difficult than the more elegant dependent types. All type-safe solutions have the advantage of not allowing other classes to register `DrawApplication` as a listener for incompatible button types.

As a last example I want to give something where I did not find a solution with Java 1.5 at all. It arises in the context of a Strategy-Pattern and is shown in listing 21

In this code `PolygonFigure` uses an anonymous implementation of the `Locator` interface to encapsulate the strategy to locate the points where to draw handles. Note that a

⁵`this(ButtonFamily)` designates the `this` member of the enclosing object of type `ButtonFamily`. It would not be necessary when using implicit scoping and has only been included to point on the dependence of types on objects

```

public class FigureFamily {
    public abstract virtual class Figure {
        public Vector handles();
    }

    public abstract virtual class Locator {
        public Point locate(Figure);
    }

    public abstract virtual class Handle {
        Figure owner;
        public Point locate();
    }

    public abstract class LocatorHandle extends Handle {
        Locator locator;
        public Point locate() {
            locator.locate(owner);
        }
    }
}

public class PolygonFigureFamily extends FigureFamily {
    override class Figure {
        public Vector handles() {
            /* ... */
            new LocatorHandle(this, new PolygonLocator());
            /* ... */
        }
    }

    class PolygonLocator extends Locator {
        public Point locate(Figure owner) {
            // can access members specific to
            // PolygonFigureFamily's Figure
        }
    }
}

```

Listing 22: Strategy pattern with virtual classes

`Locator` does not have an owner, so that the solution for `Handles` from listing 16 is not applicable here. Instead of giving every locator an owner it is passed as an argument to allow locators to be shared between figures. The invariant that makes the type-cast in the locator implementation safe is, that it is always called with a figure of the same type that created it.

In this example, three classes form a class family that should be refined in parallel. This can be done nicely with virtual classes with dependent types as shown in listing 22. Note the generic implementation of `LocatorHandle` that can be used by all figure families.

5. BENEFITS AND COSTS

For each of the category I counted the number of type-casts that I removed successfully with a certain extension. If a working implementation of the extension was available I used it to verify that the solutions do actually type-check and that `JHotDraw`'s examples still run. The results can be seen in figure 2. This count however is biased in a way that for most of the categories I only tried different approaches if it could not be solved with the extensions found in Java 1.5. This reflects the goal of keeping the complexity low and only using easy to understand extensions. As parametric polymorphism is well-understood and familiar to most

= Number of type-casts
 1.4 = solved with Java 1.4
 1.5 = solved with Java 1.5
 JM = solved with JMatch
 TT = solved with ThisType
 VC = solved with Virtual Classes

Category	#	1.4	1.5	JM	TT	VC
Collections	50	0	50	0	0	0
Heterogenous C.	29	0	0	0	0	0
Pattern	53	0	44	0	0	6
Numbers	25	0	0	0	0	0
Instance-Of	15	0	0	9	0	0
Cloning	8	0	3	0	8	0
Equals	1	0	0	1	0	0
Serialization	21	0	0	0	0	0
Reflection	1	0	0	0	0	0
Unnecessary	4	4	0	0	0	0
Unsafe	6	0	0	0	0	0
Other	1	0	0	0	0	0
Total	214	4	97	10	8	6

Figure 2: Results of applying various type-system extensions to `JHotDraw`

developers, it is justified to prefer this approach over others that are more complex or unfamiliar.

The numbers clearly show that the extensions included in Java 1.5 are a big step forward regarding the possibility to write type-safe code while maintaining good design. Even applying parametric polymorphism only to the collections framework seems worth the added complexity as the use of this framework is prevalent in Java code. However, the numbers show that it can be used successfully in other categories like design patterns as well.

Given the numbers it seems that a type-safe `instanceof` would be an useful addition supporting common Java coding style, but the full-featured pattern matching from the `JMatch` proposal is clearly unnecessary and does not fit well with object-oriented programming and their open subtypes.

The `ThisType` construct is interesting, but the added complexity — especially if exact types are used — does not seem worth the benefit of 5 fewer type-casts that could not be removed with the covariant return-types that are included in Java 1.5. Without exact types this extensions seems to be pretty much limited to the `clone`. Here, the most pragmatic approach of factoring out the type-casts into the `clone` method and relying on covariant return-types seems to be the best solution.

Virtual classes have proven to be a very interesting concept that lend themselves to natural solutions for otherwise difficult typing problems and support more generic code through their richer subtype relation. However, the numbers raise the question if this is worth the introduction of a new and rather complicated type-system.

In [11] Thorup shows that most usages of parametric polymorphism (and all of those that appear in this figure) can be replaced by virtual classes if they are extended by a form of structural subtyping. In the light of these results, virtual

classes with dependent types become a mechanisms that seems to be even more powerful than f-bounded parametric polymorphism. It has the advantage of fitting closer into the object-oriented ideas in the sense that virtual classes are seen as members of objects whereas parametric polymorphism can be seen as being closer to functional languages because parametric types are essentially functions mapping types to types. Another advantage of virtual classes is that their richer subtype-relation allows for more generic code, although this was not visible in the JHotDraw framework. Perhaps a complete redesign of the JHotDraw framework could take advantage of this property but this was out of the scope of this work.

However, at the current stage I know of no implementation of virtual classes with dependent types for Java and the concept is still very new and basic questions like decidability of the type-system are still open. Given the large number of cases that can also be handled with parametric polymorphism, it seems to be the right choice to include it (under the name Generics) in Java 1.5.

6. RELATED AND FUTURE WORK

Most type-system proposals include examples to prove their proposal to be useful. While these examples always show the positive aspects of the type-system they often do not relate the cost of added complexity to the frequency of the problem they are trying to solve in real code.

Bruce [3] describes some challenging typing problems for object-oriented languages but the examples are constructed to illustrate his points and are not taken from real-world programs.

Bruce and Thorup have compared virtual classes and parametric polymorphism in [4, 11] and gave constructed examples that demonstrate their respective strength and weaknesses.

Future work in the field of virtual classes and dependent types seems to be promising as it could lead to a more powerful type system that is able to replace parametric polymorphism in object-oriented languages.

A simple typesafe instance-of test that integrates nicely into the Java languages and its scoping rules would support developers and could be adopted in future Java versions.

To measure the real complexity versus usefulness of type-systems, one could perform studies on developers where they have to perform certain programming tasks in the style of user studies as performed in Human-Computer-Interaction.

7. CONCLUSION

I have shown that the type-extensions that will be included in Java 1.5 are powerful enough to remove a big portion of the type-casts in a typical Java program. However, especially in the context of patterns, a lot of typing problems remain open that could be solved better with approaches based on virtual classes.

The comparison of parametric polymorphism and virtual classes showed that the former is useful for abstractions

that are made to share implementation between classes while the latter has advantages in expressing subtype-relations between families of classes.

In areas that rely on Java's runtime-abilities to test if an object is an instance of a given type, a simple extension has proven to be useful in supporting common idioms to handle objects based on their dynamic type.

8. REFERENCES

- [1] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.
- [2] K. B. Bruce. Increasing java's expressiveness with thistype and match-bounded polymorphism. Technical report, Williams College, 1997.
- [3] K. B. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 82, 2003.
- [4] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. *Lecture Notes in Computer Science*, 1445:523–??, 1998.
- [5] R. J. Erich Gamma, Richard Helm and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
- [6] E. Gamma and T. Eggenschwiler. Jhotdraw. <http://www.jhotdraw.org>.
- [7] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'02)*, 2002.
- [8] B. J. James Gosling and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [9] A. C. M. Jed Liu. Jmatch: Iterable abstract pattern matching for java. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, 2003.
- [10] K. K. Thorup. Genericity in Java with virtual types. *Lecture Notes in Computer Science*, 1241:444–??, 1997.
- [11] K. K. Thorup and M. Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. *Lecture Notes in Computer Science*, 1628:186–??, 1999.
- [12] M. Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, San Diego, CA, USA, 1998.
- [13] G. van Rossum. *The Python Language Reference Manual*. Network Theory Ltd, 2003.
- [14] A. Wittmann. Towards caesar: Family polymorphism for java. Master's thesis, University of Technology, Darmstadt, 2003.