

Advanced Type-Systems in Real-World Java Programs

Sebastian Kanthak

November 27, 2003

Motivation

Proposed extensions to Java's type system:

- ▶ Parametric Polymorphism (Generics)
- ▶ Virtual Types
- ▶ ThisType
- ▶ Family Polymorphism
- ▶ Mixins
- ▶ Virtual Classes with Dependant Types
- ▶ ...

Solution

Parametric Virtual Classes with Dependant Familie of Mixins?

Problem

Solution

- Proposed Methodology
- Classification of Type-Casts
- Code Examples

Conclusion

Type-Systems are complex

- ▶ more flexible type-systems are more complex
- ▶ difficult for programmers
- ▶ difficult to combine

What is really needed?

- ▶ type-casts show failures of type-systems
- ▶ analyse type-casts in real-world-programs
- ▶ evaluate type-systems

Project to analyze

requirements:

- ▶ open-source
- ▶ good-design

chosen project: JHotDraw

Project to analyze

requirements:

- ▶ open-source
- ▶ good-design

chosen project: JHotDraw

Type-Casts in JHotDraw

Some figures:

- ▶ 17,807 lines
- ▶ 7,369 statements
- ▶ 214 type-casts
- ▶ 1.2 type-casts per 100 lines
- ▶ 2.9 type-casts per 100 statements

Type-Casts in JHotDraw

Some figures:

- ▶ 17,807 lines
- ▶ 7,369 statements
- ▶ 214 type-casts
- ▶ 1.2 type-casts per 100 lines
- ▶ 2.9 type-casts per 100 statements

Type-Casts in JHotDraw

Some figures:

- ▶ 17,807 lines
- ▶ 7,369 statements
- ▶ 214 type-casts
- ▶ 1.2 type-casts per 100 lines
- ▶ 2.9 type-casts per 100 statements

Type-Casts in JHotDraw

Some figures:

- ▶ 17,807 lines
- ▶ 7,369 statements
- ▶ 214 type-casts
- ▶ 1.2 type-casts per 100 lines
- ▶ 2.9 type-casts per 100 statements

Type-Casts in JHotDraw

Some figures:

- ▶ 17,807 lines
- ▶ 7,369 statements
- ▶ 214 type-casts
- ▶ 1.2 type-casts per 100 lines
- ▶ 2.9 type-casts per 100 statements

Type-Casts in JHotDraw

Some figures:

- ▶ 17,807 lines
- ▶ 7,369 statements
- ▶ 214 type-casts
- ▶ 1.2 type-casts per 100 lines
- ▶ 2.9 type-casts per 100 statements

Classification for Type-Casts

Category	Count	Percentage
Collections	51	23.83%
Heterogenous Collections	29	13.55 %
Pattern	53	24.77 %
Numbers	25	11.68 %
Instance-Of	15	7.01%
Cloning	9	4.21%
Equals	1	0.47%
Serialization	21	9.81%
Reflection	1	0.47%
Unnecessary	2	0.93%
Unsafe	6	2.80%
Other	1	0.47%

Numbers

```
public class Geom {
    static public Point angleToPoint(Rectangle r, double angle) {
        double si = Math.sin(angle);
        double co = Math.cos(angle);
        double e = 0.0001;
        int x = 0, y = 0;
        if (Math.abs(si) > e) {
            x = (int) ((1.0 + co/Math.abs(si))/2.0 * r.width);
            x = range(0, r.width, x);
        } else if (co >= 0.0)
            x = r.width;
        if (Math.abs(co) > e) {
            y = (int) ((1.0 + si/Math.abs(co))/2.0 * r.height);
            y = range(0, r.height, y);
        } else if (si >= 0.0)
            y = r.height;
        return new Point(r.x + x, r.y + y);
    }
}
```

- ▶ no real type-cast, but conversion

Collections

```
public abstract class CompositeFigure extends /* ... */ {
    protected Vector fFigures;
    public Figure add(Figure figure) {
        if (!fFigures.contains(figure)) {
            fFigures.addElement(figure);
            figure.addToContainer(this);
        }
        return figure;
    }
    public Figure figureAt(int i) {
        return (Figure)fFigures.elementAt(i);
    }
    public Figure findFigure(Rectangle r) {
        FigureEnumeration k = figuresReverse();
        while (k.hasMoreElements()) {
            Figure figure = k.nextFigure();
            Rectangle fr = figure.displayBox();
            if (r.intersects(fr))
                return figure;
        }
        return null;
    }
}
```

Factoring out Type-Casts

```
public final class FigureEnumerator
    implements FigureEnumeration {

    Enumeration fEnumeration;

    public FigureEnumerator(Vector v) {
        fEnumeration = v.elements();
    }

    public boolean hasMoreElements() {
        return fEnumeration.hasMoreElements();
    }

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

With Generics (in Java 1.5)

```
public abstract class CompositeFigure extends /* ... */ {
    protected Vector<Figure> fFigures;
    public Figure add(Figure figure) {
        if (!fFigures.contains(figure)) {
            fFigures.addElement(figure);
            figure.addToContainer(this);
        }
        return figure;
    }
    public Figure figureAt(int i) {
        return fFigures.elementAt(i);
    }
    public Figure findFigure(Rectangle r) {
        for (Figure figure : figuresReverse()) {
            Rectangle fr = figure.displayBox();
            if (r.intersects(fr))
                return figure;
        }
        return null;
    }
}
```

instanceof

```
public class FigureAttributes /* ... */ {
    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 if (v instanceof Integer) {
            dw.writeString(" Int");
            dw.writeInt(((Integer)v).intValue());
        }
        /* ... */
    }
}
```

type-safe instanceof

```
public class FigureAttributes /* ... */ {
    public void write(StorableOutput dw) {
        /* ... */
        if (v instanceof String as s) {
            // s has now static type String
            dw.writeString(" String");
            dw.writeString(s);
        } else if (v instanceof Color as v) {
            // v has now static type Color
            writeColor(dw, " Color", v);
        } else if (v instanceof Boolean as b) {
            // b has now static type Boolean
            dw.writeString(" Boolean");
            if (v.booleanValue())
                dw.writeString("TRUE");
            else
                dw.writeString("FALSE");
        }
        /* ... */
    }
}
```

Patterns

In implementation of *Strategy*-Pattern:

```
public class PolygonFigure extends AttributeFigure {
    public static Locator locator(final int pointIndex) {
        return new AbstractLocator() {
            public Point locate(Figure owner) {
                PolygonFigure plf = (PolygonFigure)owner;
                if (pointIndex < plf.pointCount())
                    return ((PolygonFigure)owner).pointAt(pointIndex);
                return new Point(-1, -1);
            }
        };
    }
}
```

Why is this safe?

- ▶ Locator `l` returned from `PolygonFigure pf`
- ▶ guaranteed that `l.locate` only called with argument `pf`

Results

Category	Count	Percentage
Collections	51	23.83%
Heterogenous Collections	29	13.55 %
Pattern	53	24.77 %
Numbers	25	11.68 %
Instance-Of	15	7.01%
Cloning	9	4.21%
Equals	1	0.47%
Serialization	21	9.81%
Reflection	1	0.47%
Unnecessary	2	0.93%
Unsafe	6	2.80%
Other	1	0.47%

Conclusions

- ▶ a lot of type-casts can be eliminated in Java 1.5
- ▶ further extensions required
- ▶ careful analysis to balance tradeoffs is needed
- ▶ test classification on other projects

Thank you for your attention!

Appendix

More Code examples

Clone

► clients using `Object.clone()`

```
public class CreationTool extends AbstractTool {
    private Figure fPrototype;
    protected Figure createFigure() {
        if (fPrototype == null)
            throw new HJDError("No_prototype_defined");
        return (Figure) fPrototype.clone();
    }
}
```

► implementing `clone()` method

```
public class FigureAttributes /* ... */ {
    public Object clone() {
        try {
            FigureAttributes a = (FigureAttributes) super.clone();
            a.fMap = (Hashtable) fMap.clone();
            return a;
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
}
```

Solutions

- ▶ clients using `clone`: covariant return-types
- ▶ `clone` implementation: `ThisType`