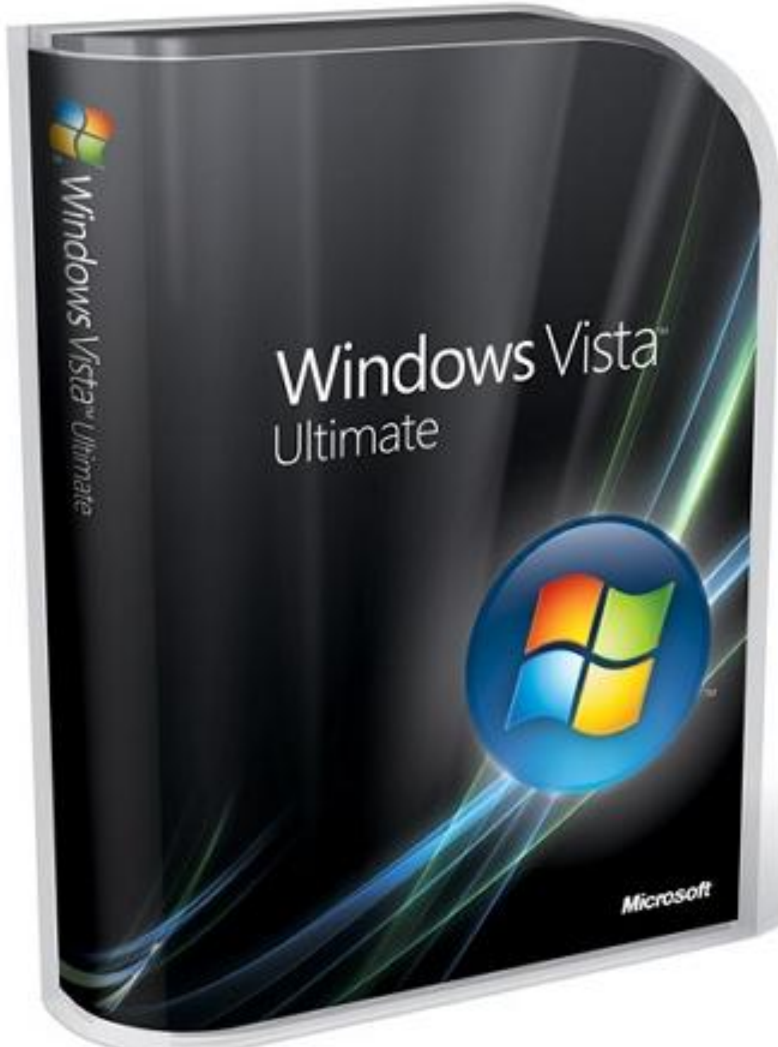
UpgradeJ: Incremental typechecking for class upgrades

Gavin Bierman

Matthew Parkinson

James Noble

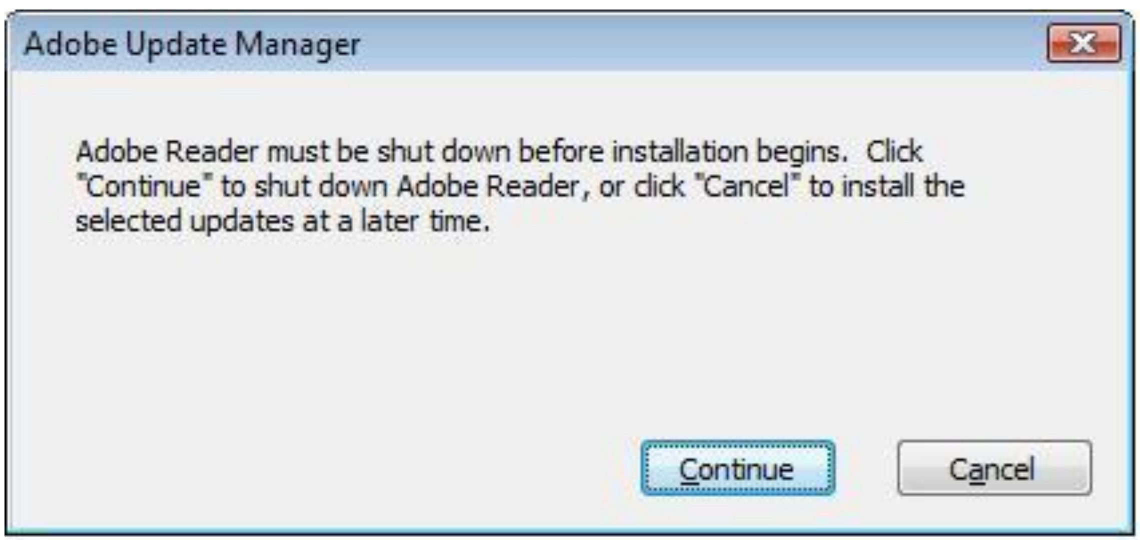
Software always evolves



Why DSU/hotswapping?

Most software needs to be fixed (debug or add functionality), *but …*

Not all software can do this:



Dynamic Software Updating (DSU)

Provide support for applications
to evolve **during execution**

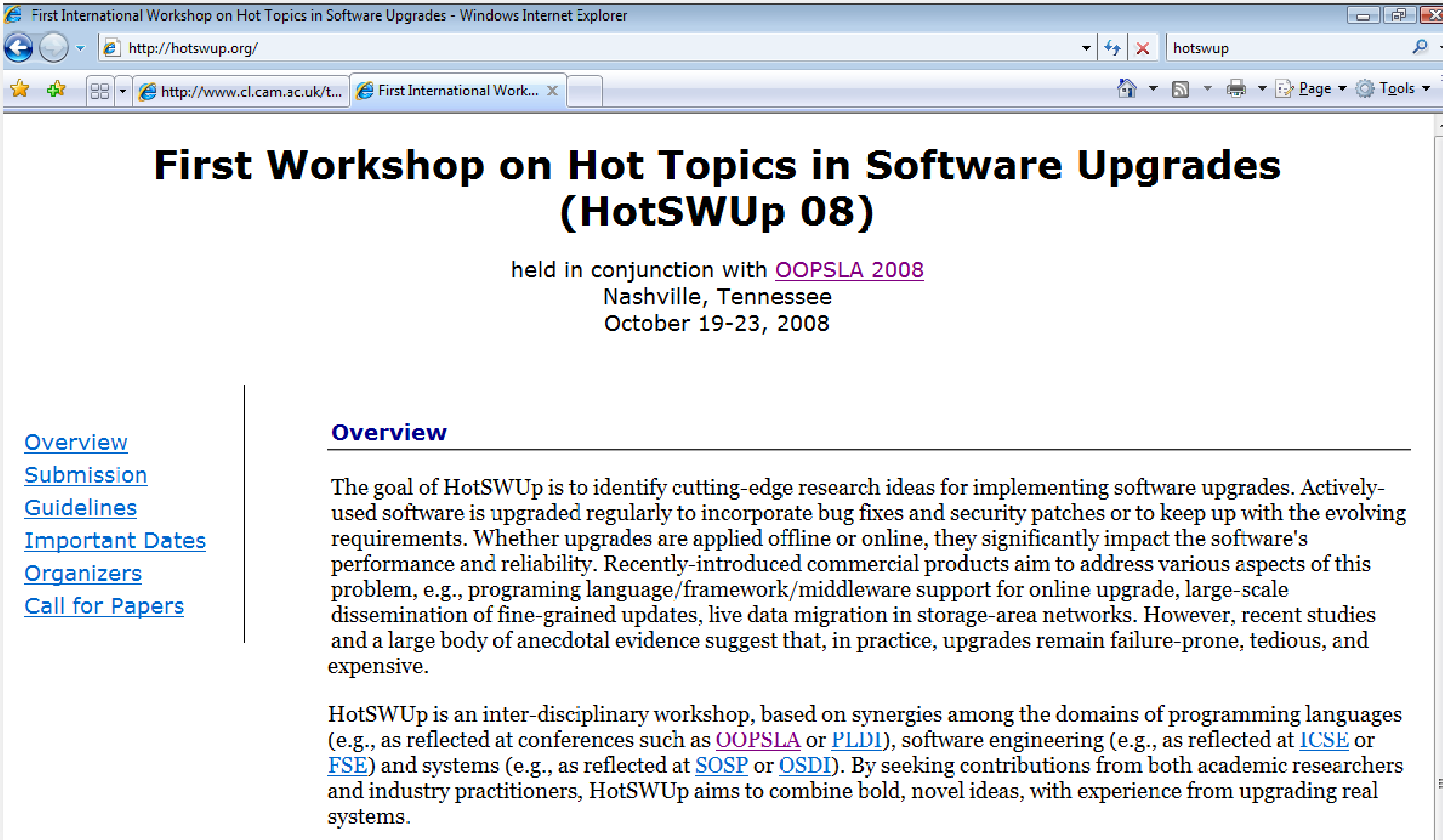
Hardcore DSU techniques

- Hardware redundancy
  - E.g. Visa uses 21 mainframes
  - E.g. ACARS digital messaging system used by airlines has *primary* and *standby* machines
- State transfer
- (Even worse): horrid binary patching

More familiar DSU techniques

- Dynamic linking
  - Yielding plug-in extensibility; but often leads to
    - DLL hell, or
    - Config-file hell
- Class loading hacking

Academic work



Academic work

- Michael Hicks (& co-authors)
  - PhD [2001] for imperative programs
  - ICFP [2003] for functional programs
  - POPL [2005] for C-like programs (theory)
  - PLDI [2006] for C-like programs (practice)
- Erlang [Armstrong et al. 96]
- Dynamic ML [Gilmore et al. 97]
- *And lots more...*

Our aim

- A **language-level** approach to the following question:

How could we write type-safe DSU applications in a **Java-like** language?

cf. “Runtime support for type-safe dynamic Java classes”
Malabarba, Pandey, Gragg, Barr and Barnes
ECOOP 2000

Warning

- DSU-programming is generally left to wizards!
- **Not** a proposal for core-Java
  - Although we have tried to be as lightweight as possible

UPGRADEJ

Design decision #1: Granularity of upgrades

- Classes *or* Packages *or* Modules *or* Concord-style Packages *or* ...

Design decision #1:

Class upgrading

Design decision #2: Versioning

- Implicit vs explicit evolution
  - i.e. Does `widgit` evolve to `widgit`, or does `widgit[n]` evolve to `widgit[n+1]`?

> **Design decision #2:**
>
> **Explicit** versions; thus support **multiple** co-existing versions

- Most DSU work takes the implicit route
  - Including Malabarba et al.

Explicit versions

No versioning of `Object`

```
class Button[1] extends Object {

    Object press() { ... }

}
class AnimatedButton[1] extends Button[1] {

     Object fancyPress() { ... this.press(); ... }

}
```

For simplicity, assume **all** class names are versioned

Design decision #2b: Versioning

- What is a version-number?
  - `a.b` *or* `a.b.c.` *or* `a.b.c.d` *or* floats *or* ...

Design decision/simplifying assumption #2b:

A version is an **integer**

Design decision #3: Control timing of upgrade

- When do upgrades happen?
  - Dynamic ML – at GC sweep-time
  - Erlang – check at every explicit function call
  - Malabarba et al. – by explicit call to special class loader

Design decision #3:

Add a new `upgrade;` statement whose effect is to **install an upgrade**

- Lots of evidence that its better for applications to be explicit about timing of upgrades

Design decision #4: Upgrades

Design decision #4:

UpgradeJ supports three forms of upgrades:

1. **New class** upgrades
2. **Revision** upgrades
3. **Evolution** upgrades

New class upgrades

- Enables the class table to be extended

```
new class AnimatedButton[1] extends Button[1] {

    Object fancyPress() {
        ... this.press(); ...
    }

}
```

Gets the inherited method from `Button[1]` as normal

Design decision #6: Incremental typechecking

- How will we check upgrades?

Design decision #5:

UpgradeJ supports **incremental typechecking**, i.e.

- No definition is **re-type checked**
- An upgrade is checked **only** when it arrives

Design decision #7: Static checking

- How will we check upgrades?

Design decision #7:

UpgradeJ does **not** inspect run-time values when checking an upgrade.

Makes upgrade validity **easier** to predict.

c.f. Dynamic ML & Hicks et al. USE [2003] calculus

Design decision #8: No time travel

Design decision #8:

UpgradeJ does **not** allow future classes to be referenced in current code.

```
...
upgrade;
b = new Button[2]();
```

```
new class Button[2]{ ... }
```

Revision upgrades

```
class Button[1] extends Object {
    Object press(){
    ...
    }
    Colour bgColour() {
        return new BeigeColour[1]();
    }
}
```

```
new class Button[2] extends Object revises Button[1]{
    Object press(){
    ...
    }
    Colour bgColour() {
        return new GreyColour[1]();
    }
}
```

Design decision #9: upgradeable instances

Design decision #9:

UpgradeJ supports instance creation to be annotated

```
Button[1] x = new Button[1=]();
Button[1] u = new Button[1+]();

x.bgColour();
u.bgColour();

upgrade;

x.bgColour();
u.bgColour();
```

Fixed at version 1

Start at version 1, and **allow revision**

`Button[1]`

`Button[1]`

`Button[1]`

`!!Button[2]!!`

```
new class Button[2]
  revises Button[1] { ... }
```

Revision upgrades

- A revision upgrade (R *revises* C) is valid if
  - R's fields are **the same** as C's
  - The "method signatures" of R must be **the same** as the method signatures of C
    - i.e. all the methods and their types that are understood by the class including inherited methods
  - There isn't already a revision of that class (single revision restriction)

Revision upgrades are like bug-fix upgrades

Design decision #10: upgradeable instances

Design decision #10:

UpgradeJ does **not** upgrade any run-time objects.

The upgrade is to the **class table**.

Upgrades thus effect which method body may be executed.

c.f. Malabarba et al. where all current objects of old version are upgraded

Multiple revisions

```
Button[1] x = new Button[1=]();
Button[1] u = new Button[1+]();

x.bgColour();     Button[1]
u.bgColour();     Button[1]

upgrade;   <--   new class Button[2]
                   revises Button[1] { ... }

x.bgColour();     Button[1]
u.bgColour();     Button[2]

upgrade;   <--   new class Button[3]
                   revises Button[2] { ... }

x.bgColour();     Button[1]
u.bgColour();     Button[3]
```

Evolution upgrades

```
new class Button[3] extends Object revises Button[2]{
    Object press(){
    ...
    }
    Colour bgColour() {
        return new TransparentAquaColour[1]();
 }
}
```

```
new class Button[4] extends Object evolves Button[3]{
    Integer animationRate;
    void tick() {this.redraw(); }
    Colour bgColour() {
        return new VistaBlackColour[1]();
    }
   ...
}
```

New!

New!

Evolution upgrades

- An evolution upgrade (E *evolves* C) is valid if
  - C's fields are **contained in** E's
  - The "method signatures" of C must be **contained in** the method signatures of E
  - There isn't already an evolution of that class (single evolution restriction)

Evolution upgrades allow
breaking changes

Another annotation

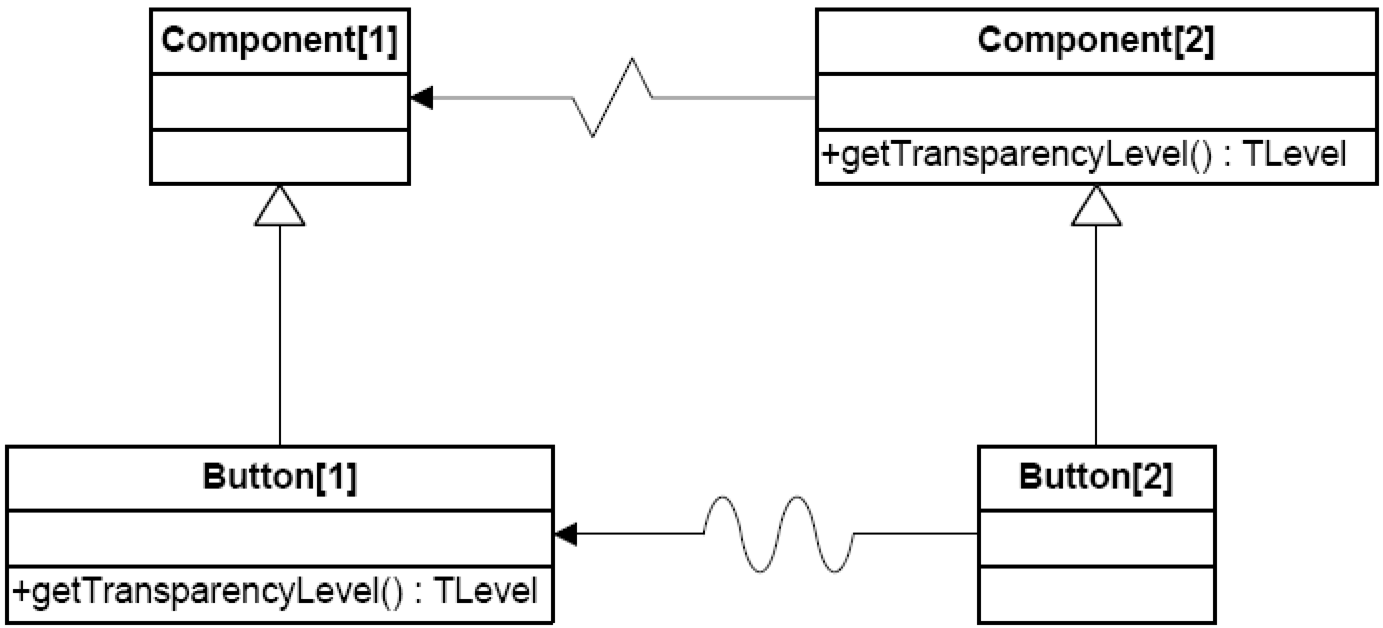
```
…
Button[1] e = new Button[1++]();
Button[1] f = new Button[1+]();

e.bgColour();          Button[3]

upgrade;

e.bgColour();          Button[3]
f.bgColour();          Button[3]

e = new Button[1++]();

e.bgColour();          Button[4]
```

```
new class Button[3]
  revises Button[2] { ... }
```

```
new class Button[4]
  evolves Button[3] { ... }
```

`new C[n++]()` gets latest **revision** of latest **evolution**

Refactoring via upgrades



Expressivity

- In the paper we show how to upgrade a server **whilst running** to:
  1. Handle new event types; and
  2. Log events being handled

- Still assessing the expressivity of UpgradeJ

FUJ

- A core fragment of UpgradeJ
- Similar to Featherweight Java:
  - Small, amenable to proof
- Different to Featherwight Java:
  - Statement-based, rather than expression-based
  - Thus we have a heap!!
  - Hence we're more confident of our results applying to UpgradeJ☺

Object creation: 3 annotations

$$(CT, S, H, \texttt{x = new C[v=]();}\overline{\texttt{s}}) \longrightarrow (CT, S[\texttt{x} \mapsto o], H', \overline{\texttt{s}})$$

where $\mathit{fields}(CT, \texttt{C[v=]}) = \overline{T}\ \overline{\texttt{f}}$, $o \notin \mathit{dom}(H)$, and $H' = H[o \mapsto \langle \texttt{C[v=]}, \{\overline{\texttt{f}} \mapsto \texttt{null}\}\rangle]$

$$(CT, S, H, \texttt{x = new C[v+]();}\overline{\texttt{s}}) \longrightarrow (CT, S[\texttt{x} \mapsto o], H', \overline{\texttt{s}})$$

where $\mathit{fields}(CT, \texttt{C[v=]}) = \overline{T}\ \overline{\texttt{f}}$, $o \notin \mathit{dom}(H)$, and $H' = H[o \mapsto \langle \texttt{C[v+]}, \{\overline{\texttt{f}} \mapsto \texttt{null}\}\rangle]$

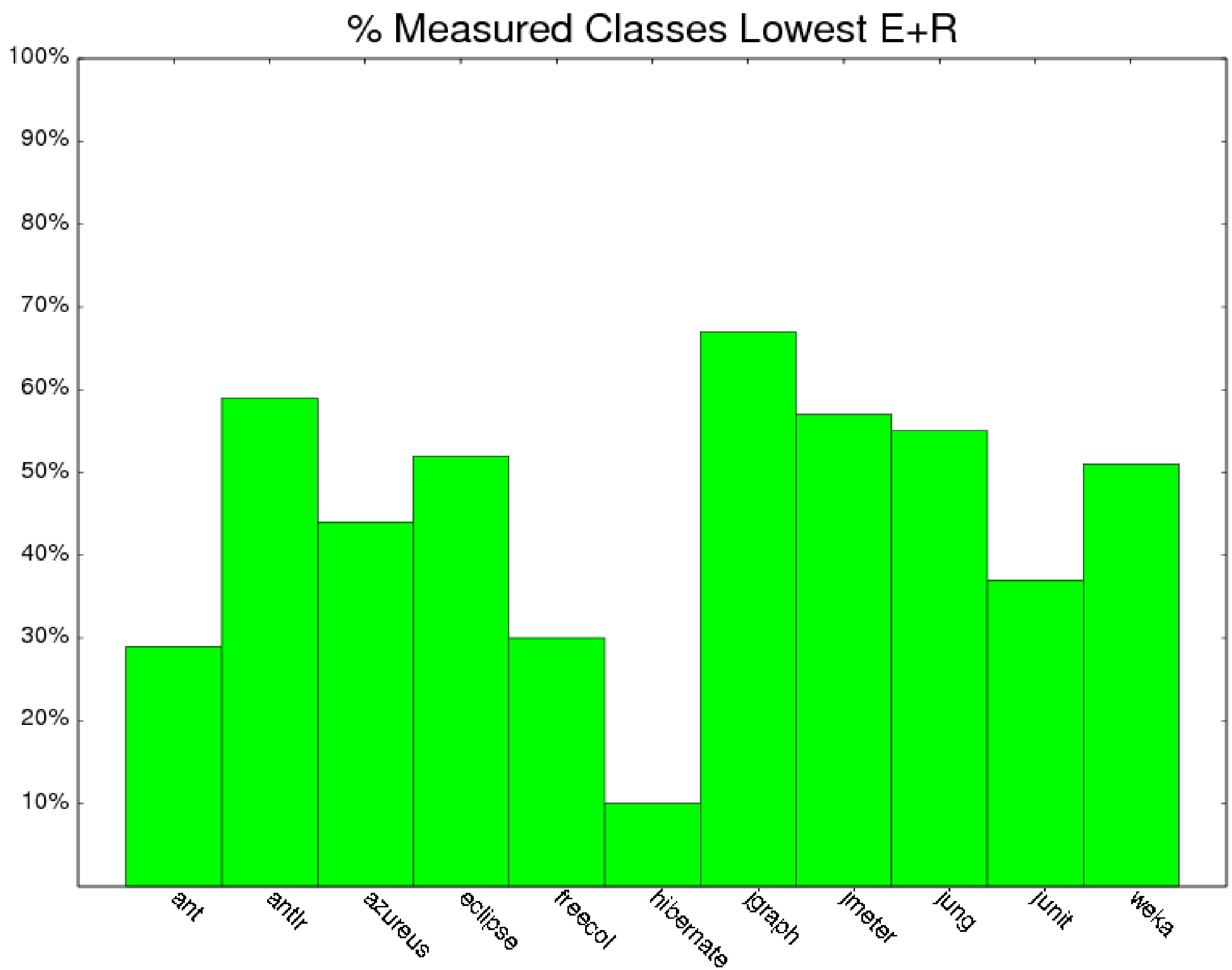
$$(CT, S, H, \texttt{x = new C[v++]();}\overline{\texttt{s}}) \longrightarrow (CT, S[\texttt{x} \mapsto o], H', \overline{\texttt{s}})$$

where $\mathit{latest}(CT, \texttt{C[v=]}) = \texttt{I}$, $\mathit{fields}(CT, \texttt{I}) = \overline{T}\ \overline{\texttt{f}}$, $o \notin \mathit{dom}(H)$, and $H' \stackrel{\text{def}}{=} H[o \mapsto \langle \texttt{I+}, \{\overline{\texttt{f}} \mapsto \texttt{null}\}\rangle]$

[On-going joint work with Ewan Tempero]

EMPIRICAL DATA

Some very preliminary data



Other people's evidence

- Tomcat 5.5.0-5.5.26: **98.6%** of method upgrades are revision upgrades
  - [From Monday's workshop☺]
    A Case Study for Aspect Based Updating.
    Susanne Cech Previtali and Thomas R. Gross

Conclusions

- UpgradeJ is an extension of Java that provides **lightweight** DSU via:
    - Multiple co-existing versions of classes
    - Incremental typechecking
  - But:
    - No checks of heap when upgrading EVER
    - No re-typechecking EVER
    - No object munging EVER


What's our big ticket item? Upgrades, people, upgrades. That's how we make the dough.
Why be you, when you can be new?