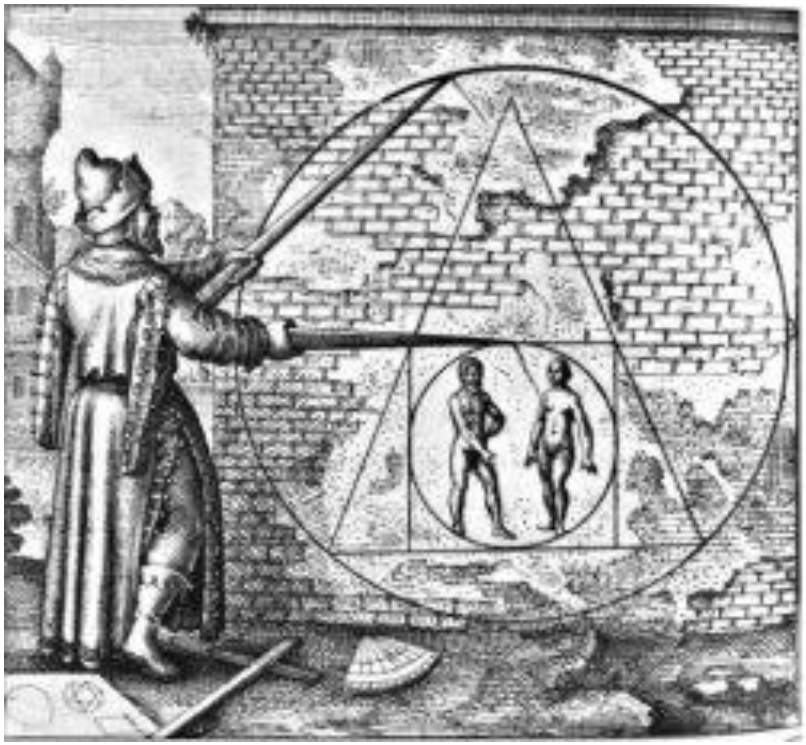
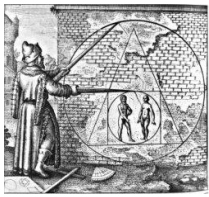
Optimising large dynamic code bases.



imagination at work

Who am I?

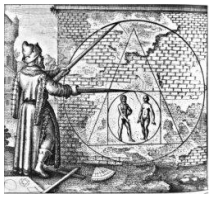


Duncan MacGregor

Lead Software Engineer on the Magik on Java project at General Electric in Cambridge

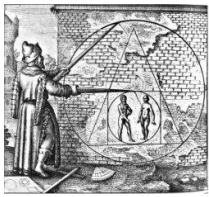
Aardvark179 on twitter

What is Magik?



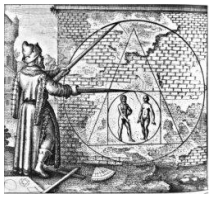
- Dynamic, weakly typed message passing based language
- Does everything "wrong" i.e. numerical type promotion, closures over mutable values, etc.
- Not that special in terms of the language.

So what does make it special?



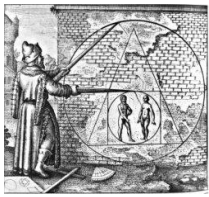
- It s tightly integrated with a long transaction version managed data store.
- There are large applications built using it.
- No, larger than that, I mean really large

So what is large?



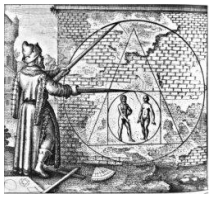
- A typical application with an open database has 200,000 methods defined
- About 10% of those are record field accessors and other code generated by the ORM

It gets worse



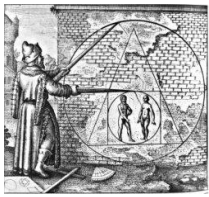
- There are bigger data models out there, some with over 4000 tables
- Many customers will use third party add ons and their own customisations which may be almost as large as the base apps

What are our main concerns?



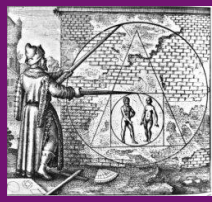
- Performance must be good enough
- Need to startup and open the database as fast as possible
- Must not use ridiculous amounts of memory as the application is used
- Warmup time needs to be as short as possible

What I'm going to talk about today



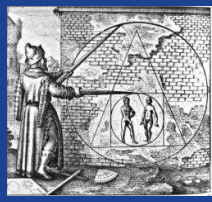
- Memory usage
- Startup time
- Smoothing the path of moving from an old VM to a new one

Memory usage



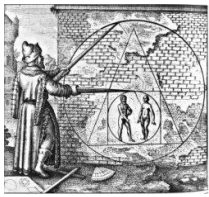
imagination at work

Where we started on Java 8



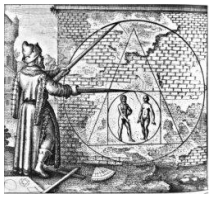
imagination at work

First steps running on Java 8



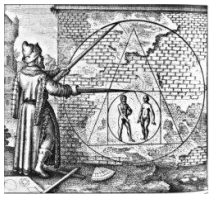
- Measure startup speed relative to Java 7 ☹
- Examine memory relative to Java 7 ☹
- Take heap dumps as the application is used
- Project memory usage of fully exercised application ☹

What's using the memory?



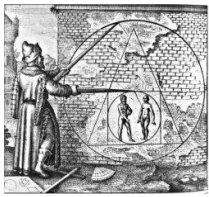
1. LambdaForms
2. LambdaForms
3. LambdaForms
4. Concurrent Collections (!)
5. Atomics
6. Everything else

LambdaForms!



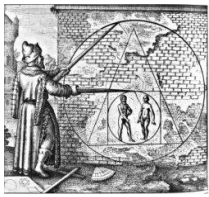
- Binding followed by adapting generated many many LFs
- Likewise drop arguments
- These combined to make SwitchPoint GWT surprising inefficient

Concurrent collections?



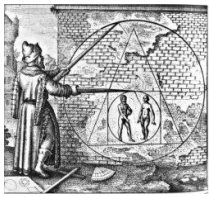
- We want a run time as lock free as possible—a bad application shouldn't lock up core language infrastructure
- Concurrent collections are great in small numbers
- If every method has a little one to hold type adapted versions then the memory adds up

Atomics



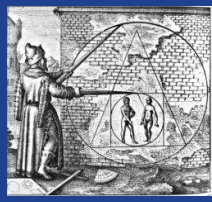
- Again, we want things lock free if possible
- Methods have
  - SwitchPoints to handle invalidation
  - Flags
  - Other meta-data
- All held in atomics to support update by multiple threads

What is everything else?



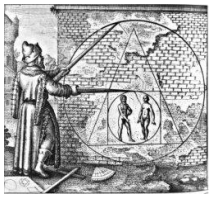
- Antlr's lexer generation produces large short arrays
- Our globals are stored in a prefix tree, we need to move them to a more compact structure
- Reflection caches can be huge
- Many other small things, but they add up

Reducing memory usage



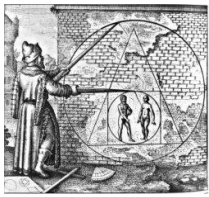
imagination at work

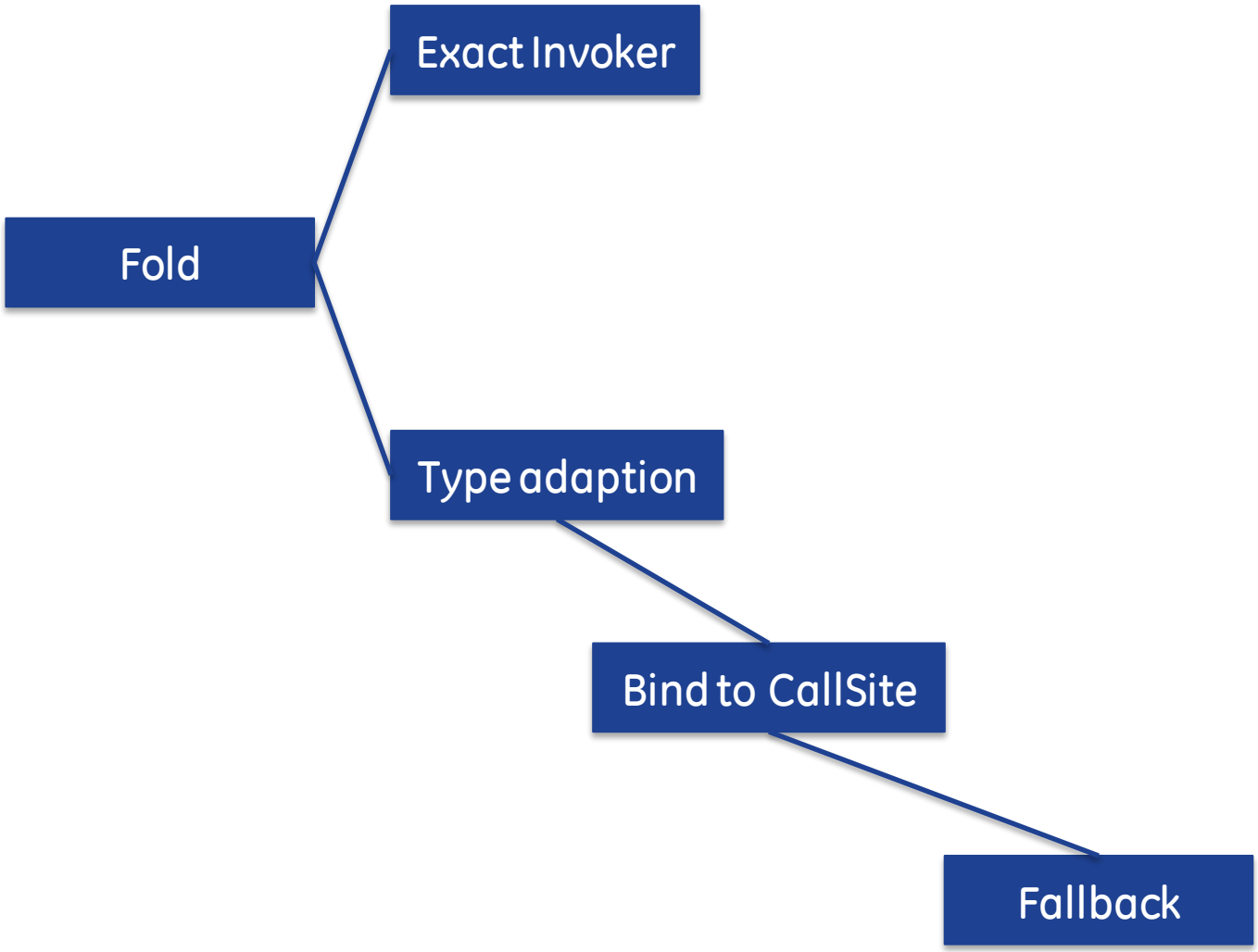
Prototyping / proof of concept



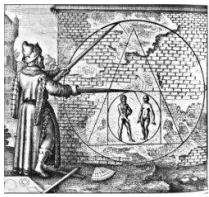
- Change MH adaption order
- Cache unbound adapters for CallSites
- Reduce the PIC of each site to 1
- Write our own SwitchPoint class

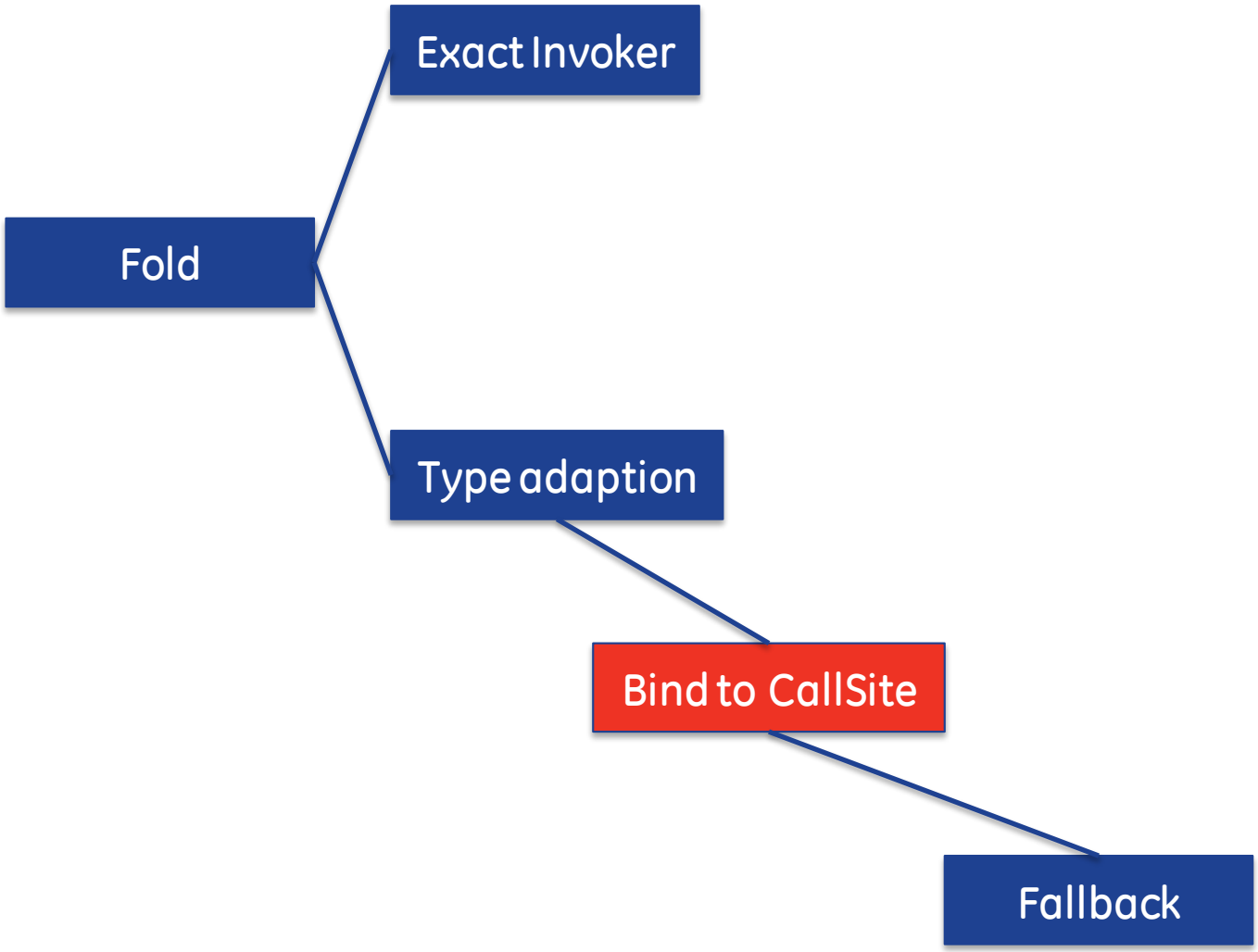
Old CallSite fallback adaption



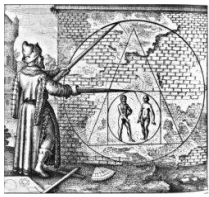


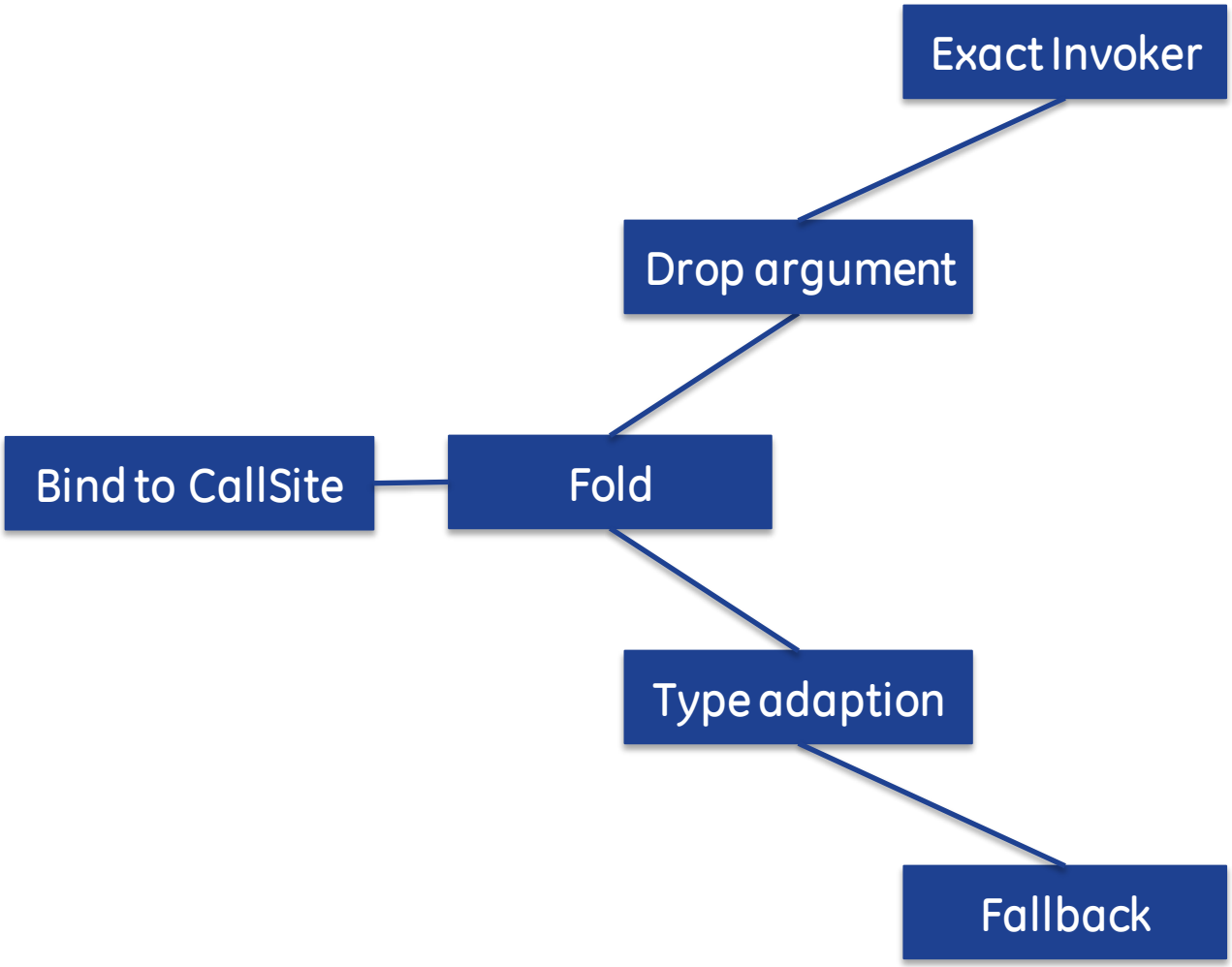
Old CallSite fallback adaption



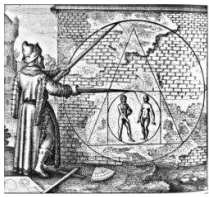


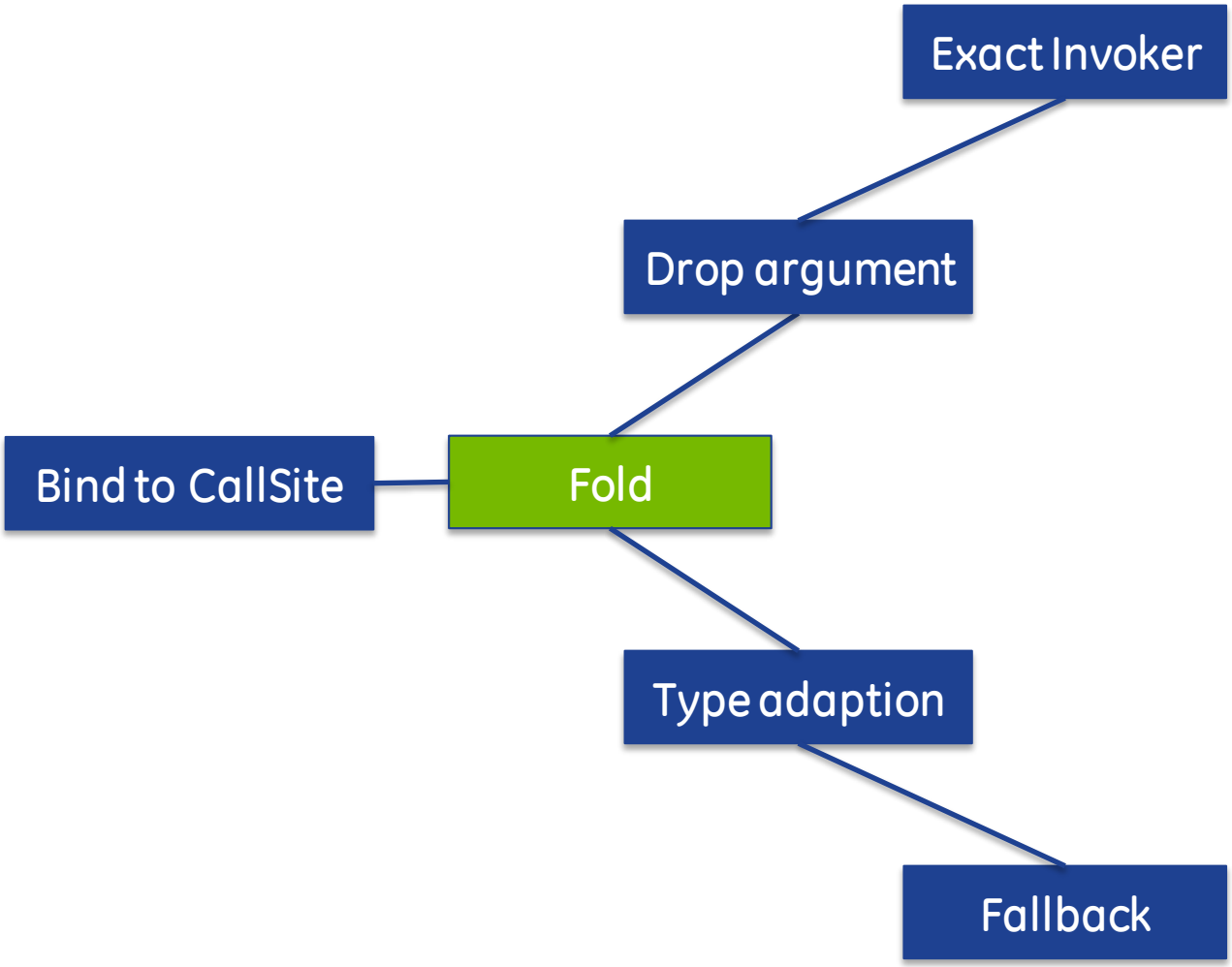
New CallSite fallback adaption



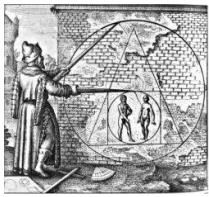


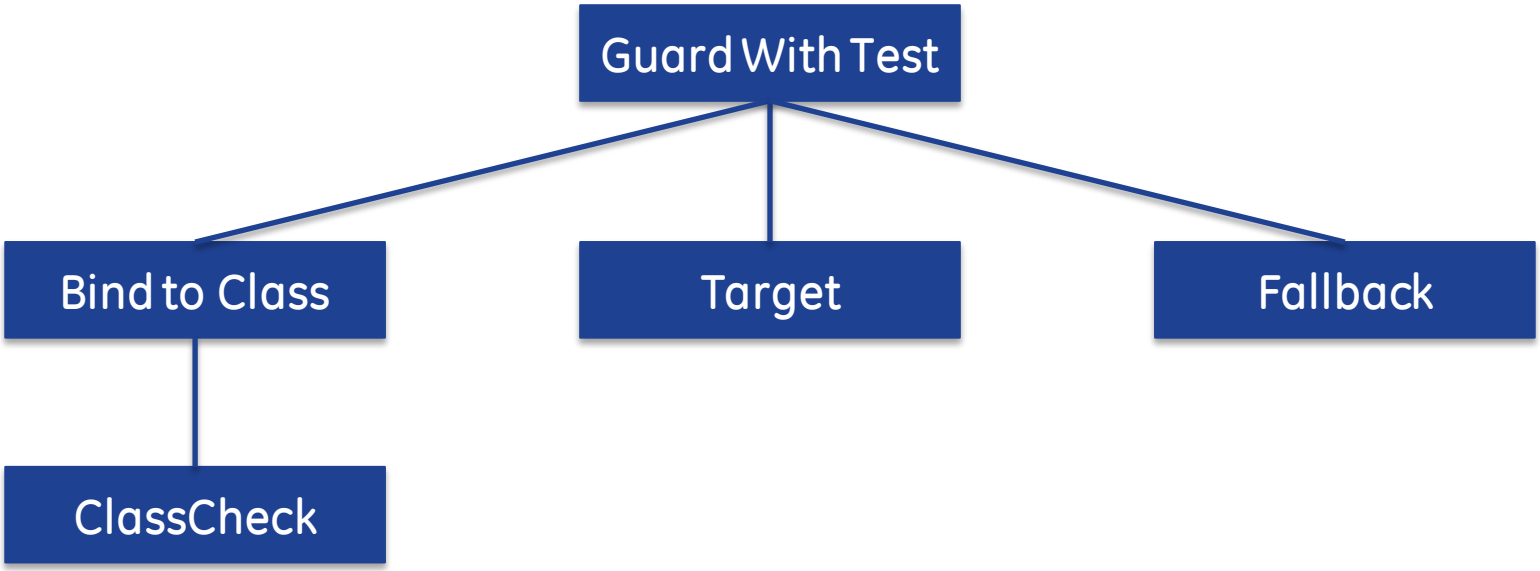
New CallSite fallback adaption



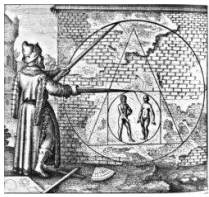


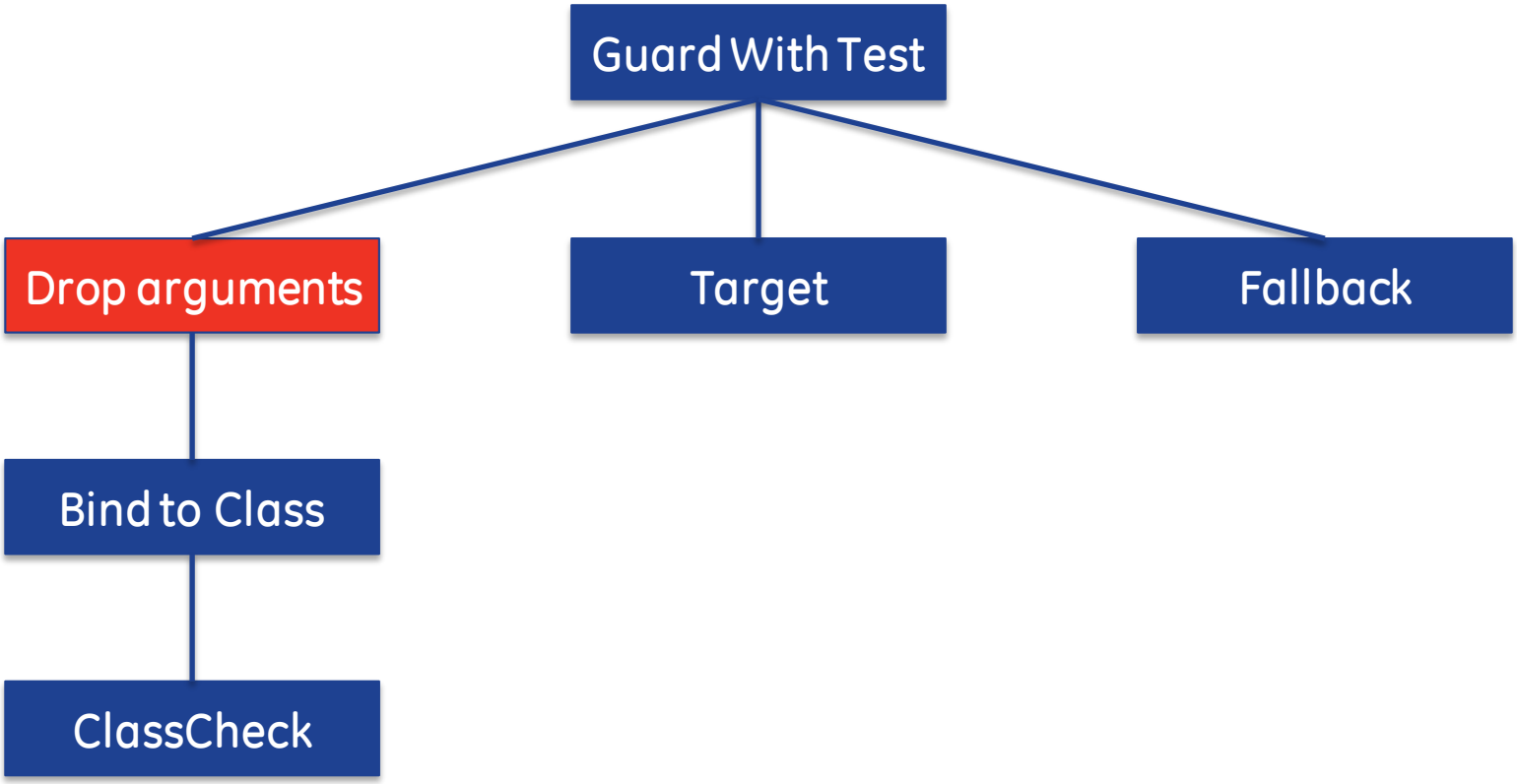
GuardWithTest issue



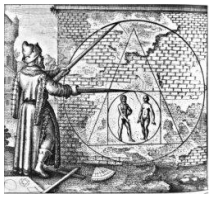


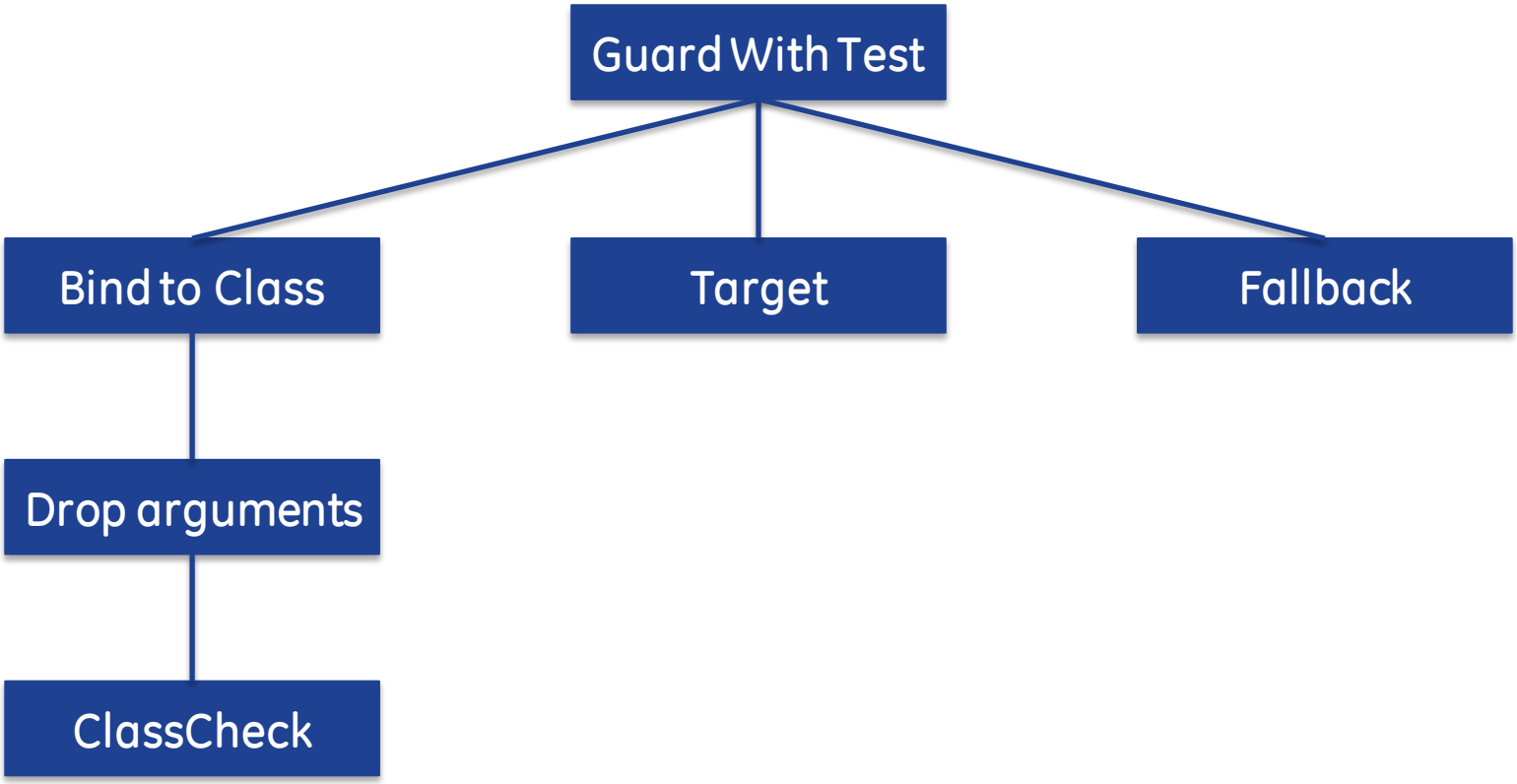
GuardWithTest issue



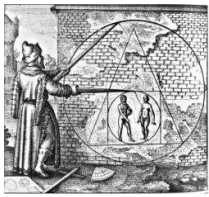


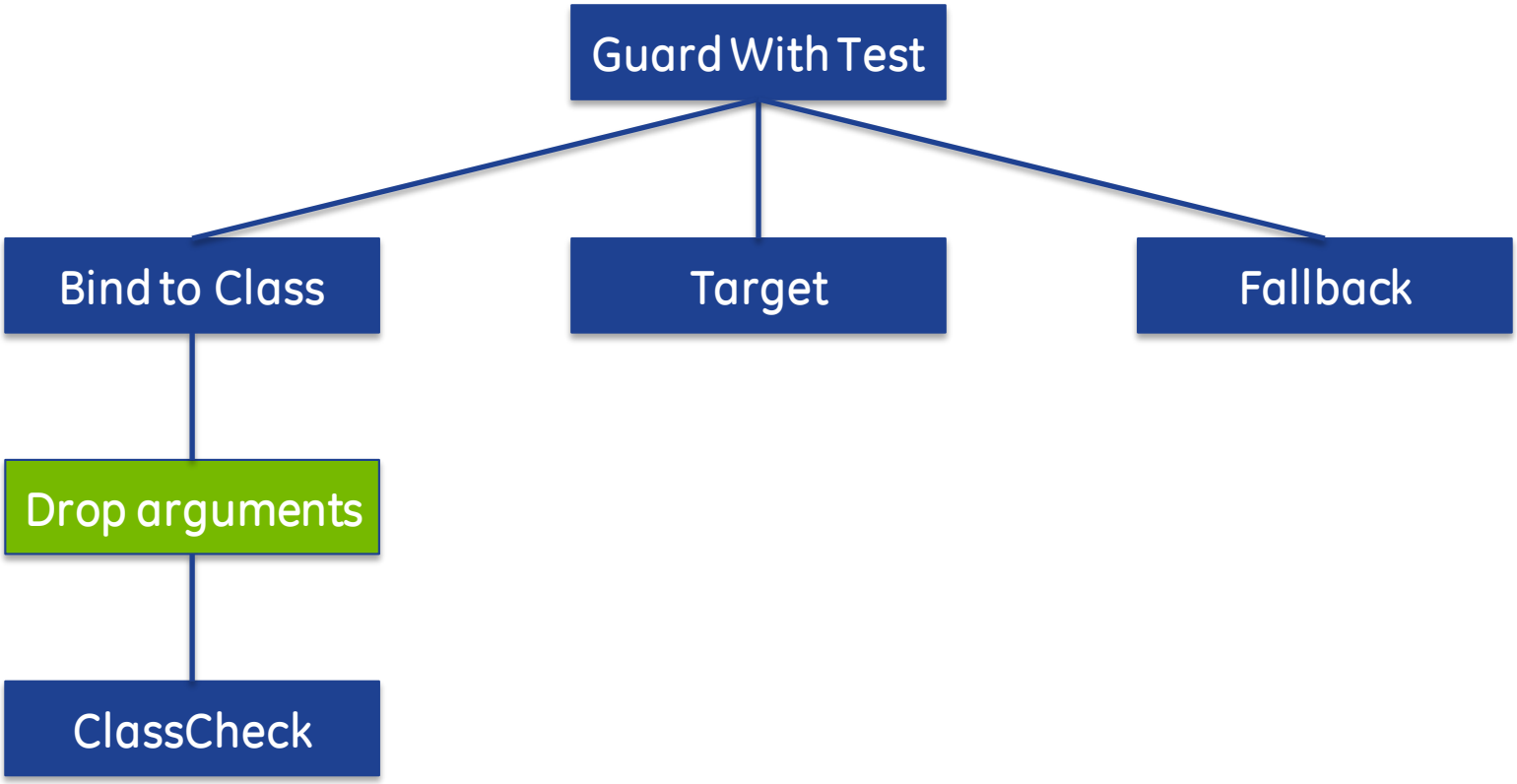
GuardWithTest solution



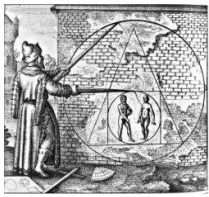


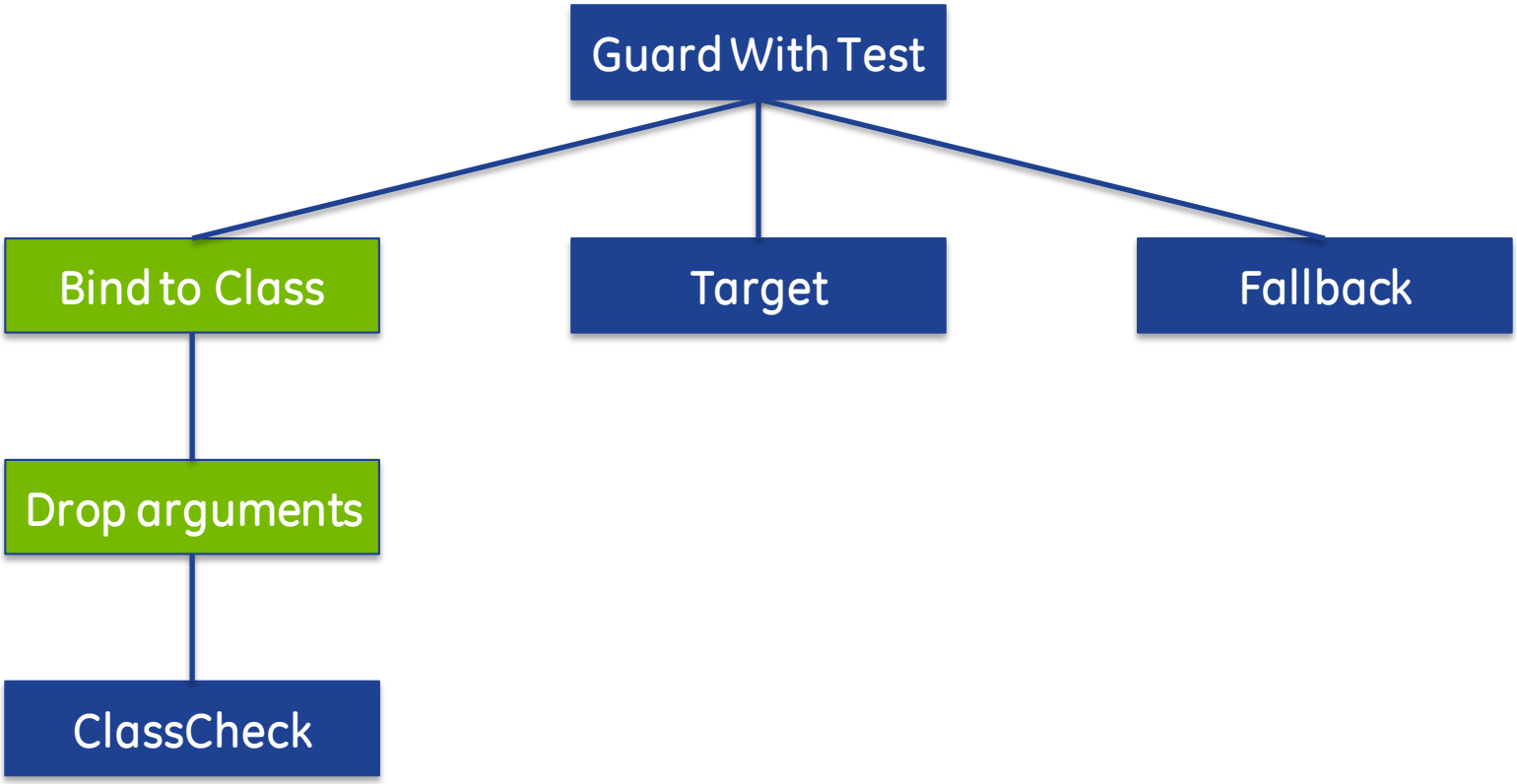
GuardWithTest solution



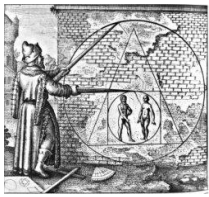


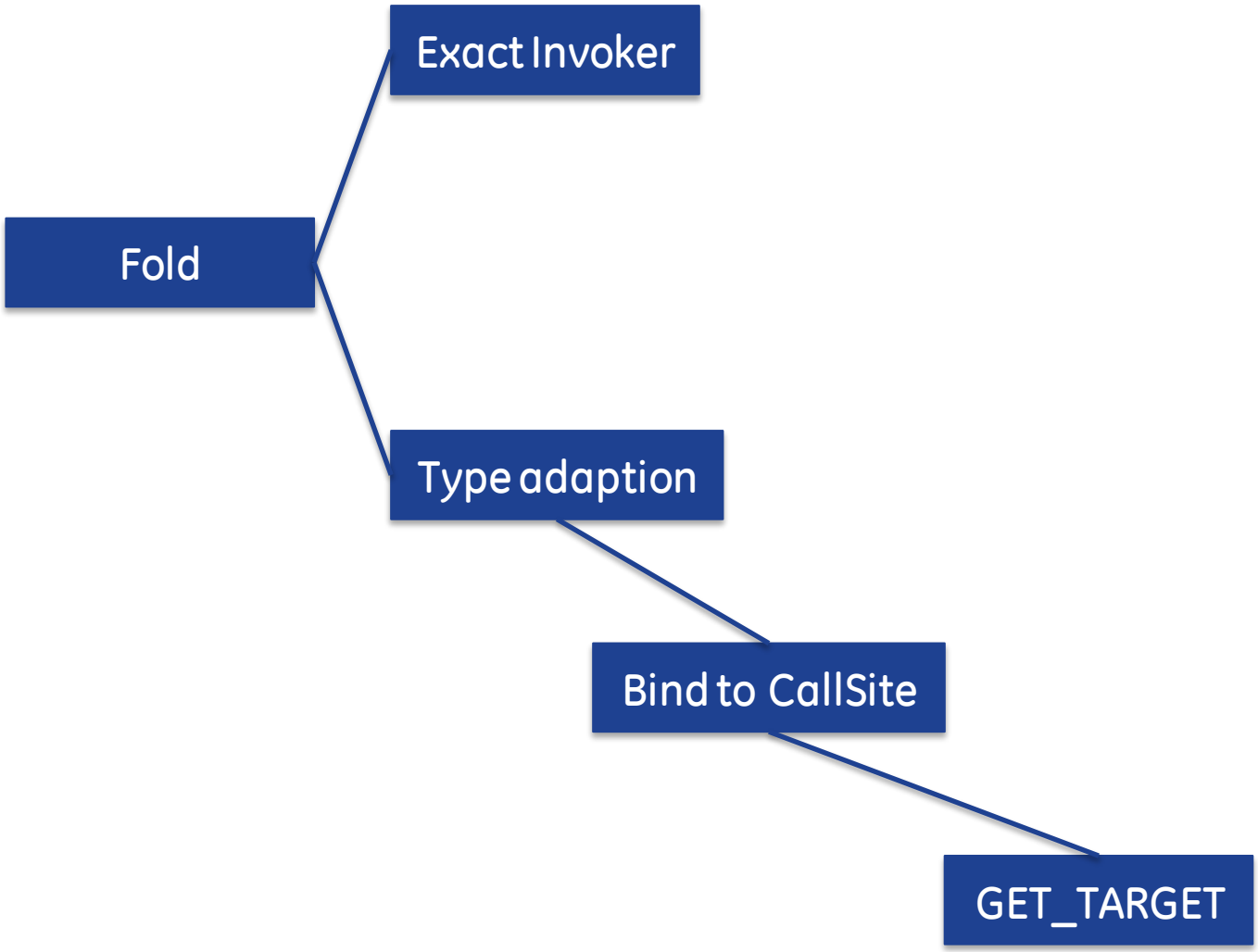
GuardWithTest solution



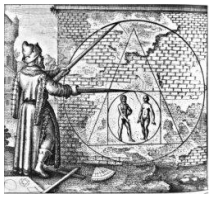


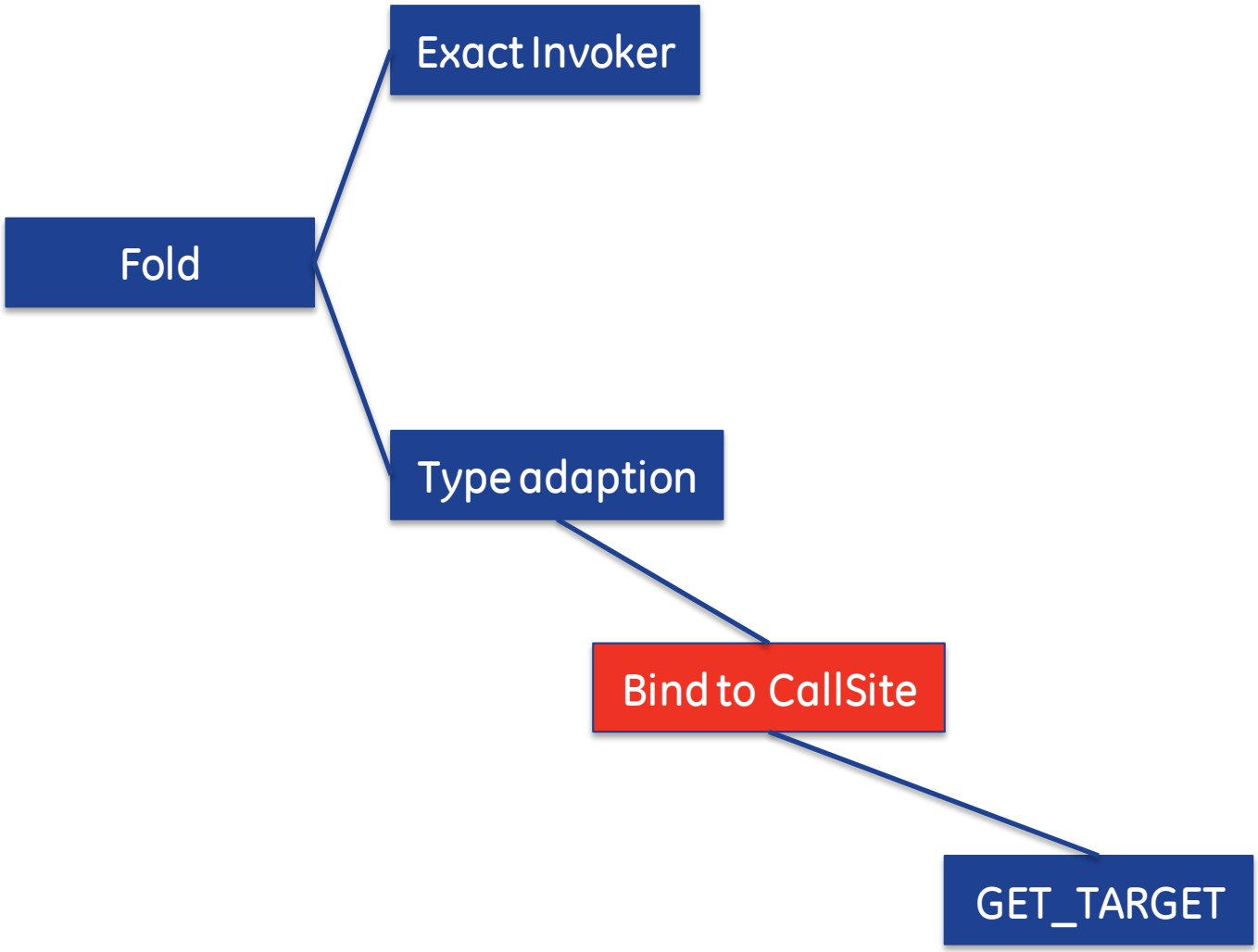
SwitchPoint dynamicInvoker



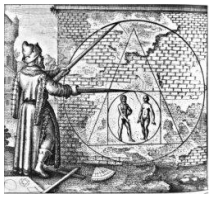


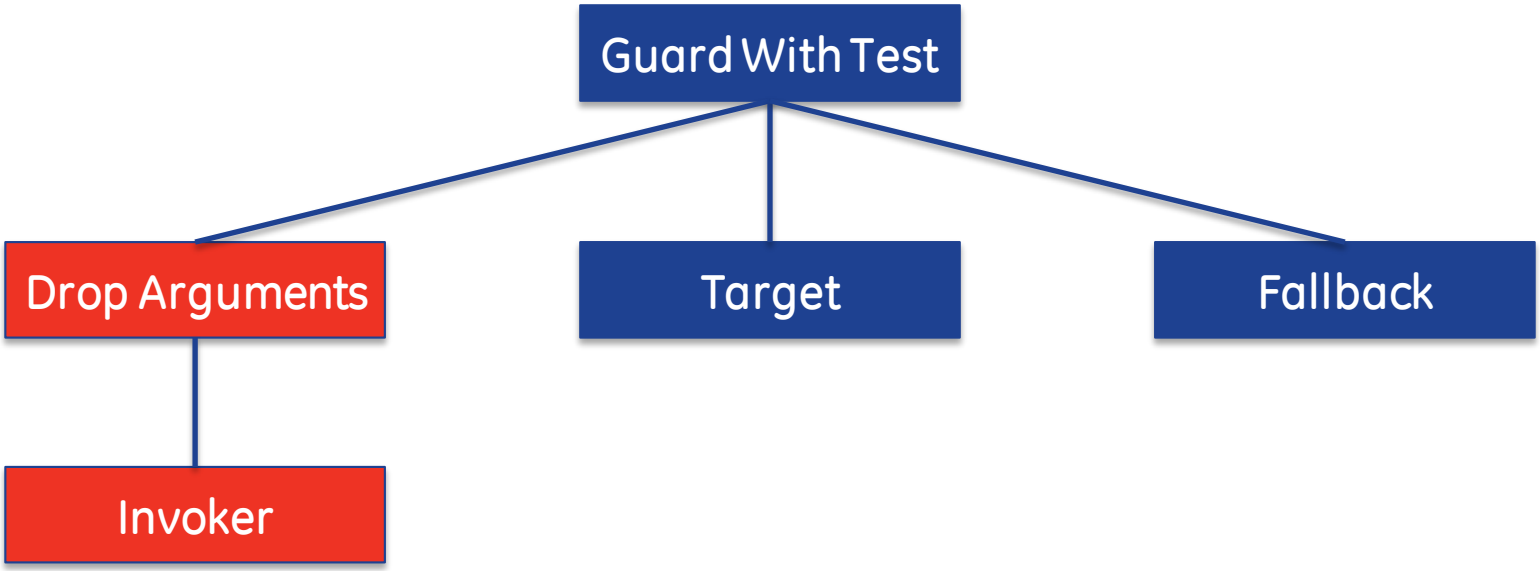
SwitchPoint dynamicInvoker



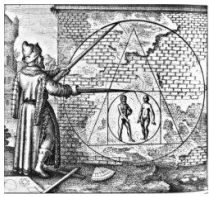


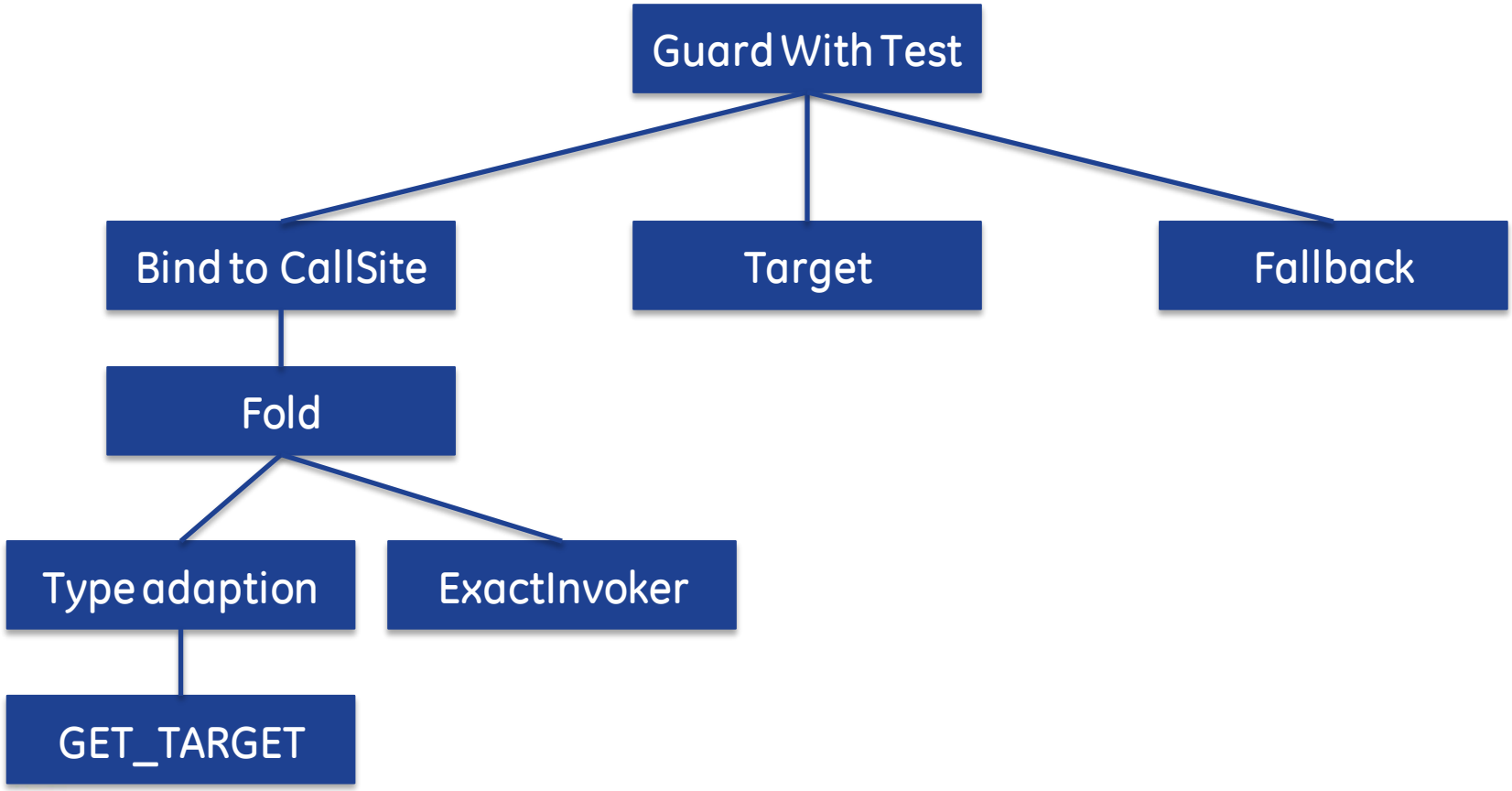
SwitchPoint guardWithTest



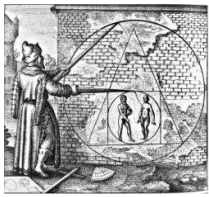


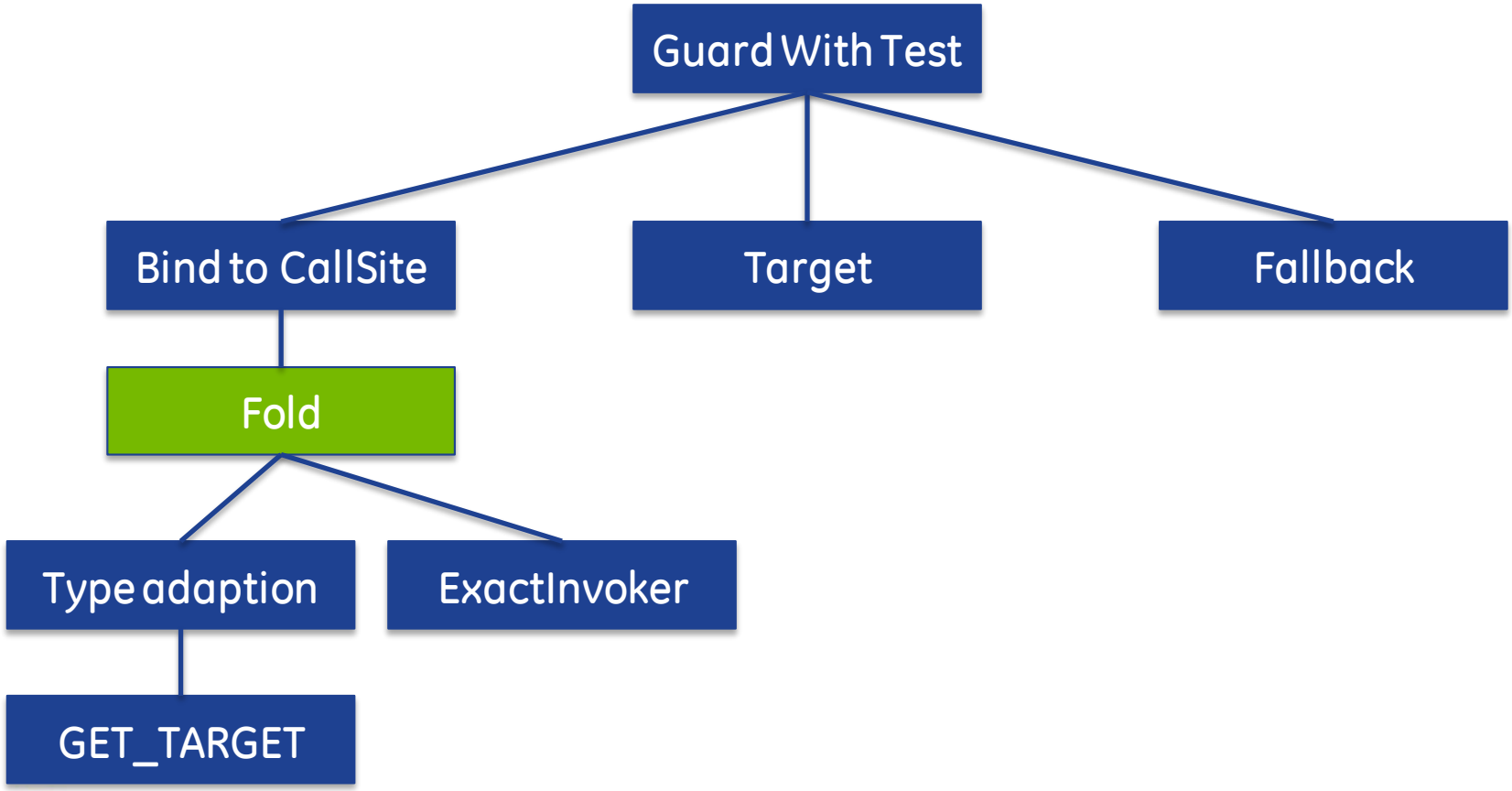
SwitchPoint guardWithTest



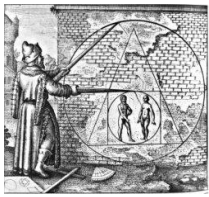


SwitchPoint guardWithTest



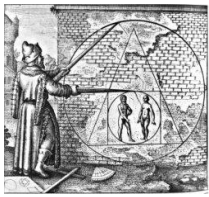


Did it help?



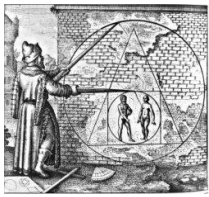
- Yes! More than halved the LFs
- Applications feel faster when first started
- Memory usage significantly reduced

Putting it into production



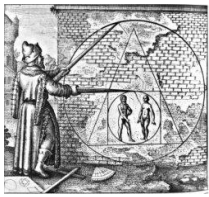
- Started by tackling easy parts. Concurrent collections, Atomics etc.
- Then refactor call sites
- This work is still on going

CallSites: optimising for the common cases



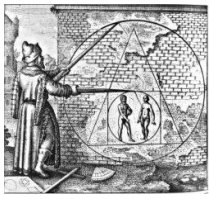
- Method calls are complex
  - Normal, super, self…
  - How many results?
- All sites supported instrumentation
- Added complexity to each instance

Refactor into many classes



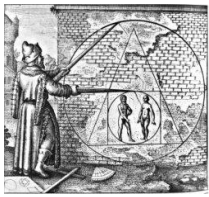
	Normal Call	Call To Self	Super Call
Single Result	Single	PrivateSingle	SuperPrivateSingle
Tuple Result	Tuple	PrivateTuple	SuperPrivateTuple
Unknown Number of Results		Private	SuperPrivate

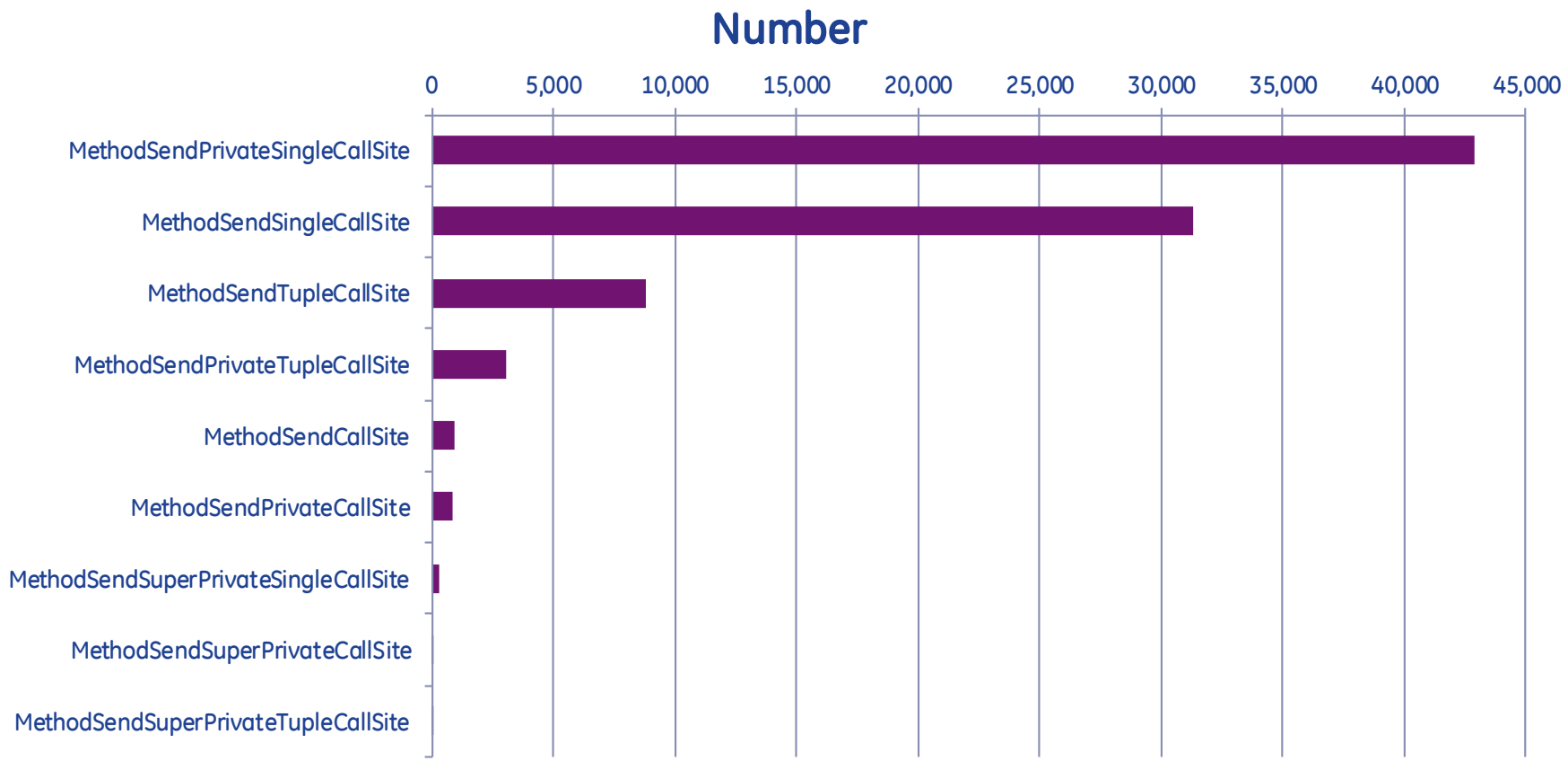
Move towards functional composition



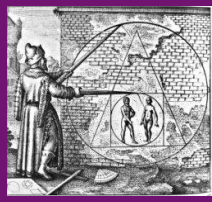
- Hard to build a good class hierarchy without repetition
- Some functionality based on choices at runtime (instrumentation etc.)
- Allows megamorphic costs to only be paid by the sites that need it

Unexpected advantages



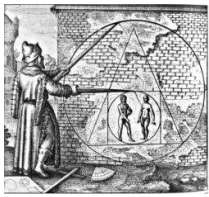


Start up and database open



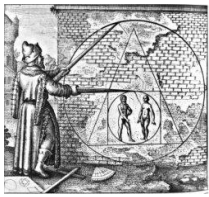
imagination at work

Bootstrapping the system



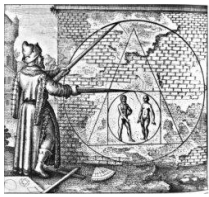
- Bootstrapping the system involves loading a lot of code that is only run once
- Loading a class looks up MethodHandle constants repeatedly
- Loading a module causes resources to be scanned on disk

Improving this



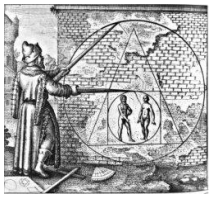
- Improving Java's MH constant caching improves our start up time by over 10%
- Emitting fewer larger classes may also help
- Restructure our resource system to stop using the file system
- We may look at more radical solutions depending on where the time is spent

Opening the database



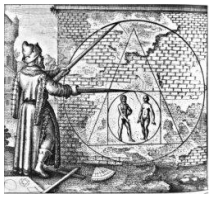
- Lots of meta-programming
  - Subclasses for record types
  - Field access methods
  - Join navigation
  - …
- Old VM allowed images to be saved after all this had been initialised

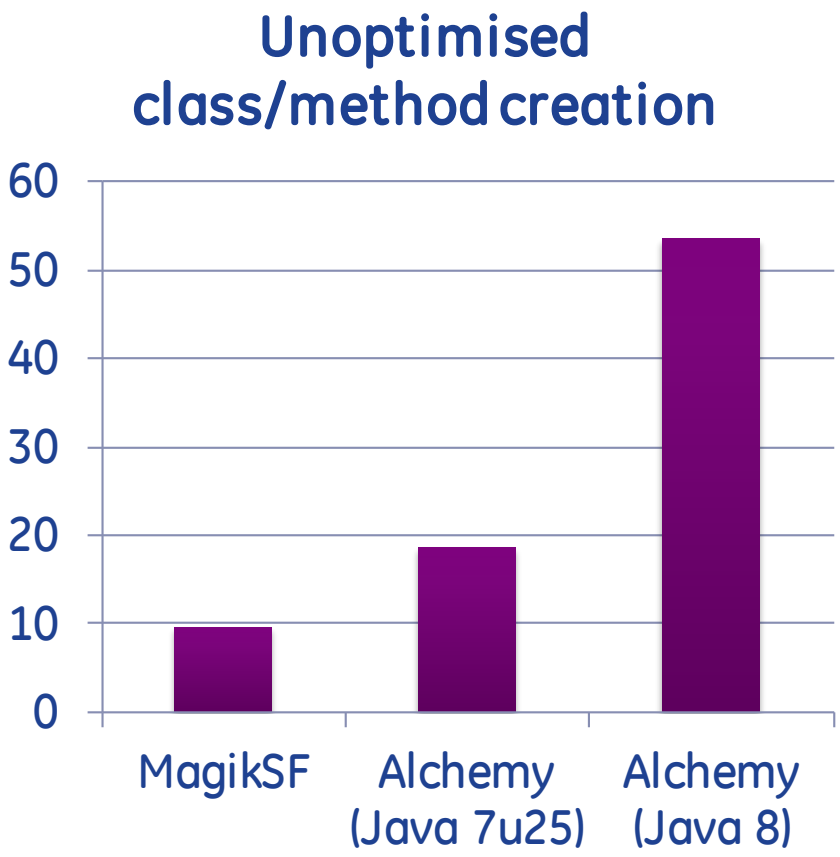
Opening the database… faster

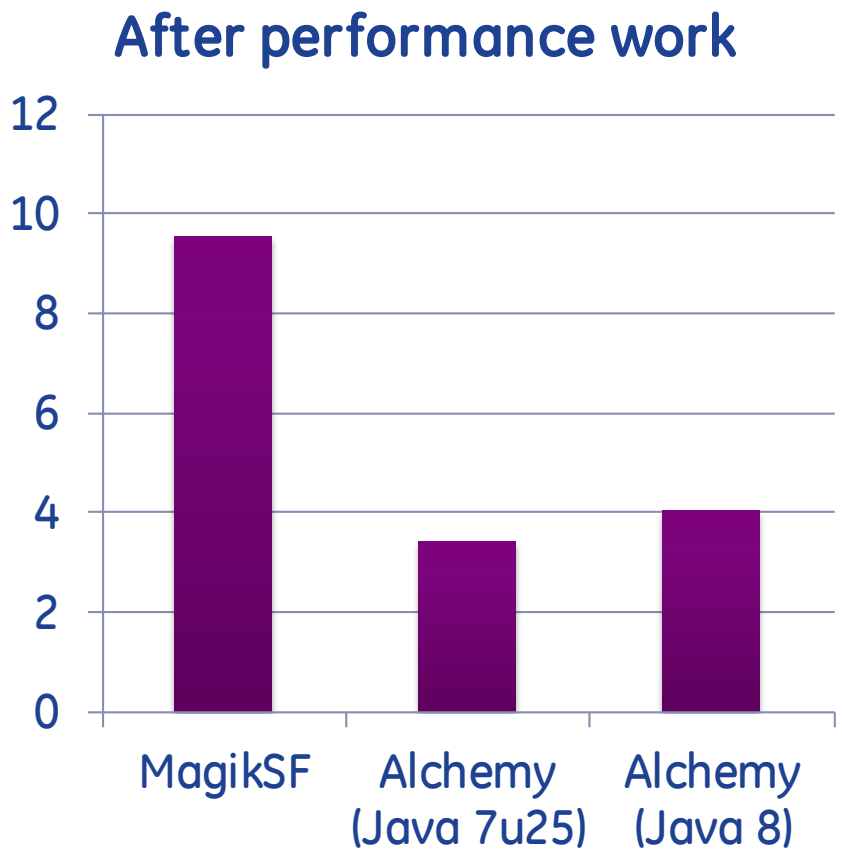


- Creating new classes requires reflection
  - Cache data on classes you know you'll need again
  - Be very careful about the reflection calls you make, some are faster than others
  - Be very focused about invalidating CallSites

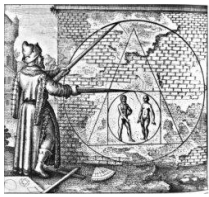
Why?




Unoptimised class/method creation
60
50
40
30
20
10
0
MagikSF
Alchemy (Java 7u25)
Alchemy (Java 8)

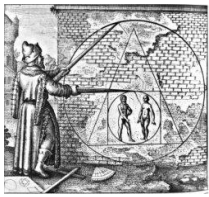

After performance work
12
10
8
6
4
2
0
MagikSF
Alchemy (Java 7u25)
Alchemy (Java 8)

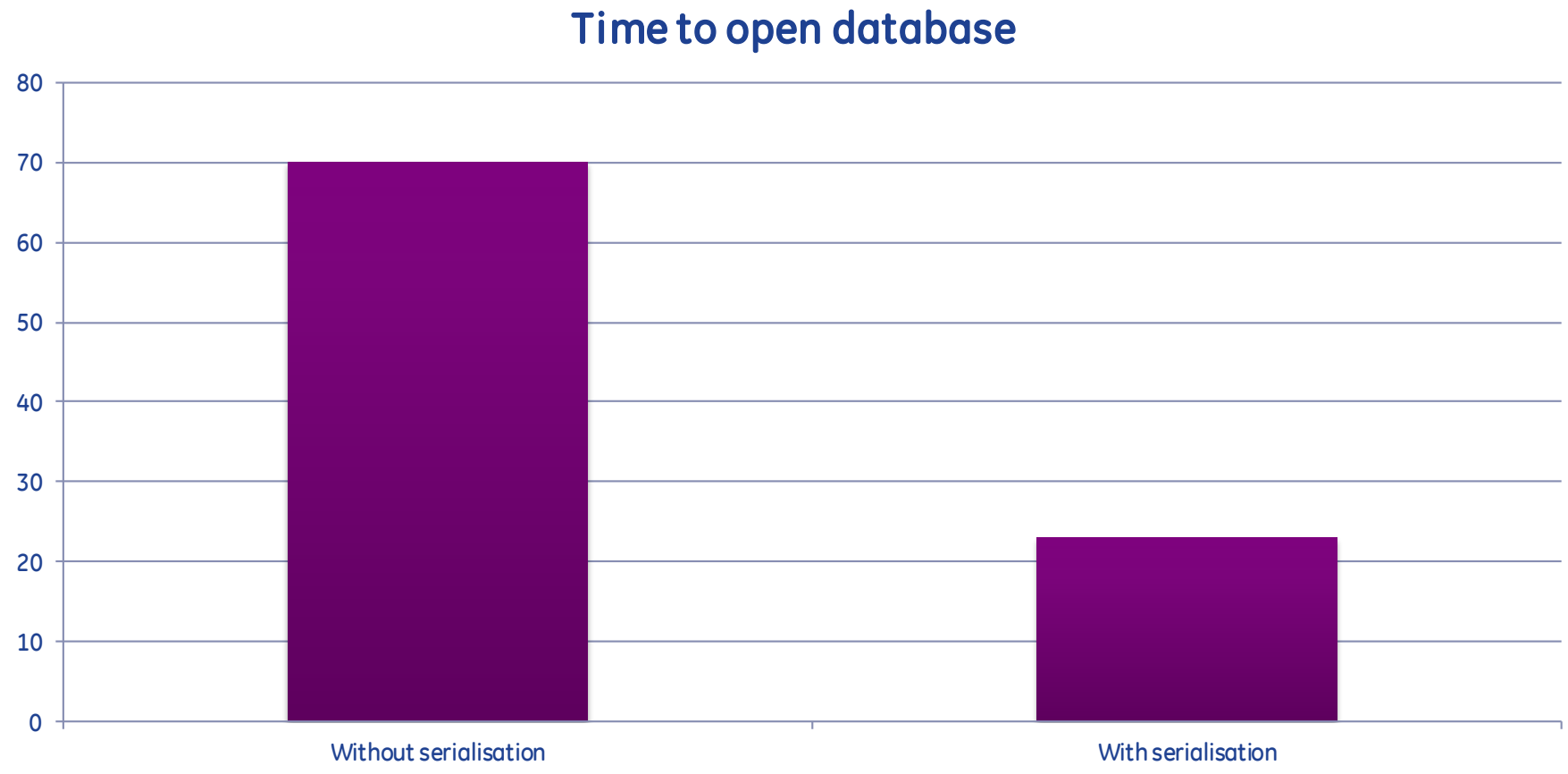
Serialisation



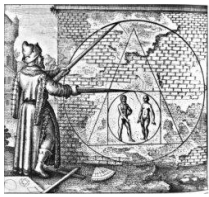
- Need to turn the whole data dictionary into a series of blobs
- Some things are hard to serialise
- Want to restore database connections in parallel—trial by fire for thread safety
- Additional work then needed to stitch everything back together

How much faster is it?



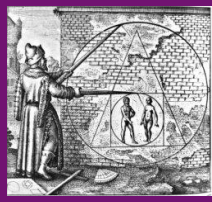


What's still to do?



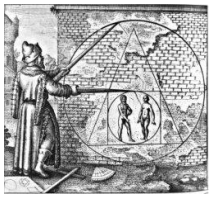
- Fix some concurrency bugs
- Test with more databases

Easing the transition



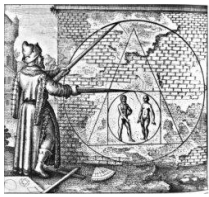
imagination at work

Need to help people moving from our old VM



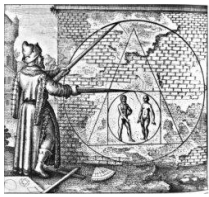
- Used to running tests with some form of coverage analysis
- Not used to threads being truly concurrent

Coverage of non-Java languages



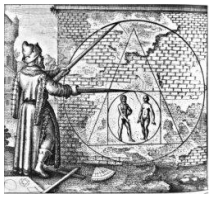
- Jacoco can compile coverage stats without any problems
  - We've run large test suites through it and nothing terrible happened
- Displaying those results is a different story

Coverage results



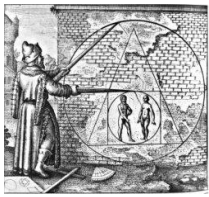
- Presenting coverage data for a large code base requires hierarchical display
- The packages for your classes need to be in a good hierarchy as well or the results still won't be managable
- If the class file contains info about the source file then use that, don't depend on Java source conventions

How much work is this?



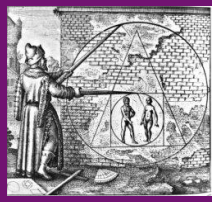
- Prototyping changes to Jacoco took an afternoon
- Haven't turned those into proper patches yet
- Changes to our compiler still to be done

Threading

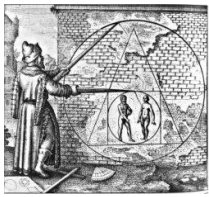


- We can instrument thread creation and use of atomic queues and locks
- That only catches the cases where people thought about thread safety at all

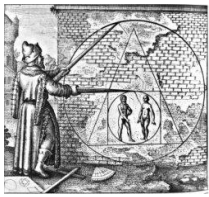
Summary



imagination at work



- MethodHandle caching and reuse is vital
- Serialisation and optimising meta-programming really help startup speeds
- No easy answers for finding concurrency issues



imagination at work