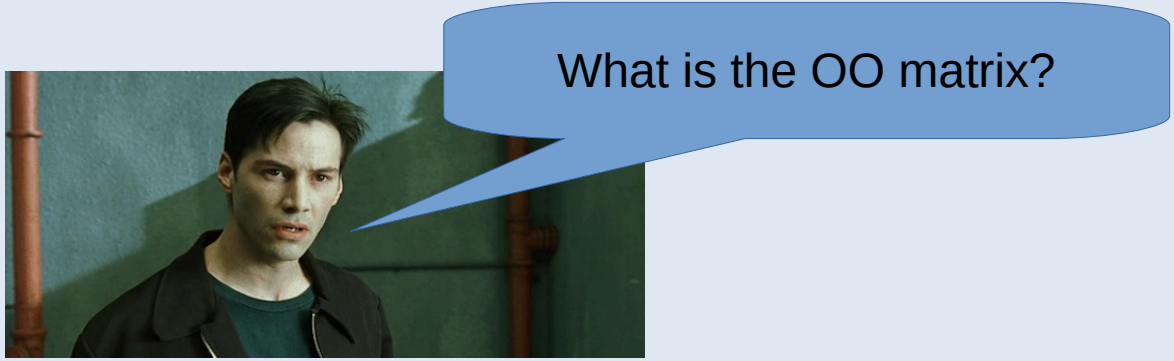
Dependency Injection

1/5
Inversion of Control

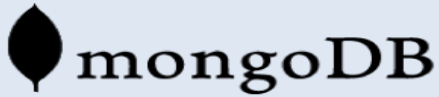


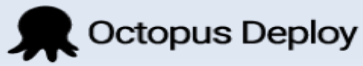
Slides:
http://officefloor.net/DDDPerth2019.pdf

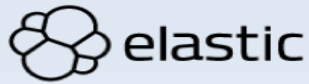
Daniel Sagenschneider
@sagenschneider

@sagenschneider
#oomatrix

Thanks to our Amazing Sponsors!







Dependency Injection


```
public class OldStyleRepository {

   public void save(Entity entity, DataSource dataSource){
      // Use data source parameter
   }
}
```

Dependency Injection

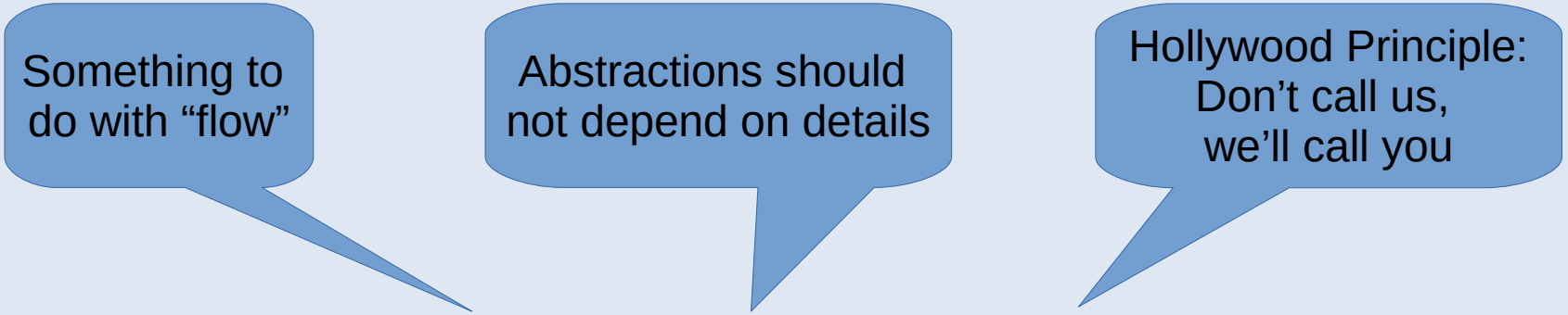
```
public class DependencyInjectionRepository {

   @Inject DataSource dataSource;

   public void save(Entity entity) {
      // Use injected data source
   }
}
```

We agree on it!

Well, except for field or constructor injection?

Inversion of Control

Control of what?

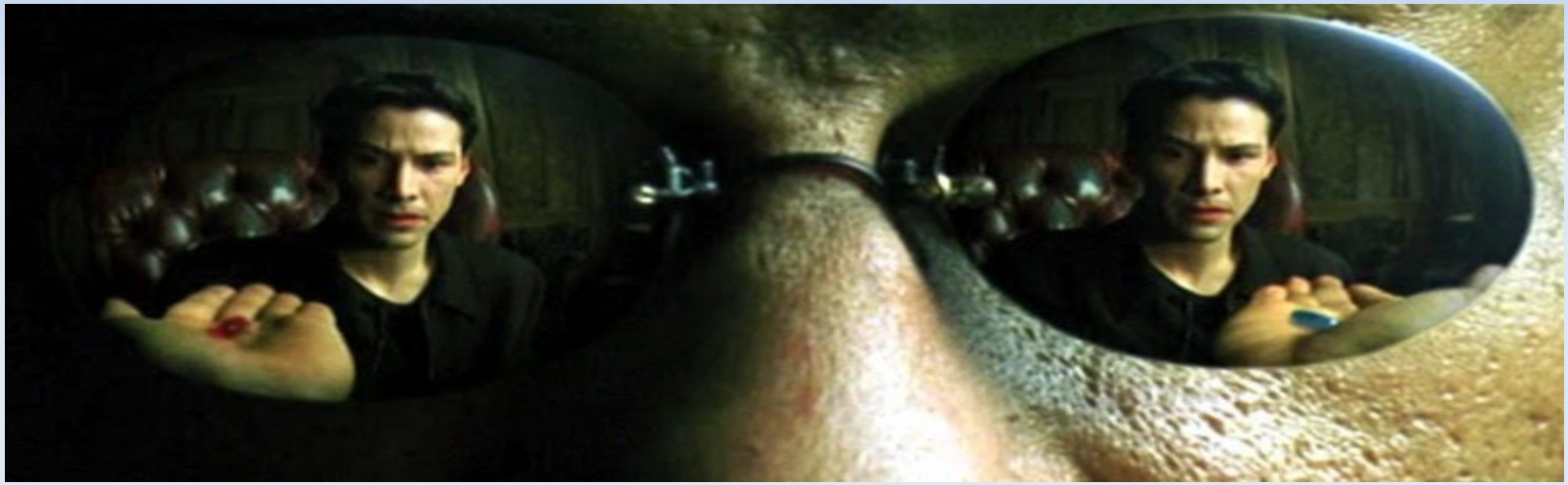


Stackoverflow full of arguments

https://stackoverflow.com/questions/3058/what-is-inversion-of-control

Red or Blue Pill

Would you like to know what’s right before your eyes?



but there is no going back once you see it

OO Matrix unseen 5 problems

DI changed the interface from

```
repository.save(entity,dataSource)
```

to

```
repository.save(entity)
```

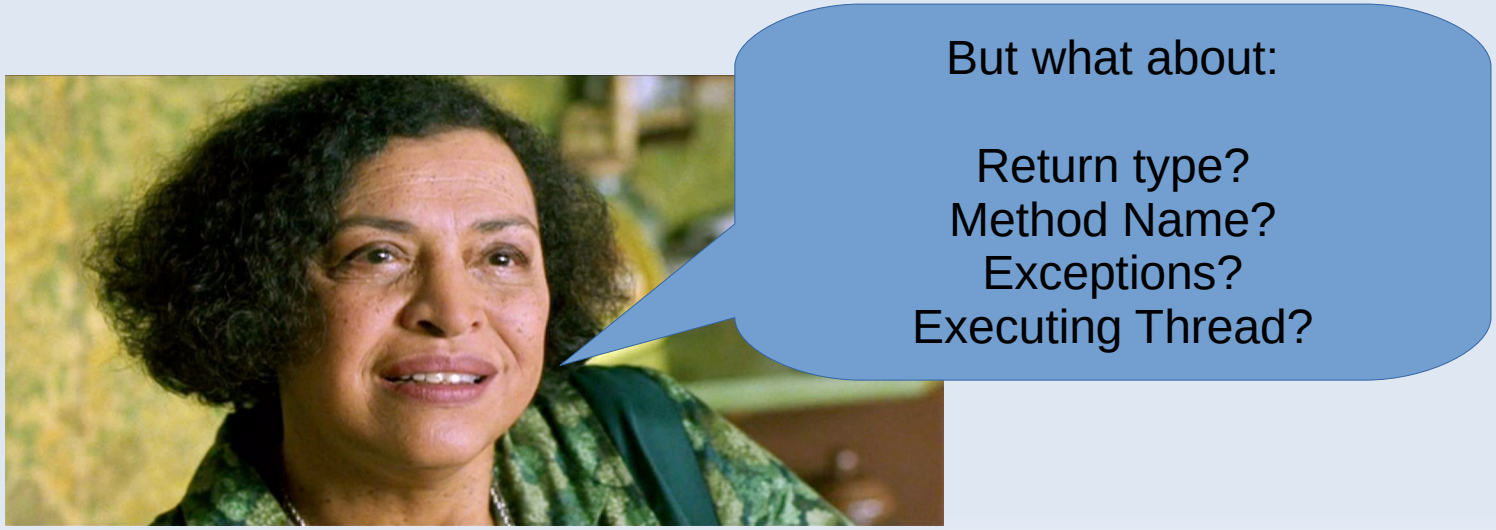
Only 1 problem has been solved

Parameter Coupling

Dependency Injection removes parameters

resulting in **looser coupling by caller**

```
R1 method(P1 p1, ~~P2 p2, P3 p3~~) throws E1, E2
```



5 Couplings of the Method

```
R1 methodName(P1 p1, P2 p2) throws E1, E2
```

Any change of the below requires **changing all client calls!**

- Return type
- Name of method
- Variable number of Parameters
- Handling various Exceptions
- Executing Thread (e.g. async vs sync)

OO Matrix

@sagenschneider
#oomatrix

Object Orientation Vision

Objects passing messages

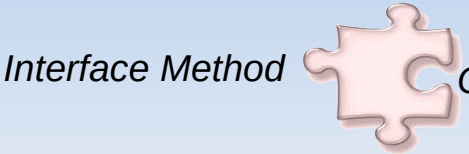
Alan Kay:

“OOP to me means **only messaging**, local retention and protection and hiding of state-process, and extreme late-binding of all things.”

http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

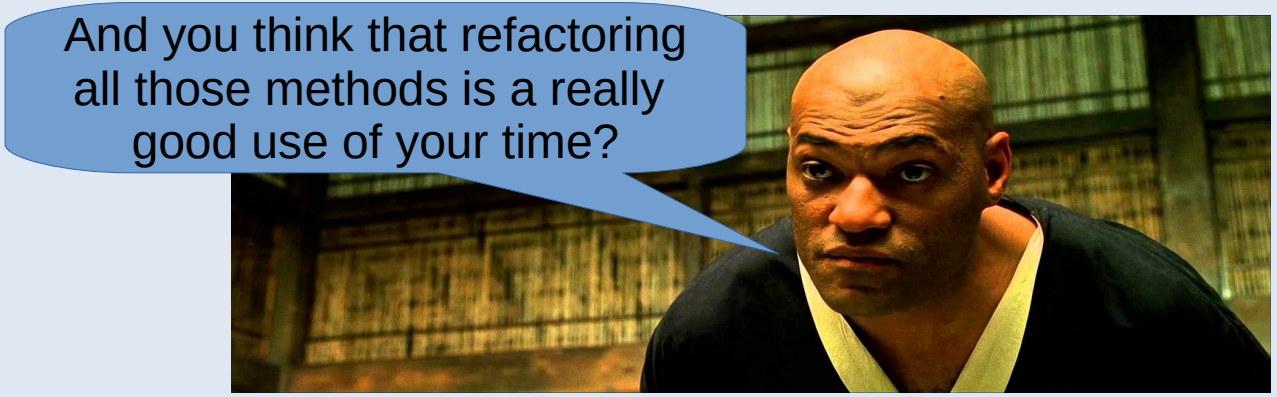
Mainstream Object Orientation

5 method couplings shaping objects differently



Object re-use difficult. It’s like using a jigsaw piece of one puzzle to complete another puzzle.

Method a problem ???



Let’s unplug you from the OO Matrix

Decouple Method Name


```
Consumer<T> f1 = (param) -> object.methodName(param);


// Client call now decoupled from method name
f1.accept(param)
```

Yes, boring but stay with me

Method name decoupled

Decouple Exceptions


```
Consumer<T> f1 = (param) -> {

  // Inject handlers for exceptions
  @Inject Consumer<E1> h1;
  @Inject Consumer<E2> h2;

  try {
    object.method(param);
  } catch (E1 e1) {
    h1.accept(e1);
  } catch (E2 e2) {
    h2.accept(e2);
  }
}
```

Pattern: Continuation Injection

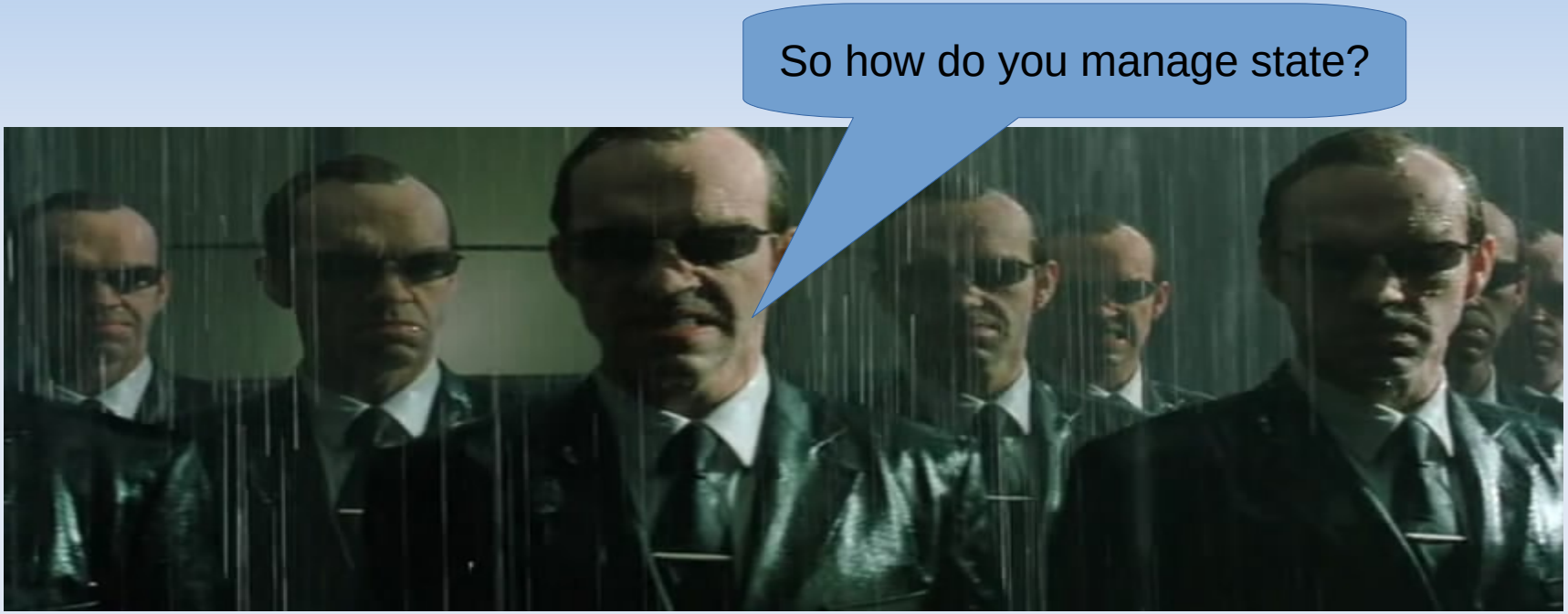
Name and Exceptions decoupled

Decouple Calling Thread


```
Consumer<T> f1 = (param) -> {

  // Inject Executor to run with appropriate thread
  @Inject Executor executor;

  executor.execute(() -> {
    object.method(param);
  });
}
```

Pattern: Thread Injection

No Return Value



Cache Dependencies

Pattern already used in Web Servers

Request Context

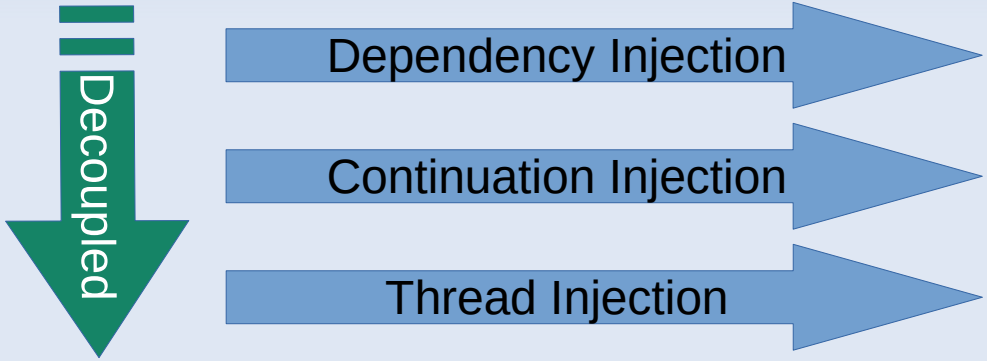
- Cache DI Objects within Request Context
- Re-used DI Objects pass state between methods

Inversion of (Coupling) Control

@sagenschneider
#oomatrix

“Non-changing” Client Interface

```
public interface Continuation<T> {
  void invoke(T message);
}
```



“Changing” Method Implementation
Injection provides decoupling

Inverts coupling **control** so method defines it

```
public class FirstClassProcedure
     implements Continuation<T> {

  public void invoke(T p1) {
    executor.execute(() -> {
      try {
        Implementation(p1, p2, p3, m1, m2);
      } catch (E1 e1) {
        h1.invoke(e1);
      } catch (E2 e2) {
        h2.invoke(e2);
      });
  }

  // Re-used DI objects to pass state
  @Inject P2 p2;
  @Inject P3 p3;

  // Continuations to call other methods
  @Inject Continuation<S> m1;
  @Inject Continuation<R> m2;
  // Continuations to handle exceptions
  @Inject Continuation<E1> h1;
  @Inject Continuation<E2> h2;

  // Inject the thread to execute the method
  @Inject Executor executor;


  private void implementation(
    T p1, P2 p2, P3 p3,
    Continuation<S> m1, Continuation<R> m2
  ) throws E1, E2 {

    // implementation logic
  }
}
```

DI vs IoC

@sagenschneider
#oomatrix

Boiler plate – can all be determined from method signature

OfficeFloor
introspects your
method signature
to run all this for you

DI Framework (e.g. Spring)

```
public class FirstClassProcedure
    implements Continuation<T> {

  public void invoke(T p1) {
    executor.execute(() -> {
      try {
        Implementation(p1, p2, p3, m1, m2);
      } catch (E1 e1) {
        h1.invoke(e1);
      } catch (E2 e2) {
        h2.invoke(e2);
      });
  }

  // Re-used DI objects to pass state
  @Inject P2 p2;
  @Inject P3 p3;

  // Continuations to call other methods
  @Inject Continuation<S> m1;
  @Inject Continuation<R> m2;
  // Continuations to handle exceptions
  @Inject Continuation<E1> h1;
  @Inject Continuation<E2> h2;

  // Inject the thread to execute the method
  @Inject Executor executor;

  private void implementation(
    T p1, P2 p2, P3 p3,
    Continuation<S> m1, Continuation<R> m2
  ) throws E1, E2 {

    // implementation logic
  }
}
```

We only write this

Function coupled like Method


```
R1 methodName(P1 p1, P2 p2) throws E1, E2
```

```
Either<R1,E1> functionName(P1 p1, P2 p2)
```

- Return type
- Name of function
- Variable number of Parameters
- Handling Exception (in return)
- Executing Thread

Mathematics say Invert

Functional programming “*perceived hard*”

“Invert, always invert”, Mathemetician Carl Jacobi 1820’s

(many hard problems can be clarified by re-expressing them in inverse form)

https://en.wikipedia.org/wiki/Carl_Gustav_Jacob_Jacobi

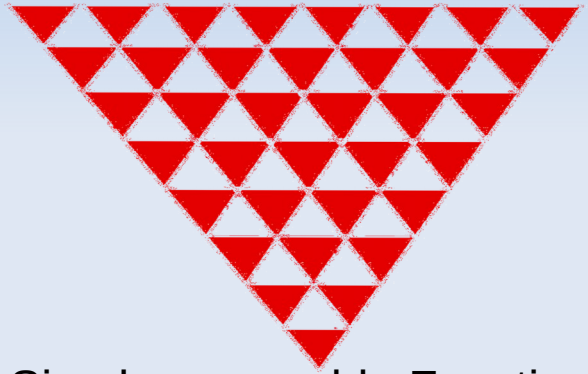
Maybe inverting functional programming will clarify it?

Inverting Functions

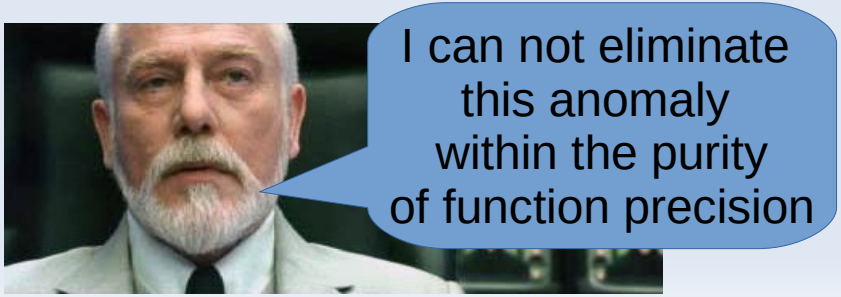
@sagenschneider
#oomatrix

Higher Order Functions

Explosion in function complexity
(limits system size)



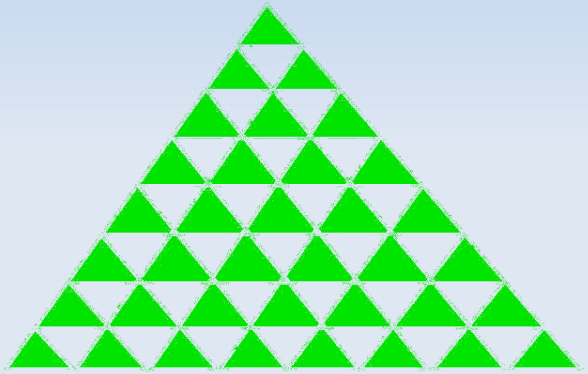
Simple re-useable Functions



more
Composition
less

IoC on Functions

Same Continuation interface



Encapsulated complexity
(enabling bigger systems)

Function IoC meets Actor principles

Thread Injection shares threads between Actors for increased scale/performance
(via Protothreads)

Nice Theory / Framework

Where is this practical?

Microservices *(heavy weight)*

@sagenschneider
#oomatrix

Client HTTP call

```
public class HttpClient
  implements Continuation<T> {

  @Inject WebClient client;

  public void invoke(T message) {
    client.post(message);
  }
}
```

Client Queue call

```
public class MessageProducer
  implements Continuation<T> {

  @Inject Queue queue;

  public void invoke(T message) {
    queue.send(message);
  }
}
```

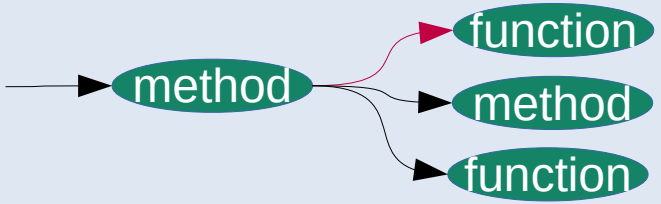
Microservice

```
public class ServicerImpl {

  @Inject SomeRepository rep;
  @Inject AnotherRepository anRep;

  @Inject Continuation<S> client;

  // Execution thread
  public void service(T message) {
    // use dependencies to service
  }
}
```

OO message passing

Alan Kay getting his vision

IoC is a light weight solution at the method / function level



Final Thought: I uncoupled from T in IT

Mechanical Mainstream Software
(“monolith”)

Method

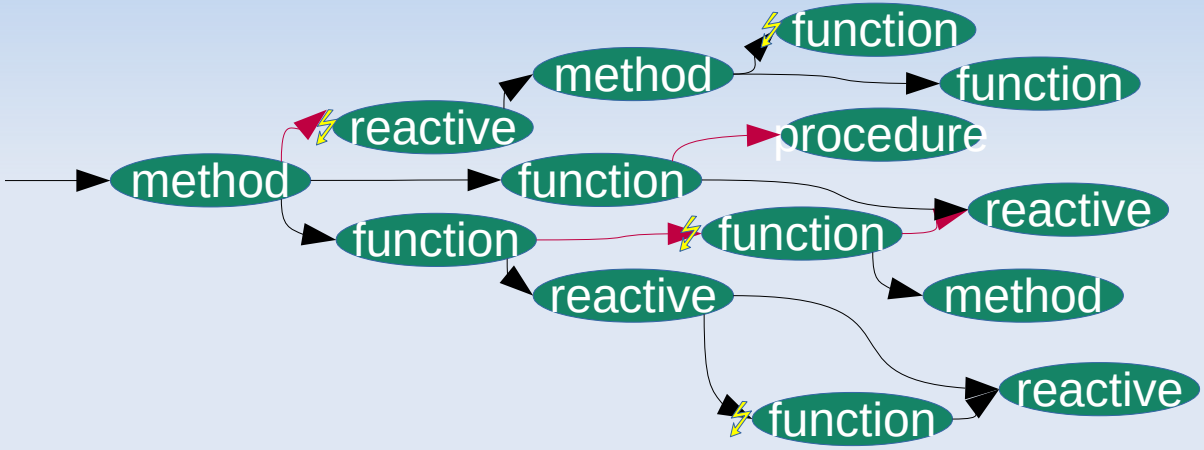
represents
Thread Stack

represents
The Machines

Interfaces via methods plug you into the Machines

Refactoring is expensive due to complex mechanical coupling

Organic IoC Software



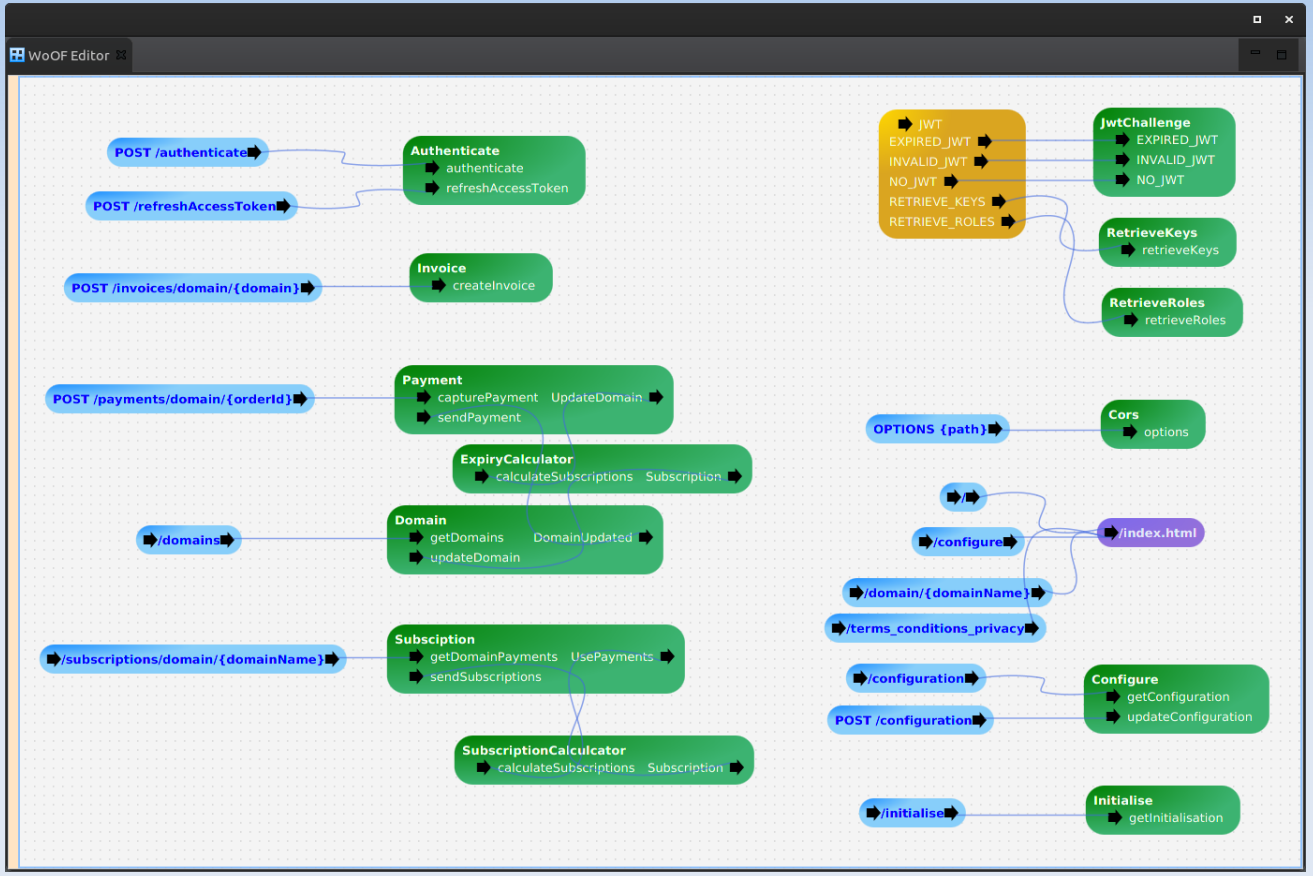
Refactoring
is now just
Reconfiguring

Example App

@sagenschneider
#oomatrix
http://officefloor.net

OfficeFloor
http://officefloor.net

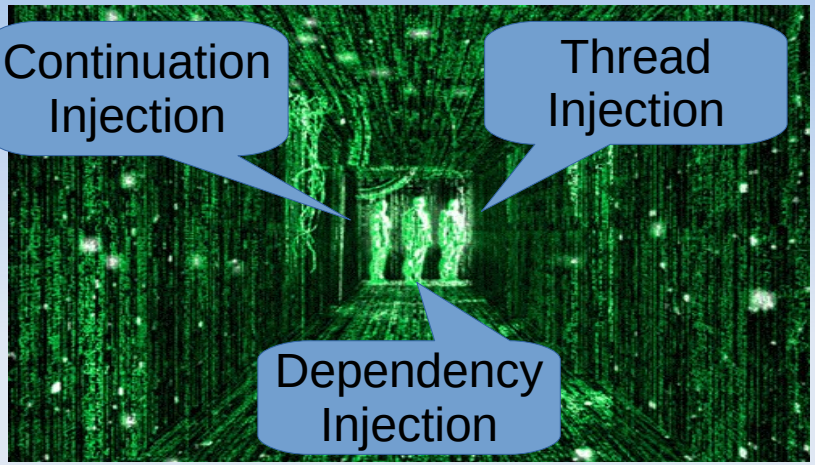
Production ready implementation



Summary

Dependency Injection
1/5
Inversion of (Coupling) Control

- Return Type (CI)
- Name of method / function (CI)
- Variable number of Parameters (DI)
- Handling various Exceptions (CI)
- Correct executing Thread (TI)



We're looking for Neo's to fight against the machines

Try out at: http://officefloor.net/tutorials

Read more at: http://sagenschneider.blogspot.com

Slides:
http://officefloor.net/DDDPerth2019.pdf

Daniel Sagenschneider
Founder of OfficeFloor

Appendices

No Behavioural References

Don’t create artifical objects so methods can access objects

Do not try to reference everything for the method

Only realise there is no behavioural object reference

Then you’ll see it is not the objects that behave, it is functions that reference

Tightly coupled methods

We’ve learnt to live with it

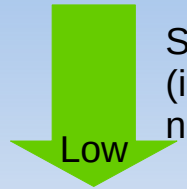
“refactoring” is just part of development

But should we live with it?

Low Coupling / High Cohesion

5 variations
(couplers)

Caller Coupling



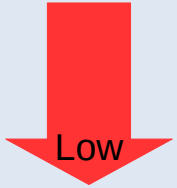
Standardised
(interchangeable,
no client impact)

```
R1 method(P1 p1) throws E1 {
  // method signature
}
```

```
@Inject P1 p1;
@Inject H1 h1;
@Inject Executor executor;
void firstClassProcedure() {
    executor.execute(() → {
        try {
            method(p1);
        } catch (E1 e1) {
            h1(e1);
        }
    })
}
```

OO jigsaw matrix

Vague component
boundaries,
e.g.:
- over referencing
- exception handling
- return types



Component Cohesion

Clear boundaries

Focused function

Synchronous + Asynchronous

First-Class Procedure allows choice to use both as necessary within the application

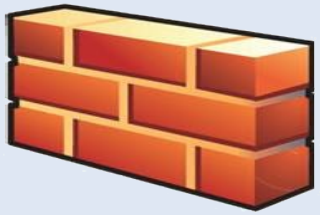
```
@Inject P1 p1;
@Inject F1 f1;
@Inject H1 h1;
@Inject Executor executor;
void firstClassProcedure() {
    executor.execute(() → {
        try {
            method(p1, f1);
        } catch (E1 e1) {
            h1(e1);
        }
    })
}
```

can embed sync within async

executor allows multiple threads

can not embed async within sync (e.g. thread per request)

async runs single threaded

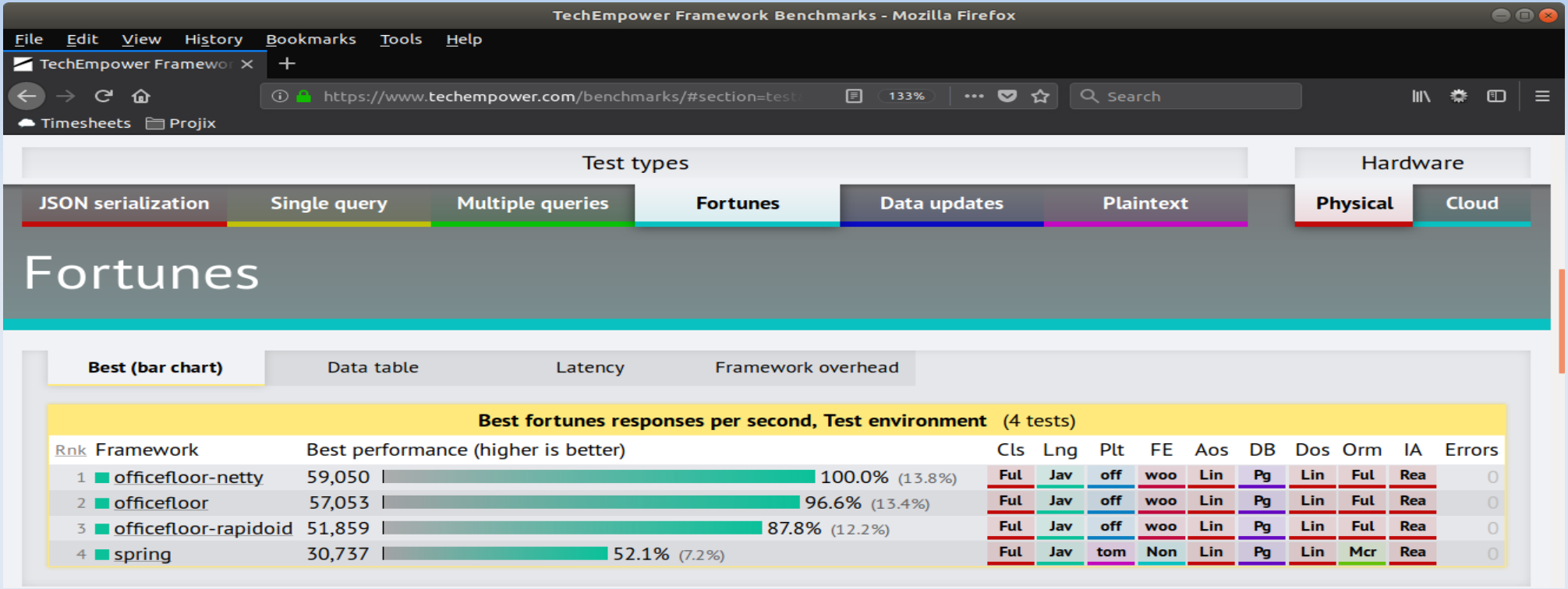


```
R1 synchronousMethod(P1 p1) {
  asynchonousMethod((result) → {
    // no callback thread
  });
  // no R1 value available yet
}
```

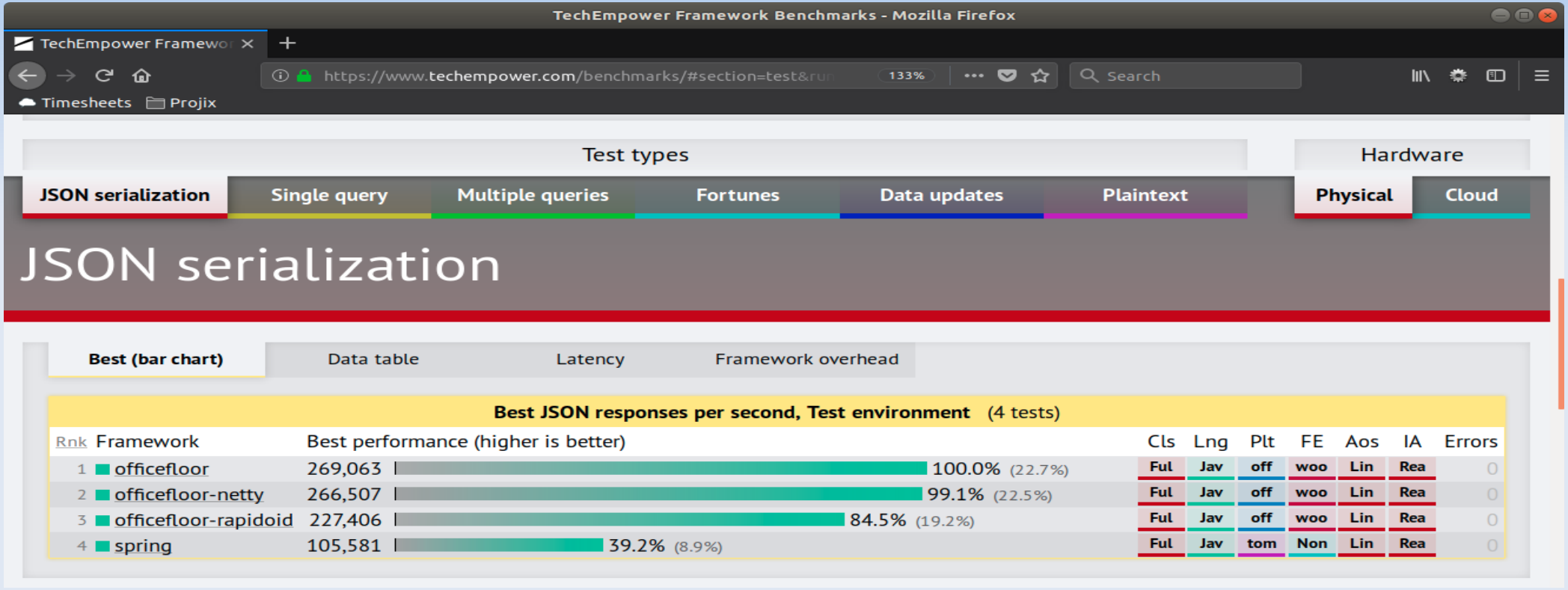
```
void asynchronousMethod(F1 callback) {
  // single thread blocked
  // whole application halts
  R1 r1 = blockingMethod();
  callback(r1);
}
```

Methods require choosing only one as application architecture

OfficeFloor vs Spring (DB)



OfficeFloor vs Spring (no DB)



Even higher performance as OfficeFloor knows no thread context switch is necessary

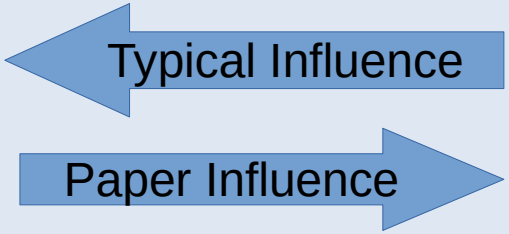
OfficeFloor

OfficeFloor: re-using Office patterns to improve Software Design

http://doi.acm.org/10.1145/2739011.2739013

(free download: http://www.officefloor.net/about.html)

Business Patterns
(solutions)



Software Patterns
(solutions)

People typically smart (lazy) and improve solution

Threads will happily loop forever

Other OfficeFloor concepts

Suppliers: Make use of other DI frameworks as object libraries

Administration: weaving of “aspects” (first-class procedures)

Governance: managing state in contexts (e.g. transactions)

Recursive typing for encapsulating complexity (e.g. modular sections created from other sub-modular sections)

- Previously not possible due to implicit threading assumptions

Executive: manages thread pools (e.g. sizes, thread affinity)

Client Continuation Interface

```
@FlowInterface
interface ClientContinuations {
  void flowOne(String message);
  void flowTwo(Results message);
}
```

Proxy built that provides implementation as such:

```
class ClientContinuationsImpl {
  @Inject Continuation<String> c1;
  @Inject Continuation<Results> c2;

  public void flowOne(String message) { c1.invoke(message); }
  public void flowTwo(Results message) { c2.invoke(message); }
}
```

Reduced Context Switching

```
@Inject P1 p1;
@Inject F1 f1;
@Inject H1 h1;
@Inject Executor executor;
firstClassProcedure() {
    Runnable runnable = () → {
        try {
            method(p1, f1);
        } catch (E1 e1) {
            h1(e1);
        }
    }
    if (executor.isOwner(Thread.currentThread()) {
        runnable.run();  // no context switch and carries on executing with current thread
    } else {
        exeuctor.execute(runnable);
    }
}
```

```
SynchronousExecutor implements Executor {
    void execute(Runnable runnable) {
        runnable.run();  // no context switch
    }
}
```