



Injection, Modularity, and Testing

An Architecturally Interesting Intersection

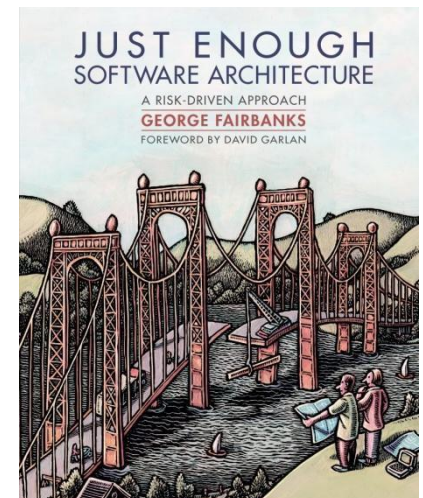
SATURN 2015
28 April 2015

George Fairbanks

Rhino Research

<http://RhinoResearch.com>

<http://GeorgeFairbanks.com>



Talk summary



Dependency injection

- Improves testability
- Without modularity, becomes too complex

Great example of “boiling the frog”

- When do mundane concerns become architecturally relevant?
- Daily rational decisions → still have trouble

Partial solutions

- Overcome cultural obstacles
- Overcome process obstacles

Story of a coffee maker



```
class CoffeeMaker {  
    Pump pump = new StandardPump();  
    Heater heater = new StandardHeater();  
  
    void brew() {  
        heater.on();  
        pump.pump();  
        wait(100);  
        print("Enjoy your coffee");  
        heater.off();  
    }  
}
```

Looks fine?

Great, let's test it.

Test the coffee maker



```
class CoffeeMakerTest {
    CoffeeMaker coffeeMaker;

    void setup() {
        Pump pump = new FakePump();
        Heater heater = new FakeHeater();
        coffeeMaker = new CoffeeMaker();
    }

    void testBrew() {
        coffeeMaker.brew();
        assert(...);
    }
}
```

To isolate dependencies,
we'll use a fake pump and
heater and ...

Oops, those are hard-
wired into the coffee
maker.

Let's fix that.

Coffee maker, try 2



```
class CoffeeMaker {  
    Pump pump;  
    Heater heater;  
  
    CoffeeMaker(Pump pump, Heater heater) {  
        this.pump = pump;  
        this.heater = heater;  
    }  
  
    void brew() {  
        ...  
    }  
}
```

Ok, now the pump and heater are passed in during construction.

Test the coffee maker, try 2



```
class CoffeeMakerTest {  
    CoffeeMaker coffeeMaker;  
  
    void setup() {  
        coffeeMaker = new CoffeeMaker(  
            new FakePump(),  
            new FakeHeater());  
    }  
  
    void testBrew() {  
        coffeeMaker.brew();  
        assert(...);  
    }  
}
```

Yay!

But now who knows how
to create a CoffeeMaker?

Coffee maker, try 3



```
class CoffeeMaker {
    Pump pump;
    Heater heater;

    @Inject
    CoffeeMaker(Pump pump, Heater heater) {
        this.pump = pump;
        this.heater = heater;
    }

    void brew() {
        ...
    }
}
```

Now the dependency injection container injects the parameters as needed.

JSR-330 defines:

- `@Inject`
- `Provider.get()`

Problem solved?

Test the coffee maker, try 2



```
class CoffeeMakerTest {  
    CoffeeMaker coffeeMaker;  
  
    void setup() {  
        coffeeMaker = new CoffeeMaker(  
            new FakePump(),  
            new FakeHeater());  
    }  
  
    void testBrew() {  
        coffeeMaker.brew();  
        assert(...);  
    }  
}
```

What about
their
dependencies?

The pump needs a water source!

The heater needs an energy source!

The dependencies are not one-level -- they are a graph!

Test the coffee maker, try 3



```
class CoffeeMakerTest {
    CoffeeMaker coffeeMaker;

    void setup() {
        Injector injector = magic();
        injector.add(FakePump.class);
        injector.add(FakeHeater.class);
        // and their dependencies, etc.
        coffeeMaker =
            injector.create(CoffeeMaker.class);
    }

    void testBrew() {
        ...
    }
}
```

Use the injector to create the CoffeeMaker.

Must populate the injector's graph of objects, so it can inject the Pump and Heater.

Ugh, setting up the injector's graph is tedious.

Let's make a helper method!

Test the coffee maker, try 4



```
class CoffeeMakerTest {
    CoffeeMaker coffeeMaker;

    void setup() {
        Injector injector = magic();
        myTestSetup(injector);
        coffeeMaker =
            injector.create(CoffeeMaker.class);
    }

    void testBrew() {
        coffeeMaker.brew();
        assert(...);
    }
}
```

Good

- Shared test setup
- Handles recursive dependencies

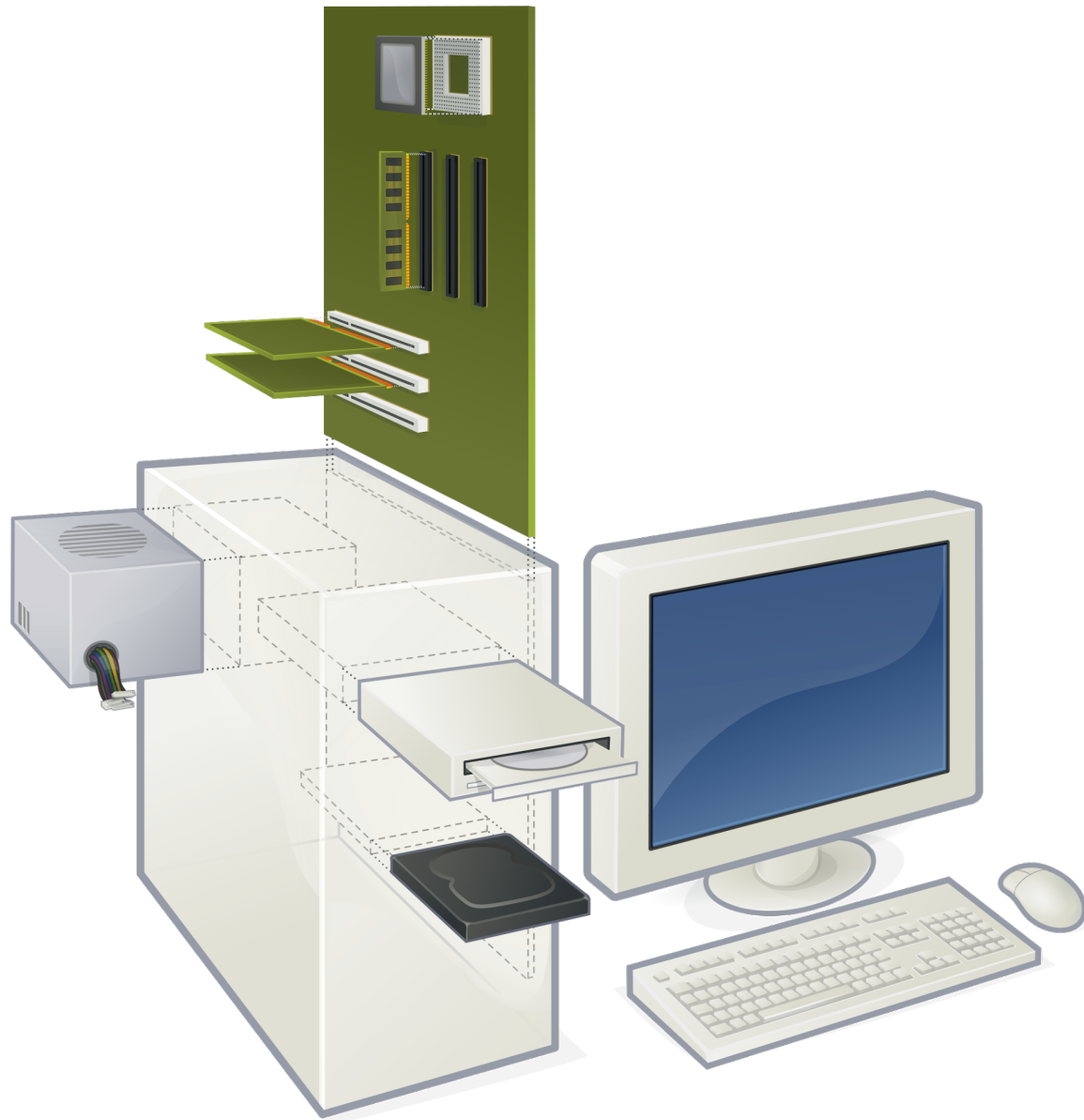
Bad

- Test setup is complex
-- do you trust it?
- myTestSetup()
becomes a dumping
ground
- Big app = lots of
dependencies

Parts only (no assembly instructions)



Modular assembly





- **Dependency Injection**

- Big graph of dependencies
- ... that change daily

- **Package Modularity**

- Injection granularity: classes, facades, modules?
- Usually: classes

- **Testing**

- Fussy: many mock/fake dependencies
- Inertia: Test case LOC > application LOC

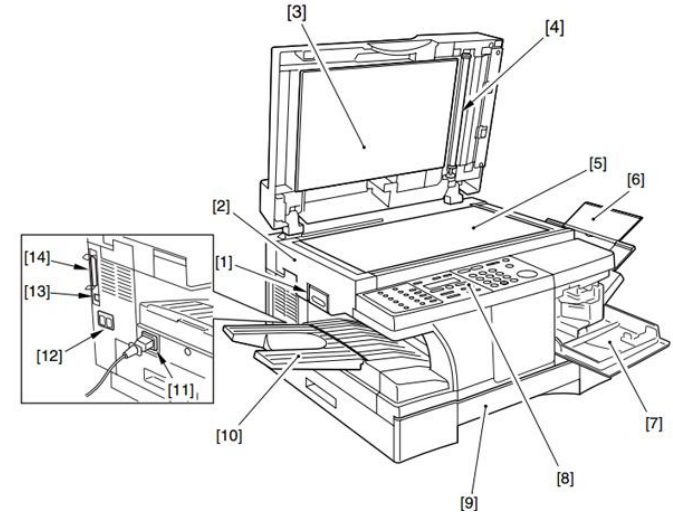
- Many fine-grained dependencies changing daily
- Forest-through-the-trees: which configurations are legal?
- Hard to get **intellectual control**

Photocopier example



Single legal configuration of photocopier

- Input: Manual-feed
- Engine: 30 copies per minute
- Output: stapling



BNF: All legal configurations

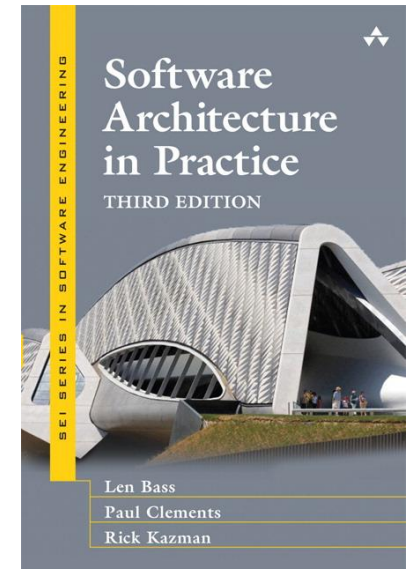
```
<photocopier> ::= <input> <engine> <output>
<input>       ::= "manual-feed input" | "auto-feed input"
<engine>      ::= "15cpm" | "30cpm" | "60cpm"
<output>      ::= "output tray" | "stapling output tray"
```

Compare with: all the screws and bolts

Architects want intellectual control



- **Concerns about modules**
 - How is the system **decomposed** into modules?
 - What are the **dependencies** between modules?
 - What are the **legal arrangements** of modules?
 - What is the **granularity** of the modules?
- **Concerns about testing**
 - Is the system **testable**?
 - Is the test infrastructure **maintainable**?



Missing abstraction: a System Configuration



- **What is a system configuration?**
 - Group code into modules
 - Describe legal arrangements of modules
- **Abstractions yield intellectual control**
 - **Zoom out:** Save us from thinking of every last screw and bolt
 - **Type vs. instance:** All legal system configurations vs. one
- **Common configuration instances**
 - Run locally
 - Run unit or integration tests
 - Run in production
- **Common configuration types**
 - <I never see these>

Hypothetical dependency injection timeline



- Start with simple system, simple dependency injection (fine-grained).
- Team grows, system gets bigger, refactors test setup into shared code.
- Shared test setup becomes complex in its own right. Rules for configuring a legal system are implicit and shared across developers.
- Shared test setup becomes a “write-only” dumping ground for bindings.

When would you “add architecture” to the timeline?



DeRemer and Kron, 1975,

“Programming-in-the-Large Versus Programming-in-the-Small”

However, current languages discourage the accurate recording of the overall solution structure; they force us to write programs in which we are so preoccupied with the trees that we lose sight of the forest, as do the readers of our programs!

An MIL should provide a means for the programmer(s) of a large system to express their intent regarding the overall program structure in a concise, precise, and checkable form. Where an MIL is not available, module interconnectivity information is usually buried partly in the modules, partly in an often amorphous collection of linkage-editor instructions, and partly in the informal documentation of the project. Aside from the issue that each of these three areas is ill-suited to express interconnectivity, the smearing of the relevant information over disjoint media is highly unreliable.

**You are the architect
Steer the project
Good luck**

Imagine this timeline



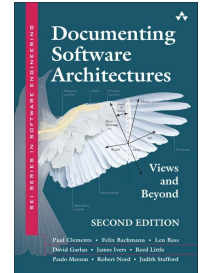
- **A software project starts.** Feature-driven using short iterations.
- **Project ramps up.** More people. Improved support infrastructure including robust test harnesses, continuous integration, automated deployments.
- **Continued growth.** Some parts treated as “legacy” but well-tested and usable. Complexity is everywhere.
- **Velocity slows.** Significant new developer on-boarding time because of complexity.
- **Developers propose refactoring** of baked-in assumptions that would take many months/years, so it’s hard to make a cost-benefit argument.
- Management chooses to **rewrite the system.**

When did any concern become architecturally relevant?

Practicing architecture is hard today



- **Document everything?** We could go “Documenting Software Architectures”, but:
 - Poor track record of keeping docs updated
 - Poor track record of developers reading docs
 - Debatable track record of “Me architect. You developer.”
- **Architecture theory seems to be reasonably well-sorted**
- **Culture obstacles**
 - Increasing disdain for abstractions & delayed gratification (see: YAGNI)
- **Process obstacles**
 - In practice, many teams seem to do better with iterative feature-driven sprints and refactoring
 - Limited success reports of agile + architecture



What should we do?



- **Culture of architecture**

- Get architecture off the whiteboard and into the code
 - Simon Brown's work on C4
 - Architecturally evident coding style
- Microservices movement
 - No, it's not the only architectural style
 - Architecture terms being used by agile / iterative teams

- **Architecture in process**

- Architecture within short iterations
 - Michael Keeling's work (and upcoming book!)
- Better tooling, frameworks
 - Today: great tooling to support iterations, continuous integration, automated deployments, etc
 - Tomorrow: great tooling to make architecture abstractions visible (Structurizr)

Moral of the story



- Dependency Injection: Great example of “boiling the frog”
 - When do mundane concerns become architecturally relevant?
 - Daily rational decisions → still have trouble
- Let’s show how to “do architecture”
 - We know how to think about it
 - What is best practice for feature-driven iterative development?