dist✪ge

Modern Staged Dependency Injection for Scala

Modular Functional Programming
with
Context Minimization
through
Garbage Collection

Septimal Mind Ltd

team@7mind.io

The motivation behind DI pattern and DI frameworks

1. Systems we work with may be represented as graphs. Nodes are components (usually instances), edges are references,
2. Graph transformation complexity grows nonlinearly with nodes count (need to add one constructor parameter, have to modify *k* classes),
3. Graph composition has combinatoric complexity (need to run tests choosing between mock/production repositories and external APIs, have to write four configurations). // too detached and unappealing

We have several possible options to address these problems:

1. Singletons and Factories: solves tight coupling but expensive tests and refactorings,
2. Service Locator: bit less coupling but still expensive,
3. Dependency Injection: less invasive and supports isolation but requires more complex machinery.

"DI doesn't compose with FP": Problems

1. Typical DI framework is OOP oriented and does not support advanced concepts required for modern FP (typeclasses, higherkinded types),
2. Almost all the DI frameworks are working in runtime while many modern FP concepts are compiletime by their nature,
3. Less guarantees: program which compiles correctly can break on wiring in runtime. After a huge delay,
4. Wiring is non-determenistic: Guice can spend several minutes trying to reinstantiate heavy instance multiple times (once per dependency) then fail,
5. Wiring is opaque: it's hard or impossible to introspect the context. E.g. in Guice it's a real pain to close all the instantiated `Closeables`. Adding missing values into the context (config injections) is not trivial as well.

“DI doesn’t compose with FP”: Notes

1. We have some compiletime DI frameworks or mechanisms (see `MacWire`) allowing us to implement DI as pattern though purely compiletime tools are not convenient when we have to deal with purely runtime entities (like plugins and config values),
2. Graph composition problem is not addressed by any existing tool.

DI implementations are broken. . .

. . . so we may build better one, which must:

1. be wellintegrated with type system of our target language (higherkinded types, implicits, typeclasses),
2. allow us to introspect and modify our context on the fly,
3. be able to detect as many as possible problems quickly, better during compilation,
4. give us a way to stop making atomic or conditional contexts.

Staged approach

1. Let's apply *Late Binding*,
2. let's collect our graph information first,
3. then build a DAG representing our context (so-called *Project Network*, let's call it *Plan*),
4. then analyse this graph for errors (missing references, conflicts),
5. then apply additional transformations,
6. then interpret the graph.

This is a cornercase of more generic pattern – PPER (Percept, Plan, Execute, Repeat).

Staged approach: outcome

What we get:

1. Planner is *pure*: it has no sideeffects,
2. A plan is a Turing-incomplete program for a simple machine. It will always terminate in known finite time,
3. An interpreter may perform instantiations at runtime or. . . just generate Scala code that will do that when compiled,
4. All the job except of instantiations can be done in compiletime,
5. Interpreter is free to run independent instantiations in parallel,
6. Extremely important: we can transform (rewrite) the plan before we run iterpreter.

Compile-Time and Runtime DI

TODO: illustration

Extension: Configuration Support

```scala
trait T {

}
```

1. hi

   `val x = 1`

Pattern: Plan Completion

1.

The Principle Behind: PPER

1.

Garbage Collector

1.

Context Minimization

1.

Context Minimization for Tests

1.

Context Minimization for Deployment

1.

Kind-Polymorphic Type Tags

1.

Typeclass instance injection (Implicit Injection)

1.

Lambda injection and Parameter Magnet

1.

Code example: IO Injection

1.

Code example: Tagless Final Style

1.

Dynamic Plugins

1.

Tags

1.

Plan Introspection

1.

Trait Completion

1. Runtime and Compile-time.

Factory Methods (Assisted Injection)

1. Useful for Akka, lot more convenient than Guice,
2. Runtime and Compile-time.

Status and things to do

`distage` is:

1. ready to use,
2. in real production,
3. all Runtime APIs are available,
4. Compile-time verification, trait completion, assisted injections and lambda injections are available.

Our plans:

1. Refactor Roles API,
2. Support running Producer within a monad (IO),
3. Support Scala.js,
4. Support optional isolated classloaders (in foreseeable future),
5. Publish compile-time Producer,
6. Check our GitHub: https://github.com/pshirshov/izumi-r2.

distage is just a part of our stack

We have a vision backed by our tools:

1. Idealingua: transport and codec agnostic gRPC alternative with rich modeling language,
2. LogStage: zero-cost logging framework,
3. *Fusional Programming and Design* guidelines. We love both FP and OOP,
4. *Continous Delivery* guidelines for Role-based process,
5. *Percept-Plan-Execute* Generative Programming approach, abstract machine and computational model. Addresses Project Planning (see Operations Research). Examples: orchestration, build systems.

Altogether these things already allowed us to significantly reduce development costs and delivery time for our client.

More slides to follow.

Teaser: LogStage

Teaser: Idealingua

Thank you for your attention

https://izumi.7mind.io/

We're looking for clients, contributors, adopters and colleagues ;)

About the author:

1. coding for 18 years, 10 years of hands-on commercial engineering experience,
2. has been leading a cluster orchestration team in Yandex, "the Russian Google",
3. implemented "*Interstellar Spaceship*" – an orchestration solution to manage 50K+ physical machines across 6 datacenters,
4. Owns an Irish R&D company, https://7mind.io,
5. Contacts: team@7mind.io,
6. Github: https://github.com/pshirshov