http://www.yegor256.com/2014/10/03/di-containers-are-evil.html

DI Containers are Code Polluters

3 October 2014 modified on 20 October 2014 Yegor Bugayenko

While dependency injection (aka, "DI") is a natural technique of composing objects in OOP (known long before the term was introduced by Martin Fowler), Spring IoC, Google Guice, Java EE6 CDI, Dagger and other DI frameworks turn it into an anti-pattern.

I'm not going to discuss obvious arguments against "setter injections" (like in Spring IoC) and "field injections" (like in PicoContainer). These mechanisms simply violate basic principles of object-oriented programming and encourage us to create incomplete, mutable objects, that get stuffed with data during the course of application execution. Remember: ideal objects must be immutable and may not contain setters.

Instead, let's talk about "constructor injection" (like in Google Guice) and its use with **dependency injection containers**. I'll try to show why I consider these containers a redundancy, at least.

What is Dependency Injection?

This is what dependency injection is (not really different from a plain old object composition):

```
public class Budget {
  private final DB db;
  public Budget(DB data) {
```

```
    this.db = data;
  }
  public long total() {
    return this.db.cell(
      "SELECT SUM(cost) FROM ledger"
    );
  }
}
```

The object `data` is called a "dependency".

A `Budget` doesn't know what kind of database it is working with. All it needs from the database is its ability to fetch a cell, using an arbitrary SQL query, via method `cell()`. We can instantiate a `Budget` with a PostgreSQL implementation of the `DB` interface, for example:

```
public class App {
  public static void main(String... args) {
    Budget budget = new Budget(
      new Postgres("jdbc:postgresql:5740/main")
    );
    System.out.println("Total is: " + budget.total());
  }
}
```

In other words, we're "injecting" a dependency into a new object `budget`.

An alternative to this "dependency injection" approach would be to let `Budget` decide what database it wants to work with:

```
public class Budget {
  private final DB db = new Postgres("jdbc:postgresql:5740/main");
  // class methods
}
```

This is very dirty and leads to 1) code duplication, 2) inability to reuse, and 3) inability to test, etc. No need to discuss why. It's obvious.

Thus, dependency injection via a constructor is an amazing technique. Well, not even a technique, really. More like a feature of Java and all other object-oriented languages. It's expected that almost any object will want to encapsulate some knowledge (aka, a "state"). That's what constructors are for.

What is a DI Container?

So far so good, but here comes the dark side — a dependency injection container. Here is how it works (let's use Google Guice as an example):

```
import javax.inject.Inject;
public class Budget {
  private final DB db;
  @Inject
  public Budget(DB data) {
    this.db = data;
  }
  // same methods as above
}
```

Pay attention: the constructor is annotated with `@Inject`.

Then, we're supposed to configure a container somewhere, when the application starts:

```
Injector injector = Guice.createInjector(
  new AbstractModule() {
    @Override
    public void configure() {
      this.bind(DB.class).toInstance(
        new Postgres("jdbc:postgresql:5740/main")
      );
    }
  }
);
```

Some frameworks even allow us to configure the injector in an XML file.

From now on, we are not allowed to instantiate `Budget` through the `new` operator, like we did before. Instead, we should use the injector we just created:

```
public class App {
  public static void main(String... args) {
    Injection injector = // as we just did in the previous snippet
    Budget budget = injector.getInstance(Budget.class);
    System.out.println("Total is: " + budget.total());
  }
}
```

The injection automatically finds out that in order to instantiate a `Budget` it has to provide an argument for its constructor. It will use an instance of class `Postgres`, which we instantiated in the injector.

This is the right and recommended way to use Guice. There are a few even darker patterns, though, which are possible but not recommended. For example, you can make your injector a singleton and use it right inside the `Budget` class. These mechanisms are considered wrong even by DI container makers, however, so let's ignore them and focus on the recommended scenario.

What Is This For?

Let me reiterate and summarize the scenarios of **incorrect usage** of dependency injection containers:

- Field injection
- Setter injection
- Passing injector as a dependency

- Making injector a global singleton

If we put all of them aside, all we have left is the constructor injection explained above. And how does that help us? Why do we need it? Why can't we use plain old `new` in the main class of the application?

The container we created simply adds more lines to the code base, or even more files, if we use XML. And it doesn't add anything, except an additional complexity. We should always remember this if we have the question: "What database is used as an argument of a Budget?"

The Right Way

Now, let me show you a real life example of using `new` to construct an application. This is how we create a "thinking engine" in rultor.com (full class is in `Agents.java`):

```
final Agent agent = new Agent.Iterative(
  new Array<Agent>(
    new Understands(
      this.github,
      new QnSince(
        49092213,
        new QnReferredTo(
          this.github.users().self().login(),
          new QnParametrized(
            new Question.FirstOf(
              new Array<Question>(
                new QnIfContains("config", new QnConfig(profile)),
                new QnIfContains("status", new QnStatus(talk)),
                new QnIfContains("version", new QnVersion()),
                new QnIfContains("hello", new QnHello()),
                new QnIfCollaborator(
                  new QnAlone(
                    talk, locks,
                    new Question.FirstOf(
                      new Array<Question>(
                        new QnIfContains(
                          "merge",
                          new QnAskedBy(
                            profile,
                            Agents.commanders("merge"),
                            new QnMerge()
                          )
                        ),
                        new QnIfContains(
                          "deploy",
```

```
                        new QnAskedBy(
                          profile,
                          Agents.commanders("deploy"),
                          new QnDeploy()
                        )
                      ),
                      new QnIfContains(
                        "release",
                        new QnAskedBy(
                          profile,
                          Agents.commanders("release"),
                          new QnRelease()
                        )
                      )
                    )
                  )
                )
              )
            )
          )
        )
      )
    ),
    new StartsRequest(profile),
    new RegistersShell(
      "b1.rultor.com", 22,
      "rultor",
      IOUtils.toString(
        this.getClass().getResourceAsStream("rultor.key"),
        CharEncoding.UTF_8
      )
    ),
    new StartsDaemon(profile),
    new KillsDaemon(TimeUnit.HOURS.toMinutes(2L)),
    new EndsDaemon(),
    new EndsRequest(),
    new Tweets(
      this.github,
      new OAuthTwitter(
        Manifests.read("Rultor-TwitterKey"),
        Manifests.read("Rultor-TwitterSecret"),
        Manifests.read("Rultor-TwitterToken"),
        Manifests.read("Rultor-TwitterTokenSecret")
      )
    ),
    new CommentsTag(this.github),
    new Reports(this.github),
    new RemovesShell(),
    new ArchivesDaemon(
      new ReRegion(
        new Region.Simple(
          Manifests.read("Rultor-S3Key"),
          Manifests.read("Rultor-S3Secret")
        )
      ).bucket(Manifests.read("Rultor-S3Bucket"))
    ),
    new Publishes(profile)
  )
);
```

Agents.java hosted with ❤ by **GitHub** **view raw**

Impressive? This is a true object composition. I believe this is how a proper object-oriented application should be instantiated.

And DI containers? In my opinion, they just add unnecessary noise.