Dependency Injection & Toothpick

https://github.com/stephanenicolas/toothpick

About us

Stéphane Nicolas

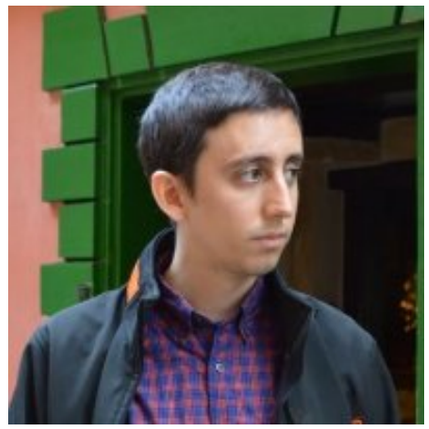
Senior Android Dev at

Groupon

stephanenicolas

snicolas

+stephane nicolas



Daniel Molinero

Android Dev at

Groupon

dlemures

@D_Lemures

+DaniLemures

- Introduction
- Dependency Injection
- Libraries
- Toothpick
- Toothpick Example
- Bonus: Advanced scenarios

Introduction

Dependencies?

- Any object can be a dependency
- A dependency of X is an object used by X to accomplish its task

Snow Cone

Granizado

Granité

Slushie

冰沙

Granita

Introduction

Dependencies?

- Snow Cone Machine.
- Dependencies: Ice Machine + Syrup Dispenser

Introduction

Snow Cone Machine

- Dependencies: Ice Machine + Syrup Dispenser

```
public class SnowConeMachine {
    private IceMachine iceMachine;
    private SyrupDispenser syrupDispenser;

    public SnowConeMachine() {
        iceMachine = new IceMachine();
        syrupDispenser = new LemonSyrupDispenser();
    }

    public SnowCone makeSnowCone() {
        SnowCone snowCone = new SnowCone();
        snowCone.addIce(iceMachine.makeIce());
        snowCone.addSyrup(syrupDispenser.dispenseSyrup());
        return snowCone;
    }
}
```

Introduction

Disadvantages

- The class is **highly coupled** to its dependencies
- It depends on its dependencies implementation details
- Breaks single responsibility principle
- **Non reusable** code
- **Non testable** code

```
public class SnowConeMachine {
    private IceMachine iceMachine;
    private SyrupDispenser syrupDispenser;

    public SnowConeMachine() {
        iceMachine = new IceMachine();
        syrupDispenser = new LemonSyrupDispenser();
    }

    public SnowCone makeSnowCone() {
        SnowCone snowCone = new SnowCone();
        snowCone.addIce(iceMachine.makeIce());
        snowCone.addSyrup(syrupDispenser.dispenseSyrup());
        return snowCone;
    }
}
```

Dependency Injection

What is Dependency Injection (DI)?

- Software design pattern
- Implements **inversion of control** to resolve dependencies
- **Dependencies are passed** to the object, it does not build or find them
- Separates responsibilities of use and construction
- An object is **independent** of its dependencies **implementation** details

Dependency Injection

Snow Cone Machine

- Dependencies: Ice Machine + Syrup Dispenser

```
public class SnowConeMachine {
    private IceMachine iceMachine;
    private SyrupDispenser syrupDispenser;

    public SnowConeMachine(IceMachine iceMachine, SyrupDispenser syrupDispenser) {
        this.iceMachine = iceMachine;
        this.syrupDispenser = syrupDispenser;
    }

    public SnowCone makeSnowCone() {
        SnowCone snowCone = new SnowCone();
        snowCone.addIce(iceMachine.makeIce());
        snowCone.addSyrup(syrupDispenser.dispenseSyrup());
        return snowCone;
    }
}
```

Dependency Injection

Advantages

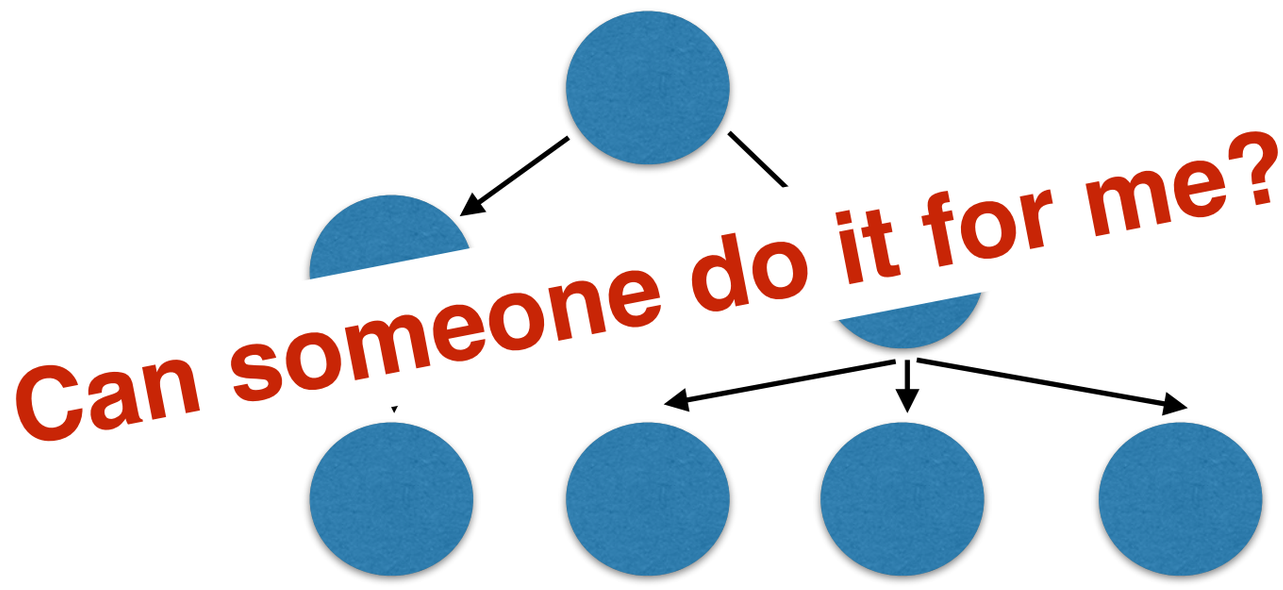
- **Decreases coupling** between the object and its dependencies
- It is **configurable**. It does not need to know about a concrete implementation
- **Reusable** code
- **Testable** code

```
public class SnowConeMachine {
    private IceMachine iceMachine;
    private SyrupDispenser syrupDispenser;

    public SnowConeMachine(IceMachine iceMachine, SyrupDispenser syrupDispenser) {
        this.iceMachine = iceMachine;
        this.syrupDispenser = syrupDispenser;
    }

    public SnowCone makeSnowCone() {
        SnowCone snowCone = new SnowCone();
        snowCone.addIce(iceMachine.makeIce());
        snowCone.addSyrup(syrupDispenser.dispenseSyrup());
        return snowCone;
    }
}
```

Libraries

But…

- Is everything so easy and perfect?
- What about something like:



YES

Libraries

Dependency Injection Libraries

- They **create** and **inject** dependencies for us
- Just **specify** what are the **dependencies** and **how to build** them
- Two main libraries for Android:
  - Roboguice
  - Dagger

Libraries

Roboguice

- Android oriented
- It uses Guice, a Google DI library
- Does most of work at **runtime**
- Really **simple** and **powerful** solution
- Uses **reflection** heavily, being quite **slow**
- Uses Guava, with thousand of methods, contributing to reaching the **dex limit** and **big APKs**

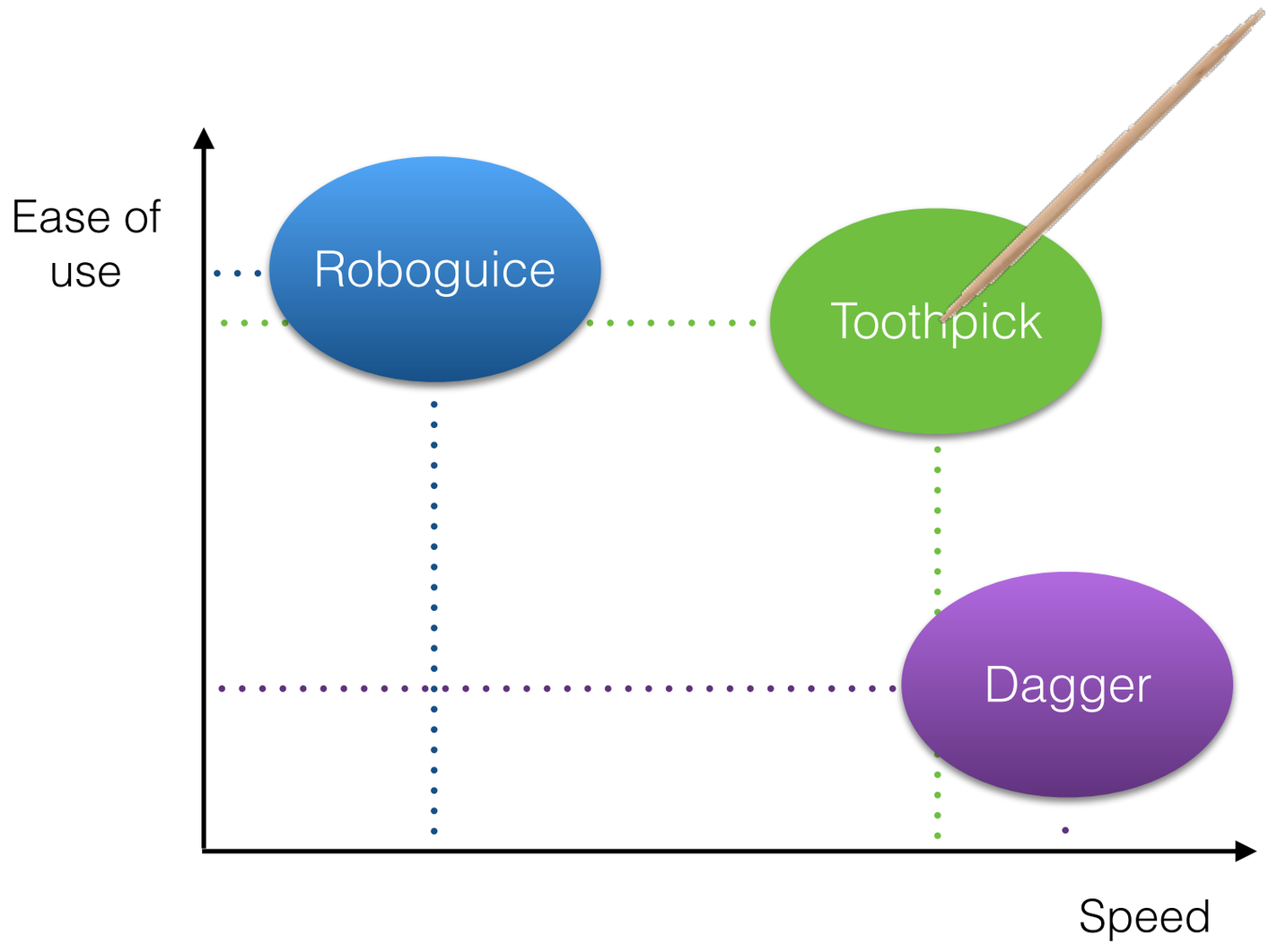
Libraries

Dagger

- Java and Android oriented
- Does most of work at **compile time**
- Reflection replaced by **annotation processing**
- Great **performance**
- **Harder to use** and require **boiler plate** code
- Dependency Injection graph is completely **static**, defined at compile time

Libraries

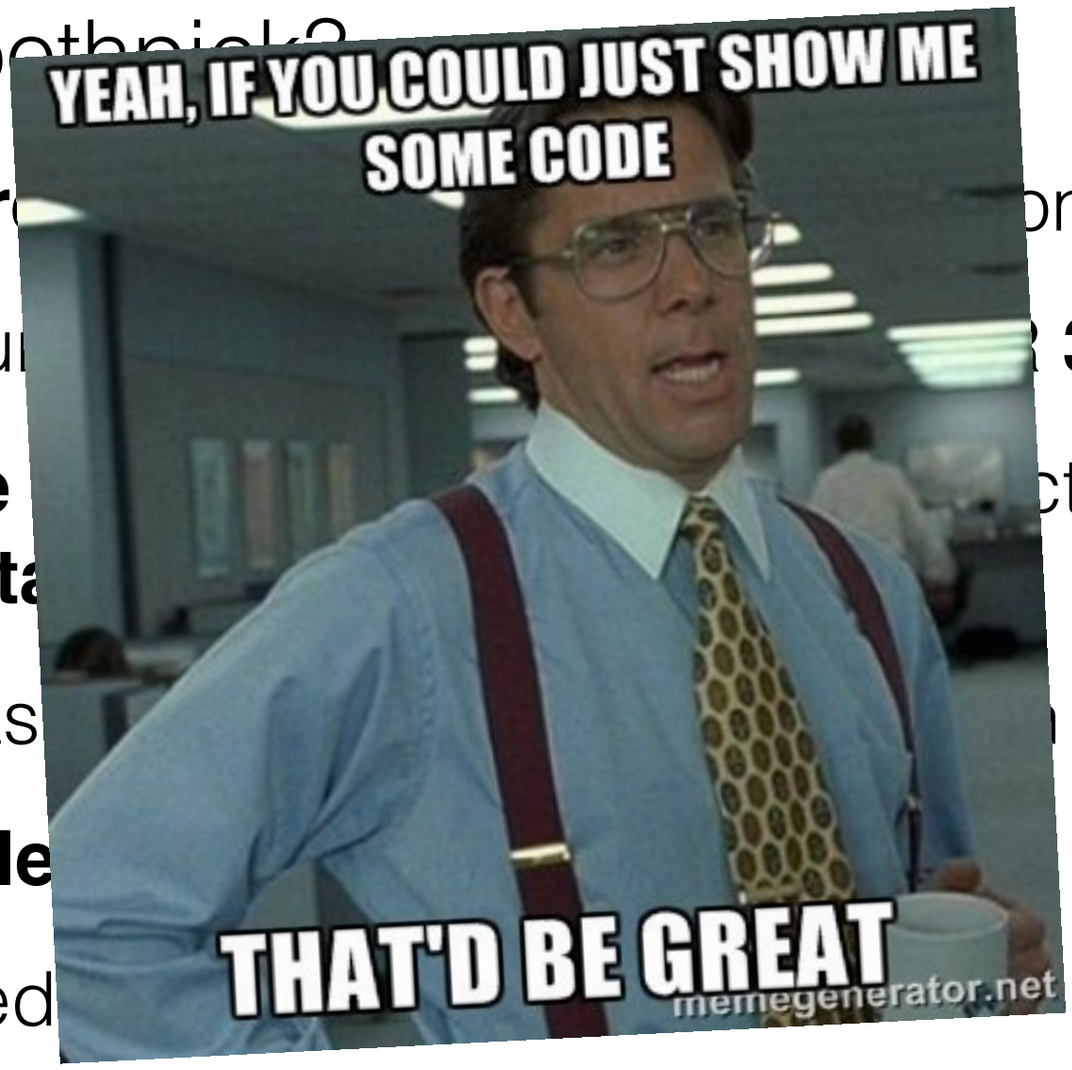
High level overview



Toothpick

What is Toothpick?

- **Scope tr** on library
- Full featu **330**
- **Runtime** ction. Replaced by **annota**
- Almost as some cases
- As **simple**
- Advanced



Toothpick

Snow Cone Machine

- Dependencies: Ice Machine + Syrup Dispenser

Specify Dependencies

```
public class SnowConeMachine {
    @Inject IceMachine iceMachine;
    @Inject SyrupDispenser syrupDispenser;

    public SnowConeMachine() {
        Scope scope = Toothpick.openScope("TP demo");
        Toothpick.inject(this, scope);
    }

    public SnowCone makeSnowCone() {
        SnowCone snowCone = new SnowCone();
        snowCone.addIce(iceMachine.makeIce());
        snowCone.addSyrup(syrupDispenser.dispenseSyrup());
        return snowCone;
    }
}
```

Inject all dependencies and transitive dependencies

Toothpick

Snow Cone Machine

- Dependencies: Ice Machine + Syrup Dispenser

```
public abstract class SyrupDispenser {
    @Inject Pump pump;

    public Syrup dispenseSyrup() {...}
}
```

Transitive Dependencies

Annotated Constructor

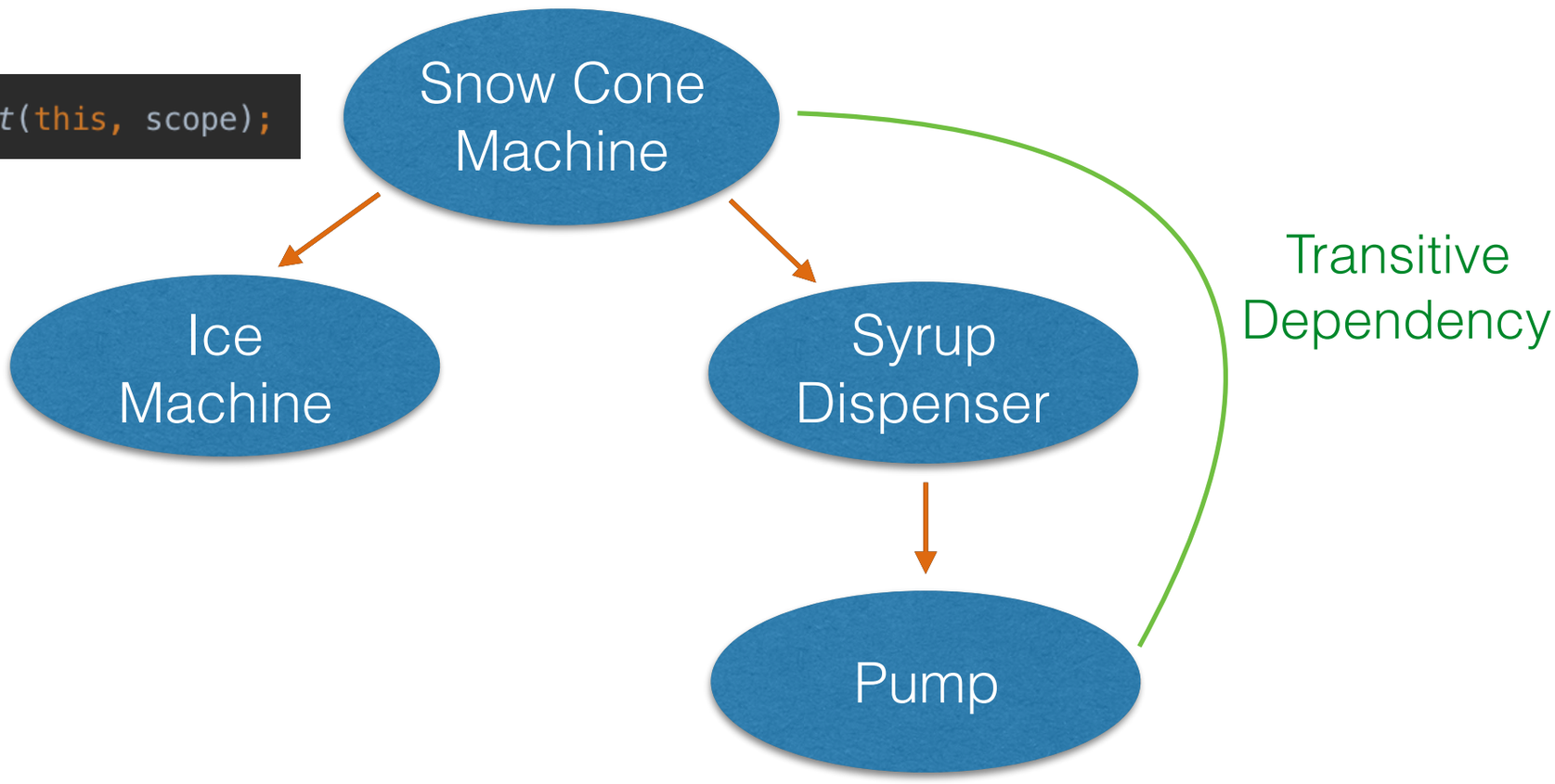
```
public class IceMachine {

    @Inject
    public IceMachine() {
    }

    public Ice makeIce() {...}
}
```

Toothpick

Snow Cone Machine

- Transitive dependencies

```
Toothpick.inject(this, scope);
```



Toothpick Example

Requirements

- Different **flavor** Snow Cone **Machines**: Lemon and Apple
- Only **one Ice Machine**, it is shared
- It should be an easy and fast solution:

Toothpick to the rescue!!

Toothpick Example

2 Snow Cone Machines: Lemon and Apple

- **Scopes** are used to define overrides: **bindings**
- Bindings are attached to the Scopes through **modules**

```
public class LemonConeMachineModule extends Module {

    public LemonConeMachineModule() {
        bind(SyrupDispenser.class).to(LemonSyrupDispenser.class);
    }
}
```

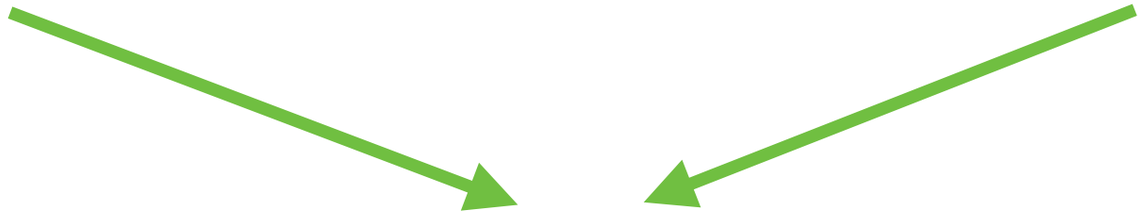
```
public class AppleConeMachineModule extends Module {

    public AppleConeMachineModule() {
        bind(SyrupDispenser.class).to(AppleSyrupDispenser.class);
    }
}
```

Toothpick Example

2 Snow Cone Machines: Lemon and Apple

```java
public class LemonConeMachineModule extends Module {

    public LemonConeMachineModule() {
        bind(SyrupDispenser.class).to(LemonSyrupDispenser.class);
    }
}
```

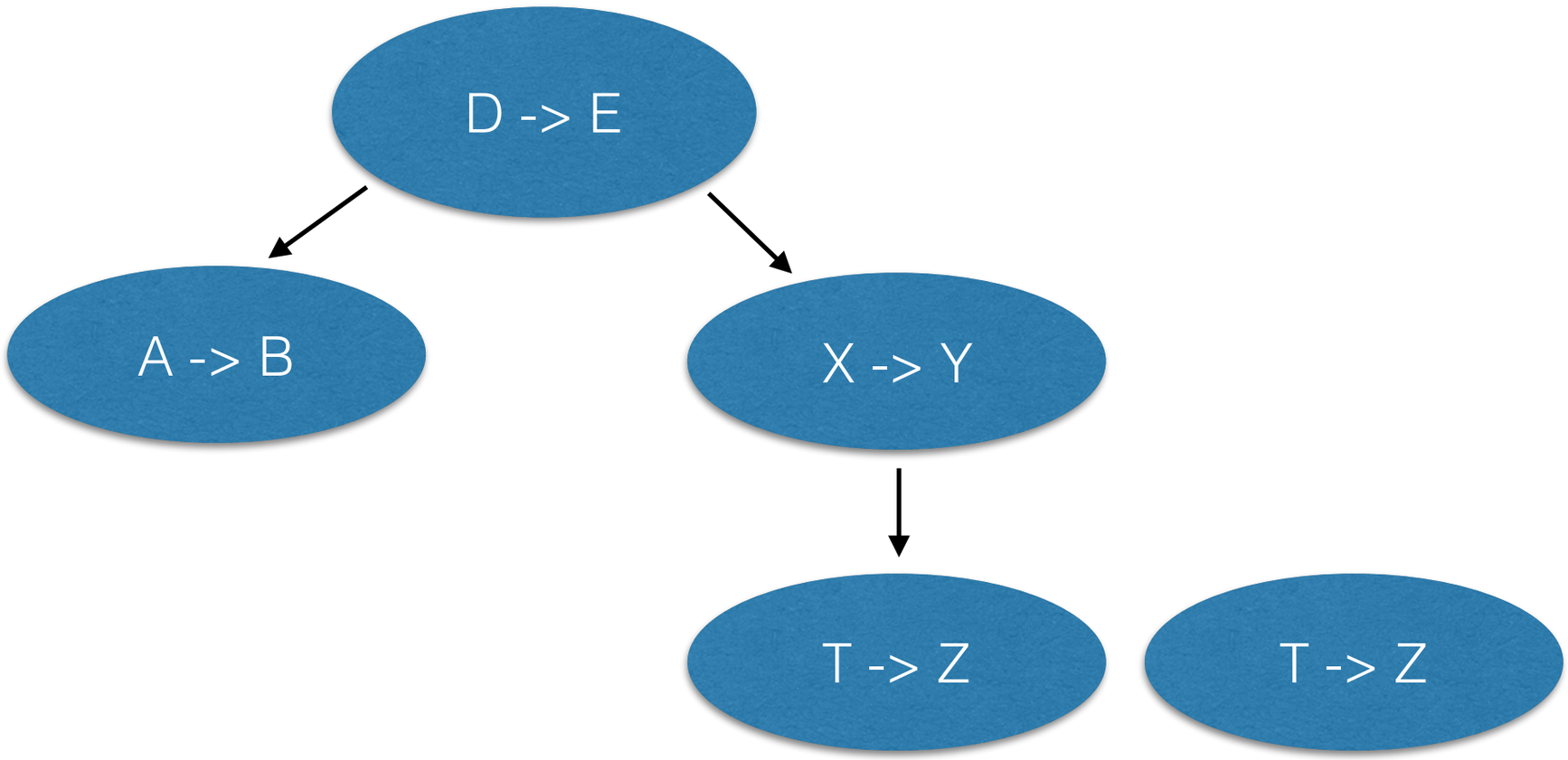
```java
public class AppleConeMachineModule extends Module {

    public AppleConeMachineModule() {
        bind(SyrupDispenser.class).to(AppleSyrupDispenser.class);
    }
}
```



```java
Scope lemonScope = Toothpick.openScope("lemon");
lemonScope.installModules(new LemonConeMachineModule());
Toothpick.inject(this, lemonScope);

Scope appleScope = Toothpick.openScope("apple");
lemonScope.installModules(new AppleConeMachineModule());
Toothpick.inject(this, appleScope);
```

Toothpick Example

Scope tree based



Toothpick Example

Only one Ice Machine, it is shared

- The Ice Machine is a **Singleton**
- Use @Singleton annotation

```
@Singleton
public class IceMachine {

    @Inject
    public IceMachine() {
    }

    public Ice makeIce() {...}
}
```

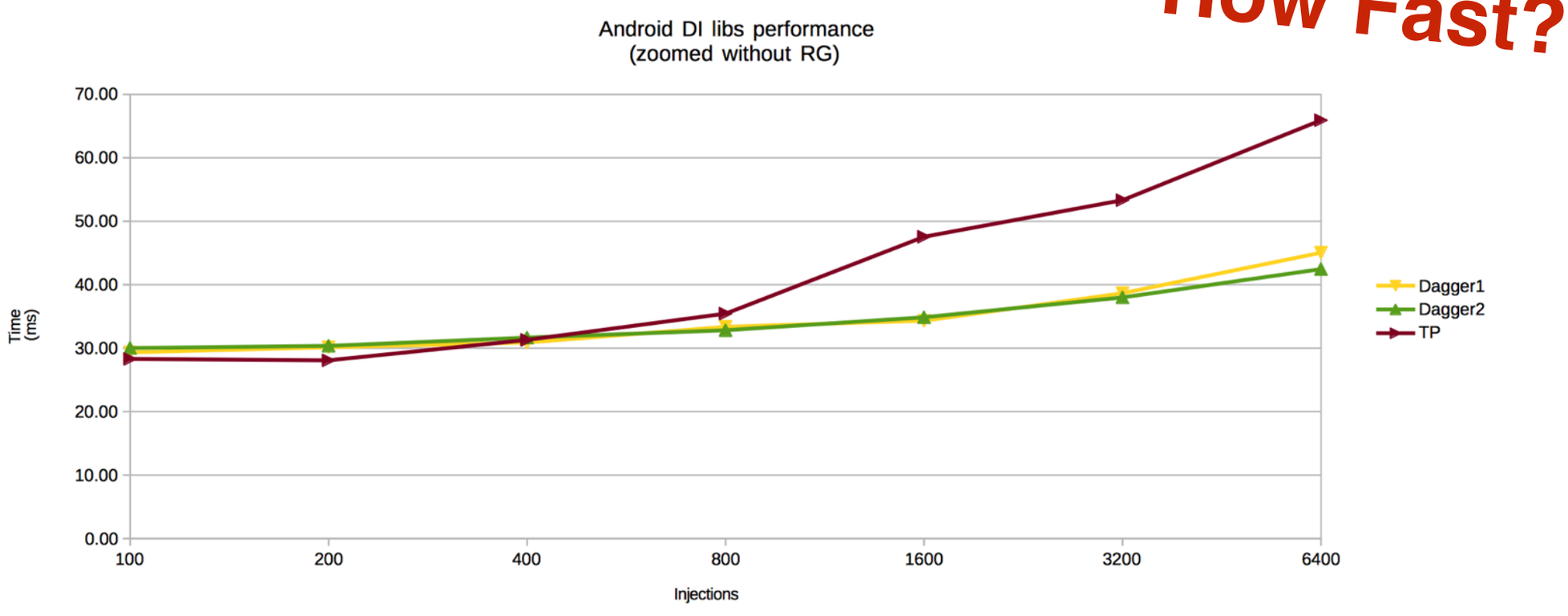
Bonus: Advanced scenarios

- MVP, keeping the **presenter** state
  - Nowadays -> Retained fragments, Loaders, …
  - Toothpick -> Use a **scope** to **retain** the instance of the **Presenter**
- **User flow** data
  - Nowadays -> Send data over and over through Intents, Database, …
  - Toothpick -> Use a common **scope** for the **whole flow** to keep flow related data

Conclusion

- Toothpick is **scope tree** based DI framework
- Advanced **testing support**
- Nice scenarios: **MVP**, **Flow Scope**, …
- **Fast** and **simple**

How Fast?



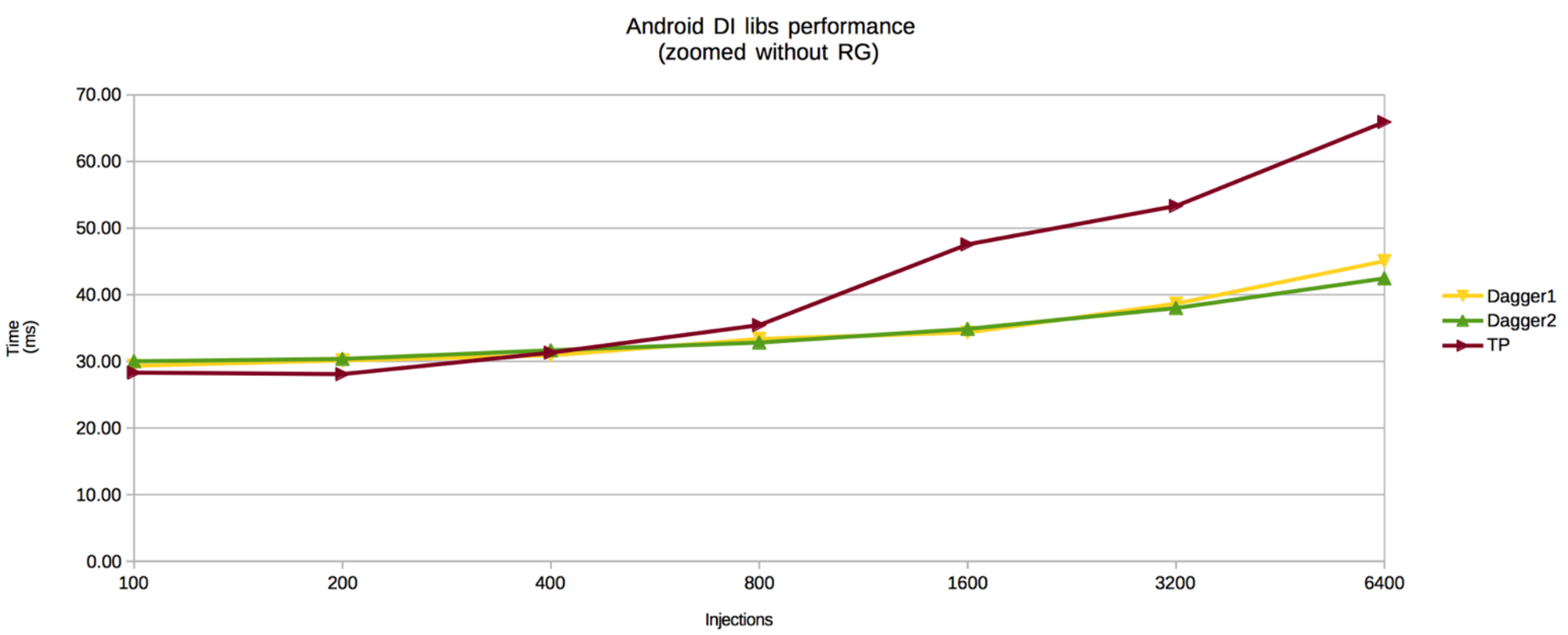
Conclusion

1000 injections

- Dagger 1: 33 ms
- Dagger 2: 31 ms
- Toothpick: 35 ms

6400 injections

- Dagger 1: 45 ms
- Dagger 2: 42 ms
- Toothpick: 66 ms



The Android Team is Hiring

https://jobs.groupon.com/#/categories/engineering