SOLIDify your code

...

Arnaud Bellizzi

About me

@Oodrive (since 2013)

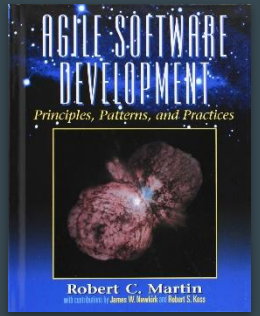
Team Backup & Archive

Architect

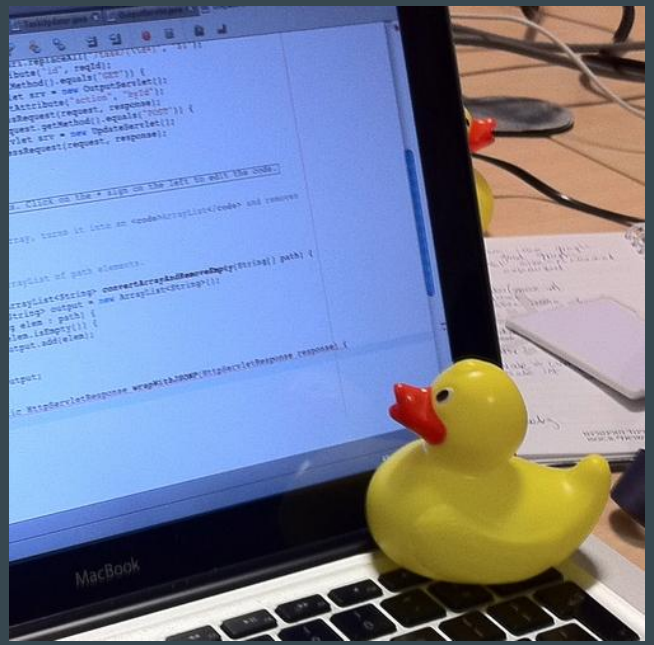
SOLID

- Robert C. Martin - Agile Software Development (2003)

- 5 Principles - S. O. L. I. D.
- **Object Oriented** design
- About Software **Maintainability**



Rubber Duck Debugging



SOLID Principles

Single Responsibility

"One class - One reason to change"

SOLID Principles

Single Responsibility — "One class - One reason to change"

Open/Closed — "Open for extension, Closed for modification"

SOLID Principles

Single Responsibility — “One class - One reason to change”

Open/Closed — “Open for extension, Closed for modification”

Liskov Substitution — “Subtyping should not break anything”

SOLID Principles

Single Responsibility	"One class - One reason to change"
Open/Closed	"Open for extension, Closed for modification"
Liskov Substitution	"Subtyping should not break anything"
Interface Segregation	"Use small cohesive interfaces"

SOLID Principles

Single Responsibility	"One class - One reason to change"
Open/Closed	"Open for extension, Closed for modification"
Liskov Substitution	"Subtyping should not break anything"
Interface Segregation	"Use small cohesive interfaces"
Dependency Inversion	

Wait, I lied

Wait, I lied

~~SOLI~~ Dify your code

Dependency Inversion Principle

1. *High-level modules should not depend on low-level modules.*

Dependency Inversion Principle

1. *High-level modules should not depend on low-level modules.*
   Both should depend on abstractions.

Dependency Inversion Principle

1. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*

2. *Abstractions should not depend on details.*

Dependency Inversion Principle

1. *High-level modules should not depend on low-level modules.*
   Both should depend on abstractions.

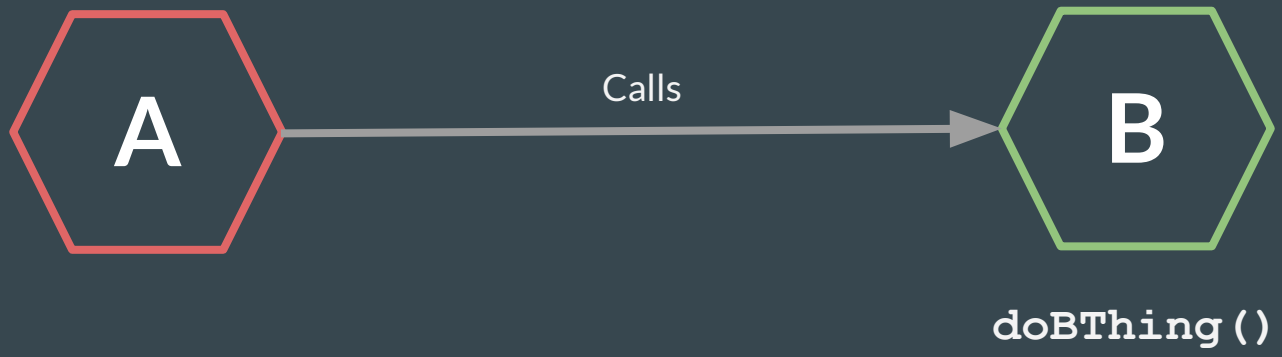
2. *Abstractions should not depend on details.*
   Details should depend on abstractions.

Dependencies

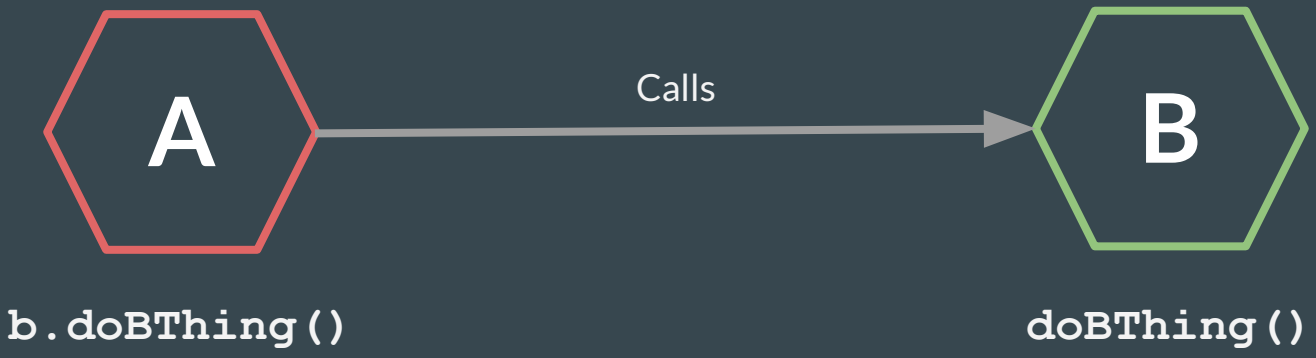
Dependencies


```
doBThing()
```

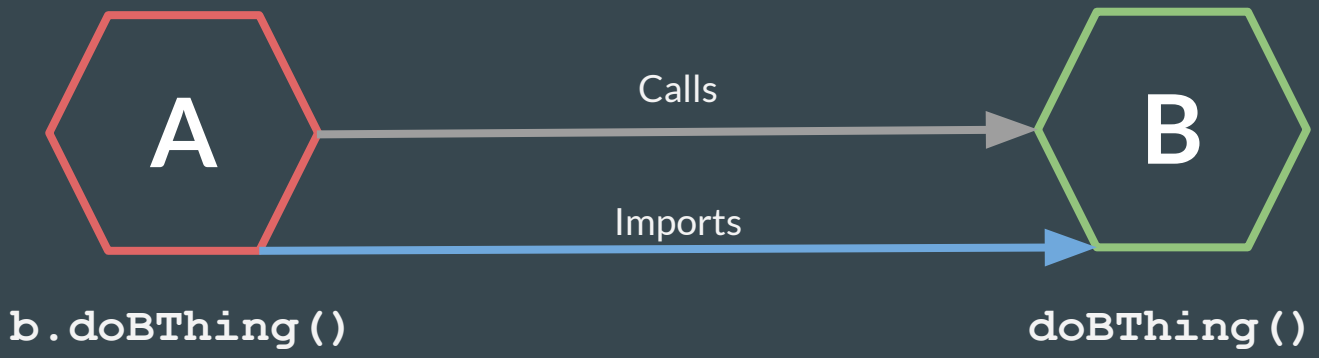
Dependencies



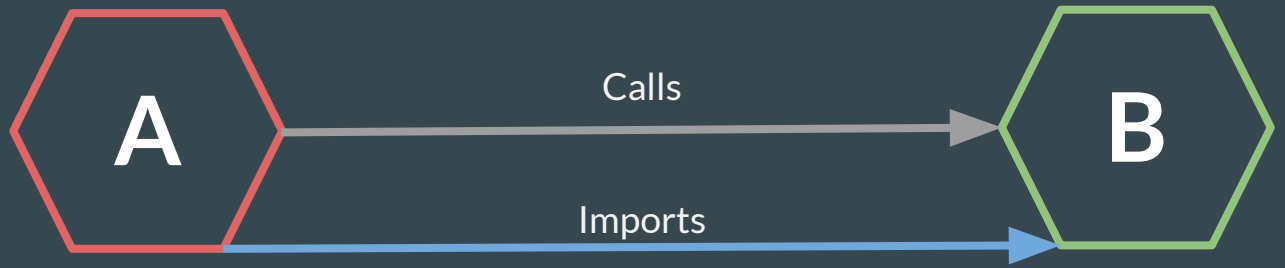
Dependencies



Dependencies



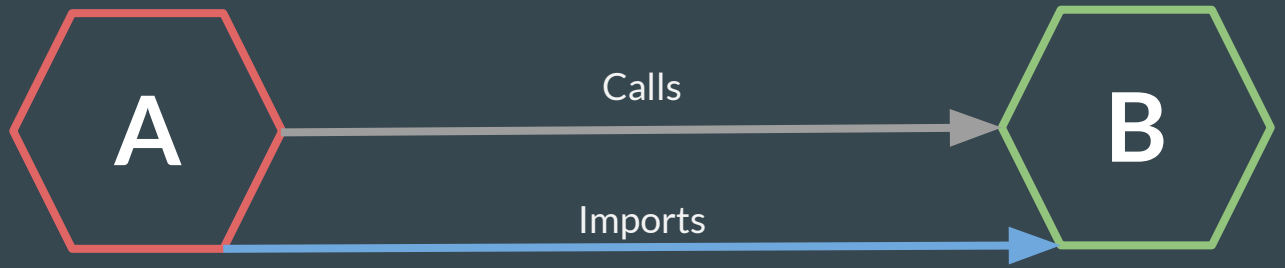
Dependencies + Change




```
b.doBThing()
```

~~`doBThing()`~~
`doBetter()`

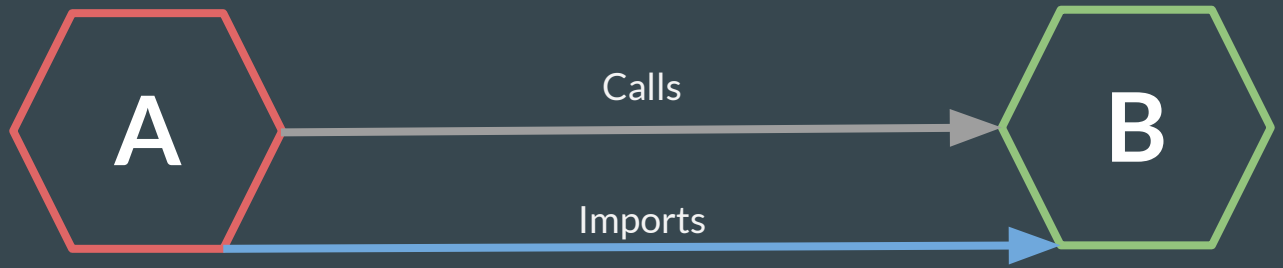
Dependencies + Change



~~`b.doBThing()`~~
`b.doBetter()`

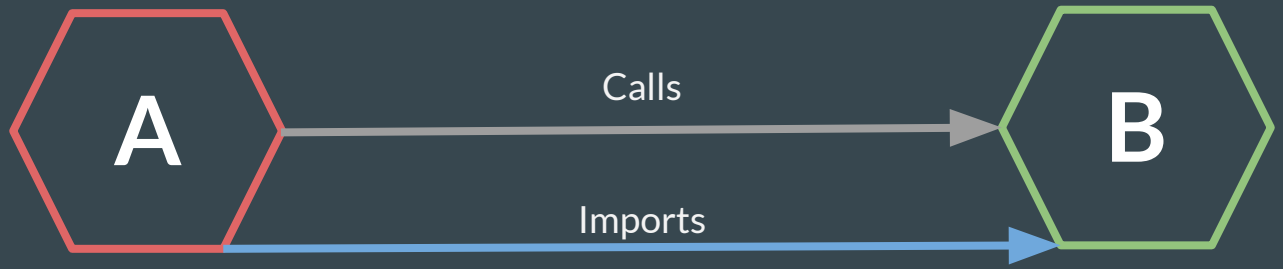
~~`doBThing()`~~
`doBetter()`

Dependencies + Change



Changes in B imply changes in A

Dependencies + Change



Changes in **B** imply changes in **A**

External API

Business

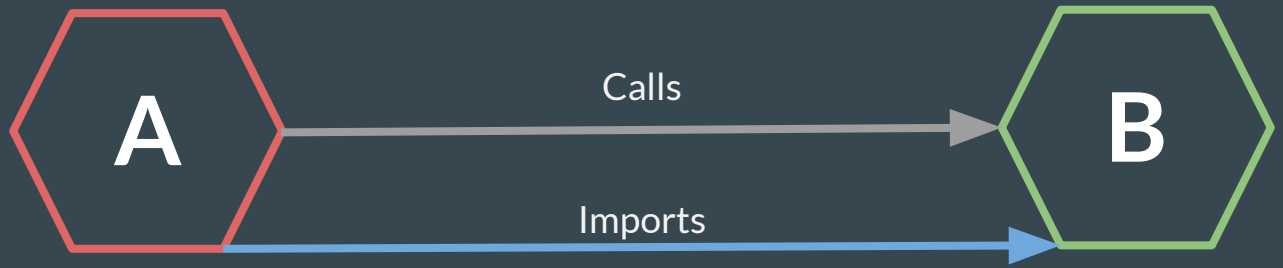
Dependencies + Change

Changes in **B** imply changes in **A**

External API

Business

Dependencies + Change



Changes in **B** imply changes in **A**

Business **Persistence**

Dependencies + Change

Changes in **B** imply changes in **A**

Business

Persistence

Dependencies

There is a problem when . . .

Dependencies

There is a problem when . . .

- **B** changes often

Dependencies

There is a problem when . . .

- ~~B changes often~~
- We want to Reuse A

Dependency Inversion Principle

1. ***High-level*** *modules should not depend on* ***low-level*** *modules.*
   Both should depend on ***abstractions****.*

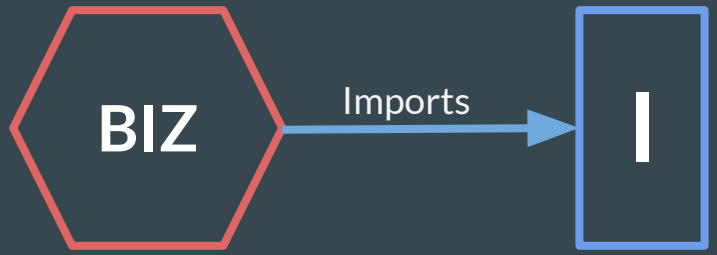
2. ***Abstractions*** *should not depend on* ***details****.*
   Details *should depend on* ***abstractions****.*

Dependencies Inverted

Dependencies Inverted

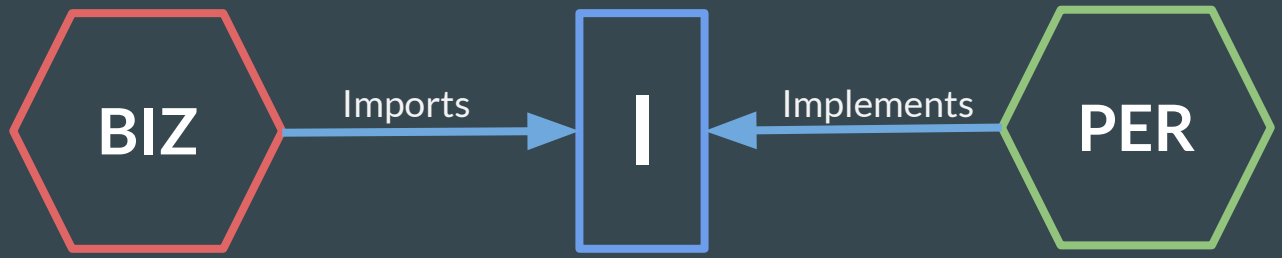

```
store(...)
```

Dependencies Inverted



`i.store(...)` `store(...)`

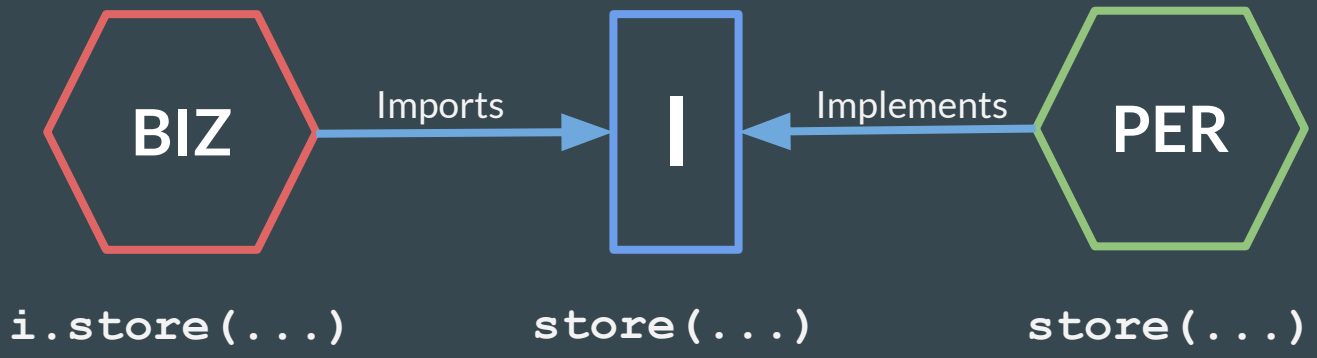
Dependencies Inverted



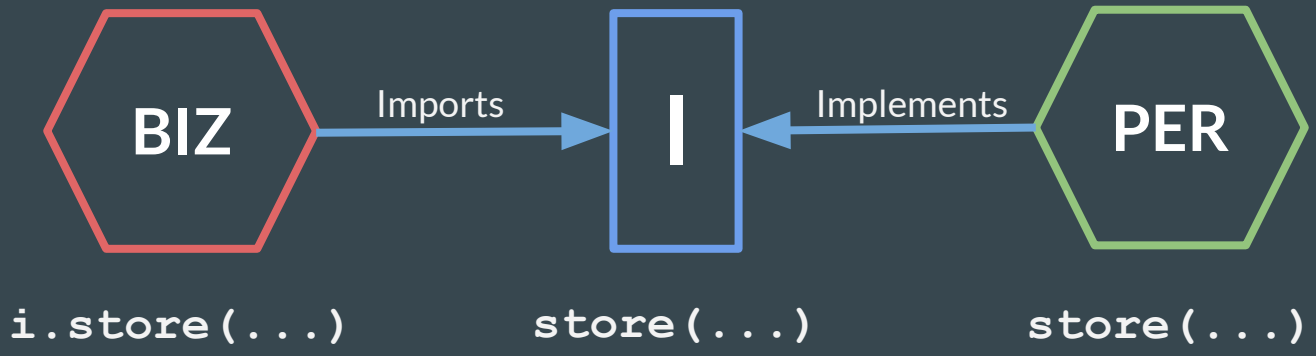

```
i.store(...)
```

```
store(...)
```

Dependencies Inverted

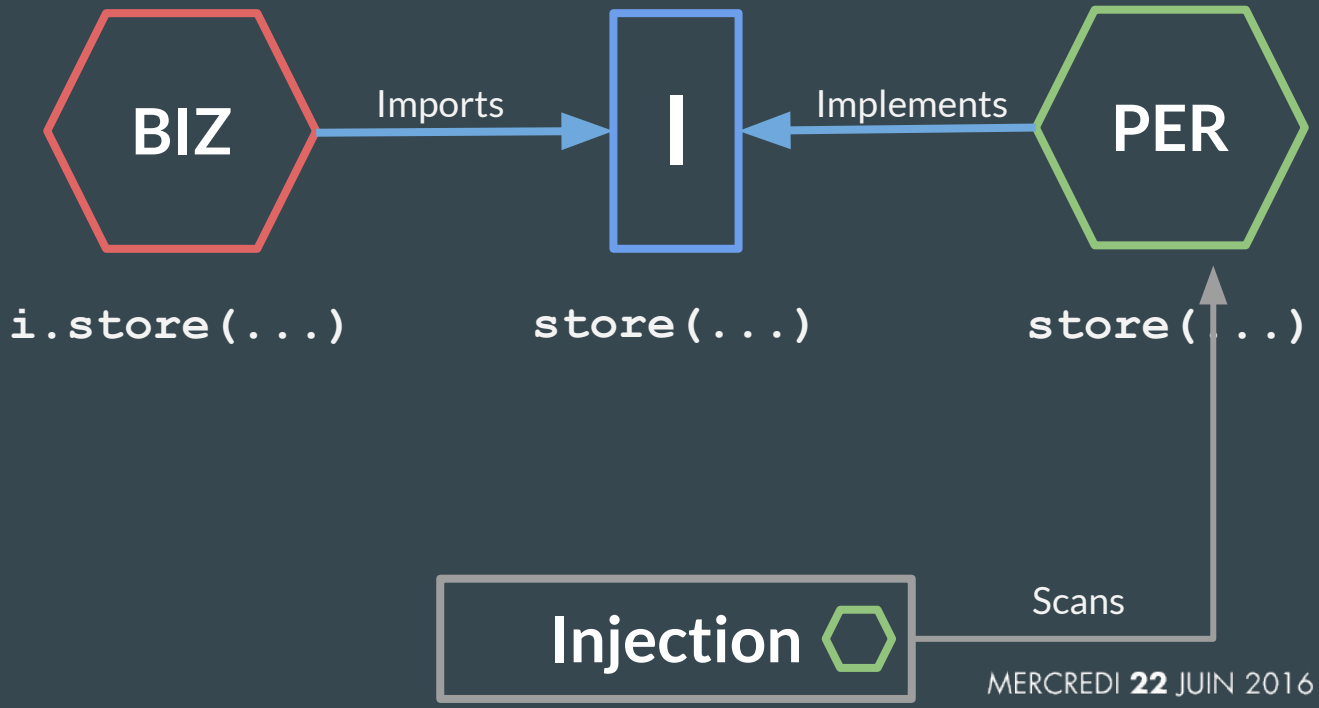


Dependencies Inverted

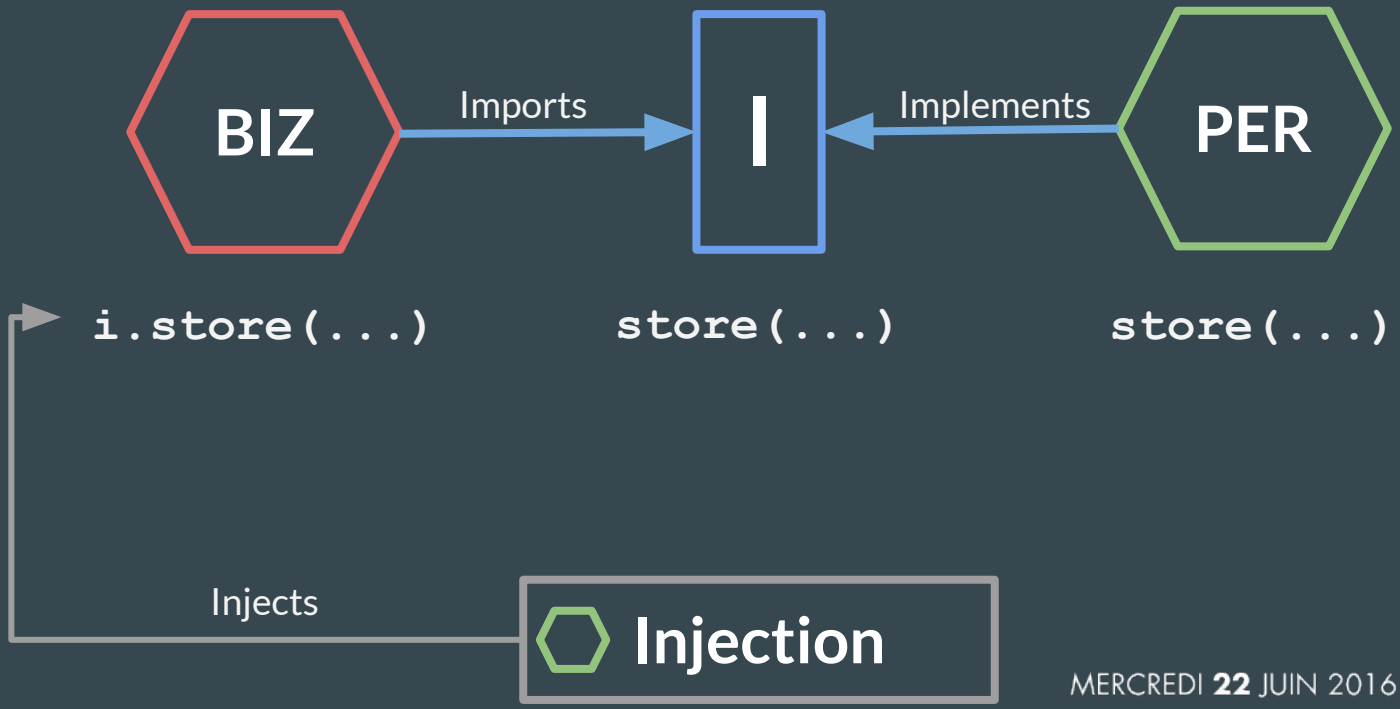


Injection

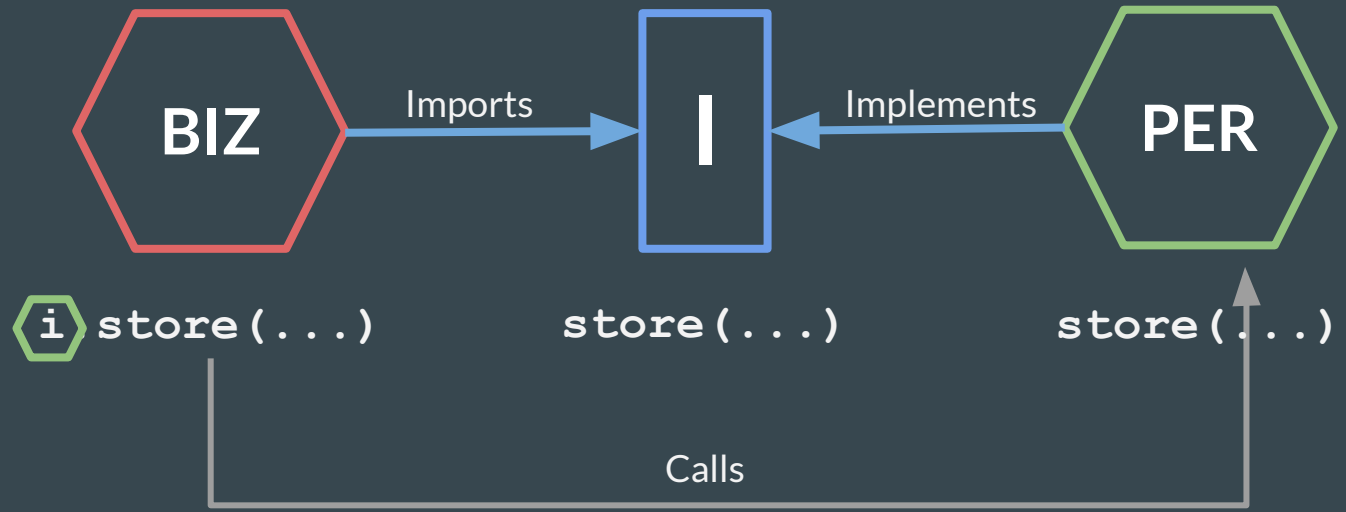
Dependencies Inverted



Dependencies Inverted

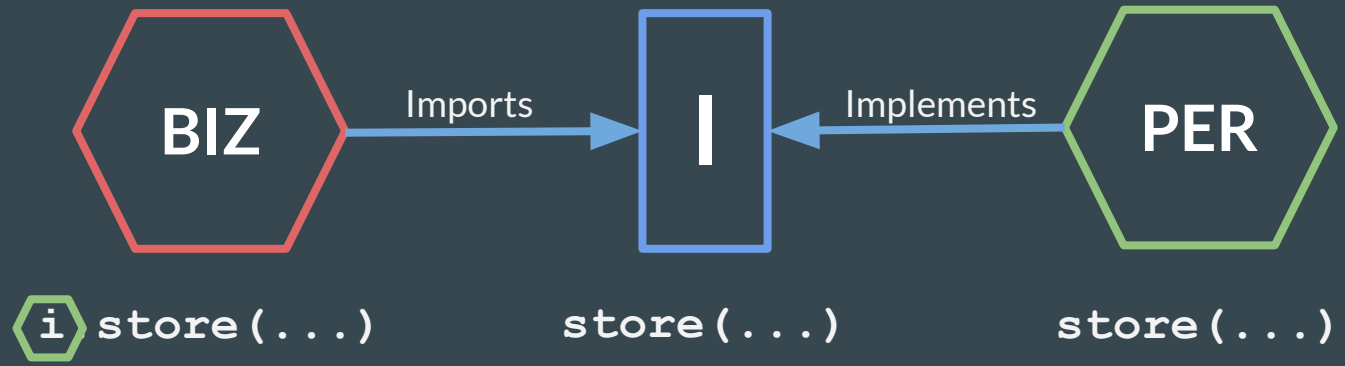


Dependencies Inverted



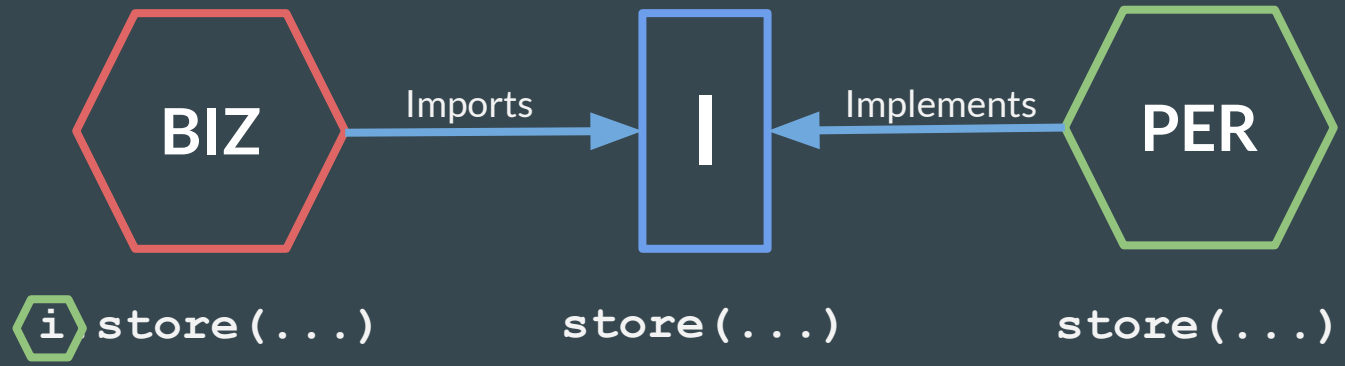
Injection

Dependencies Inverted + Change



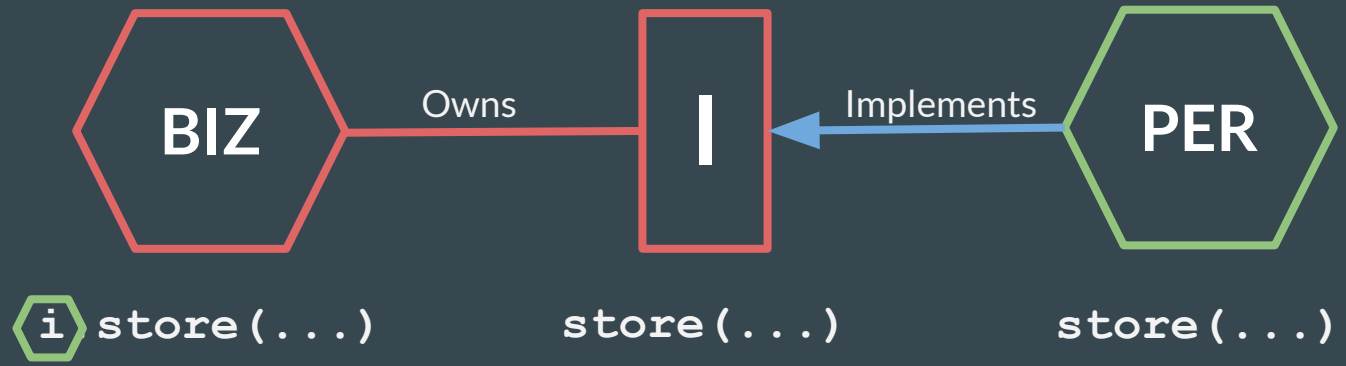
Changes in **PER** are restricted by **Contract**

Dependencies Inverted + Change



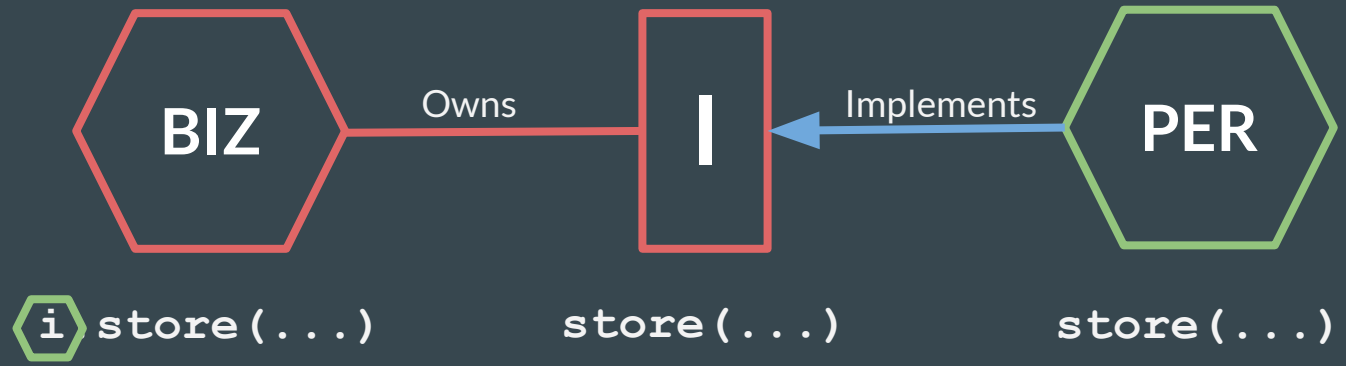
Changes in Contract affect BIZ & PER

Dependencies Inverted + Change



BIZ owns the Contract

Dependencies Inverted + Change



Changes in PER are unseen by BIZ

Dependencies Inverted + Change

Changes in **PER** are unseen by **BIZ**

Costs

- Requires more code

Costs

- Requires more code
- Performance overhead

Costs

- Requires more code
- Performance overhead
- Where is my implementation ?

Gains

- Protect valuable code from unnecessary change

Gains

- Protect valuable code from unnecessary change

- Increased readability of valuable code

Gains

- Protect valuable code from unnecessary change

- Increased readability of valuable code

Your **domain** , your rules

Code inspection

Evaluating our modules

Code inspection

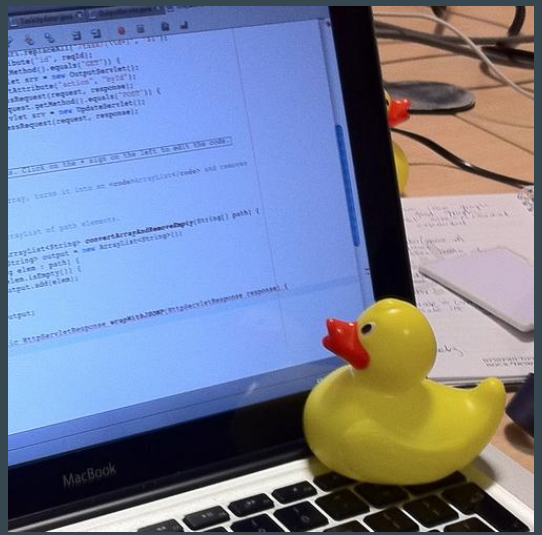
Evaluating our modules

Explain their responsibilities

Code inspection

Evaluating our modules

Explain their responsibilities



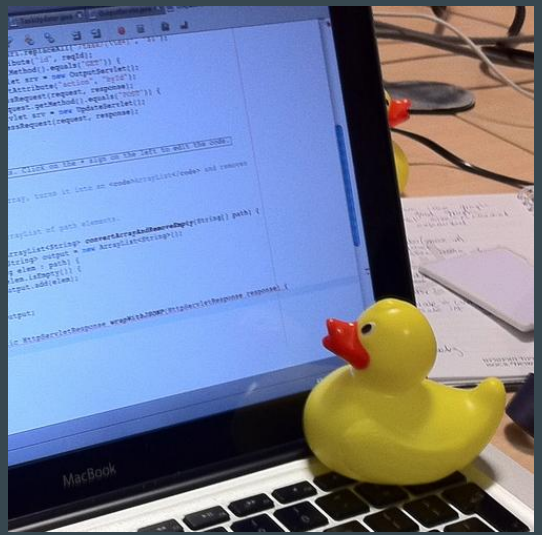
Code inspection

Evaluating our modules

Explain their responsibilities

Business knows details about

- Persistence
- Messaging



Invert all the things

But Agile wins

- Don’t code for tomorrow
- Don’t plan for reuse
- Refactor towards perfection

But Agile wins

- Don’t code for tomorrow
- Don’t plan for reuse
- Refactor towards perfection

So where do we start ?

Start on new code ?

+ Manageable refactors

Start on new code ?

\+ Manageable refactors (as in no refactor)

Start on new code ?

+ Manageable refactors (as in no refactor)

- Possibly useless

Start on new code ?

+ Manageable refactors (as in no refactor)

- ~~Possibly useless~~
- Probably useless

Start on required changes

+ Probably valuable

Start on required changes

+ Probably valuable (at least it changed once)

Start on required changes

+ Probably valuable (at least it changed once)

- Keep refactors manageable

Start on required changes

+ Probably valuable (at least it changed once)

- Keep refactors manageable

  ↳ 1 use case at a time

Day to day

- Code Review : Look for details
- Pitch abstractions to outsiders

Thank you