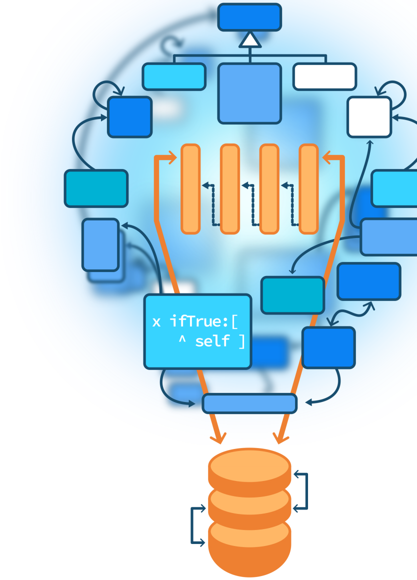
Advanced Object-Oriented Design

About coupling and encapsulation

S.Ducasse, L. Fabresse, G. Polito, and P. Tesone



http://www.pharo.org

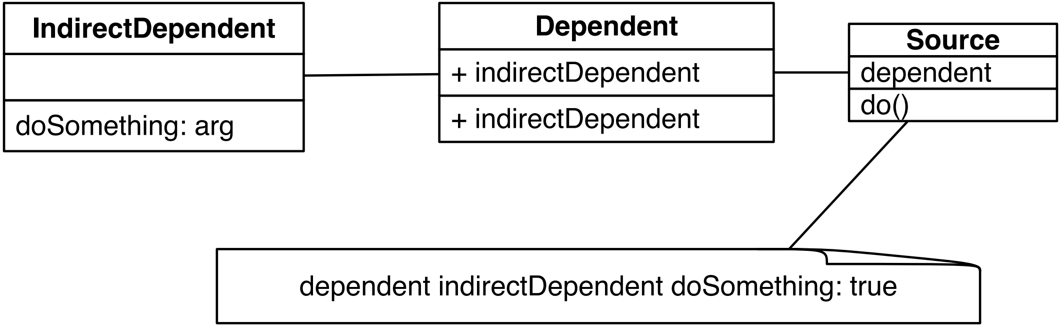
Goal and outline

- Think about coupling
- Present Law of Demeter
- **Move Behavior closer to Data** from Object-Oriented Reengineering Pattern book
- Tradeoffs

Symptoms of costly coupling

- **Reuse:** I cannot reuse this component in another application
- **Substitution:** I cannot easily substitute this part for another one
- **Encapsulation:** When a change happens far away, I get impacted
- **Untestable:** I cannot test this part

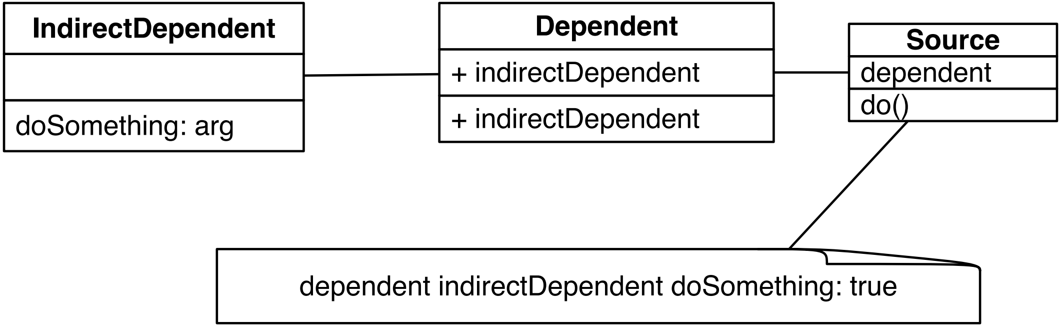
Core of the problem illustrated



- Related to Feature Envy code smell

Changes are natural

- When you change, your dependents should change
- The problem is: **waves of changes when dependents of dependents change**



Waves are evil

- Waves are created by leaks of references of **far/indirect** objects
- Waves are due to **violation of encapsulation**

How to **limit** wave creation?

- Do not leak **far** references!

Law of Demeter

You should **only** send messages to:

- an argument passed to you
- instance variables
- an object you create
- self, super, and your class

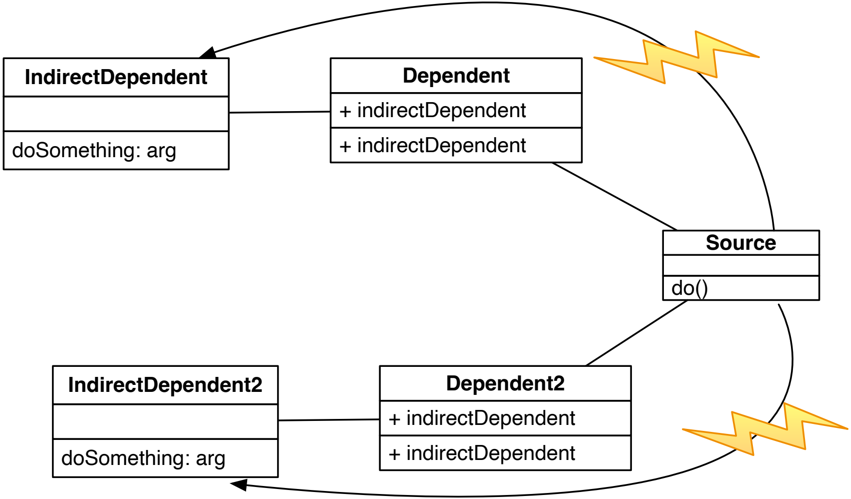
You should **avoid**

- **global** variables
- objects returned from message sends other than self

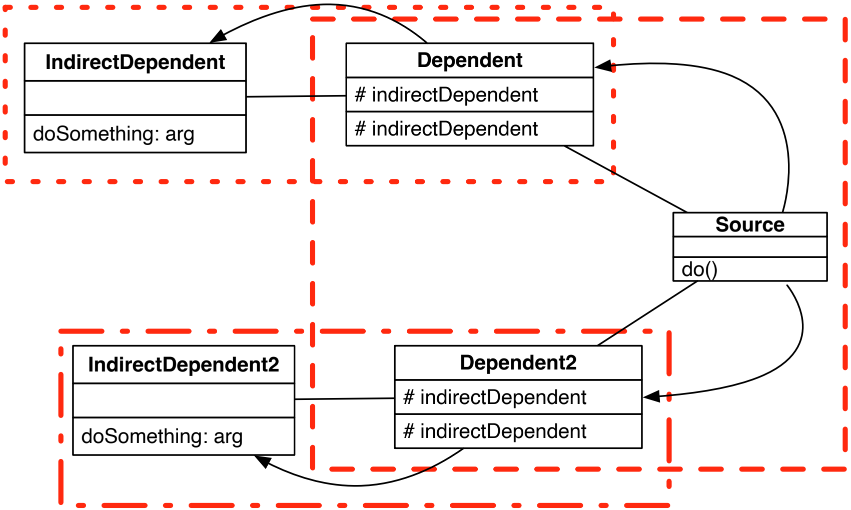
Only talk to your immediate friends

```
someMethod: aParameter
  self foo.
  super someMethod: aParameter.
  self class foo.
  self instVarOne foo.
  instVarOne foo.
  aParameter foo.
  thing := Thing new.
  thing foo
```

Don’t skip your intermediates

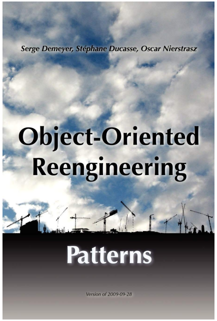


Solution: Respect encapsulation

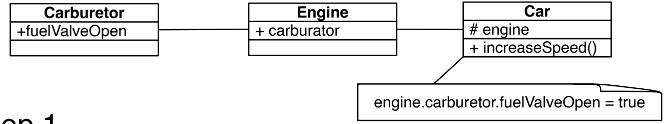


Let us "Move behavior close to data"

- Apply **Move behavior close to data** object-oriented reengineering pattern
- **Intent:** Strengthen encapsulation by moving behavior from indirect clients to the class containing the data it operates on
  - if data and behavior are not close (Feature Envy code smell)
  - then logic is distributed/duplicated in clients!


Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz
Object-Oriented Reengineering
Patterns
Version of 2009-09-28

Move behavior close to data: Transformation



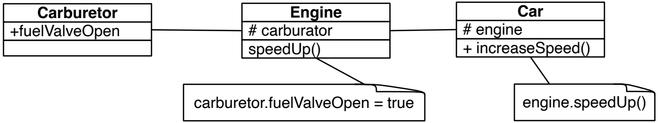
Carburetor
+fuelValveOpen

Engine
+ carburator

Car
# engine
+ increaseSpeed()

engine.carburetor.fuelValveOpen = true

Step 1



Carburetor
+fuelValveOpen

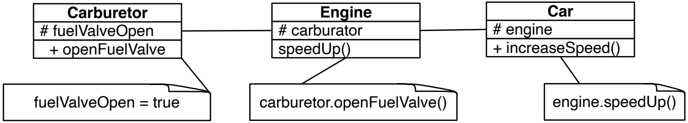
Engine
# carburator
speedUp()

Car
# engine
+ increaseSpeed()

carburetor.fuelValveOpen = true

engine.speedUp()

Step 2



Carburetor
# fuelValveOpen
+ openFuelValve

Engine
# carburator
speedUp()

Car
# engine
+ increaseSpeed()

fuelValveOpen = true

carburetor.openFuelValve()

engine.speedUp()

Real (fixed) example

```
OSWindowMorphicEventHandler >> visitWindowResolutionChangeEvent: anEvent
  "Resolution (dpi) changed. For now just check for a new size."
  "We need to reset the render if the resolution changes."

  morphicWorld worldState worldRenderer window backendWindow renderer destroy.
  morphicWorld worldState worldRenderer window backendWindow renderer validate.
  morphicWorld worldState doFullRepaint.
  morphicWorld worldState worldRenderer window backendWindow renderer
      updateAll.
  morphicWorld worldState worldRenderer checkForNewScreenSize
```

Solution

```
OSWindowMorphicEventHandler >> visitWindowResolutionChangeEvent: anEvent
  morphicWorld worldState updateToNewResolution: anEvent
```

```
WorldState >> updateToNewResolution: originalEvent
  "We need to reset the render if the resolution changes."

  self doFullRepaint.
  self worldRenderer updateToNewResolution.
  self worldRenderer checkForNewScreenSize
```

```
OSSDL2BackendWindow >> updateToNewResolution
  "Force the regeneration of the renderer because we have a new resolution"
  renderer destroy.
  renderer validate.
  renderer updateAll.

NullWorldRenderer >> updateToNewResolution
  lf
```

Analysis

Going from mere navigation to better logic

```
WorldState >> updateToNewResolution: originalEvent
  "We need to reset the render if the resolution changes."

  self doFullRepaint.
  self worldRenderer updateToNewResolution.
  self worldRenderer checkForNewScreenSize
```

LOD is a "heuristic"

- Pay attention! A too strict application of the LOD can lead to **bloated class API**
- Encapsulating collections may produce large interfaces so not applying the LoD may help
- Understand when it is **reasonable to leak**

LOD can produce bloated APIs

Do we create around 50 methods per instance variable holding a collection?

```
Object subclass: #FMMethods
  instVar: 'senders'
  ...
```

```
FMMethods >> do: aBlock
  senders do: aBlock
FMMethods >> collect: aBlock
  ^ senders collect: aBlock
FMMethods >> select: aBlock
  ^ senders select: aBlock
FMMethods >> detect: aBlock
  ^ senders detect: aBlock
FMMethods >> isEmpty
  ^ senders isEmpty
...
```

Conclusion

- Think **about impact** of changes
- Avoid **chaining** messages
- Law of Demeter is a **heuristic**
- **Move behavior close to data** reengineering pattern

Produced as part of the course on http://www.fun-mooc.fr

Advanced Object-Oriented Design and Development with Pharo

A course by

S.Ducasse, L. Fabresse, G. Polito, and P. Tesone

Inria
LearningLab

IMT Nord Europe
École Mines-Télécom
IMT-Université de Lille

2023