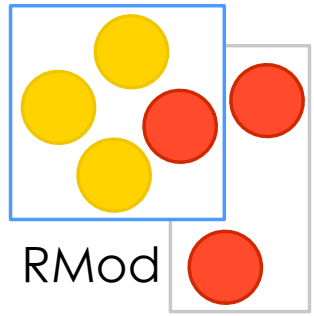
INRIA



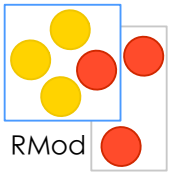
Elements of Design: Law of Demeter

Stéphane Ducasse
stephane.ducasse@inria.fr
http://stephane.ducasse.free.fr/

About Coupling

- What is coupling?
- Why coupled classes are fragile
- Law of Demeter
- Thoughts about accessor use

Coupling (Due to Constantine)



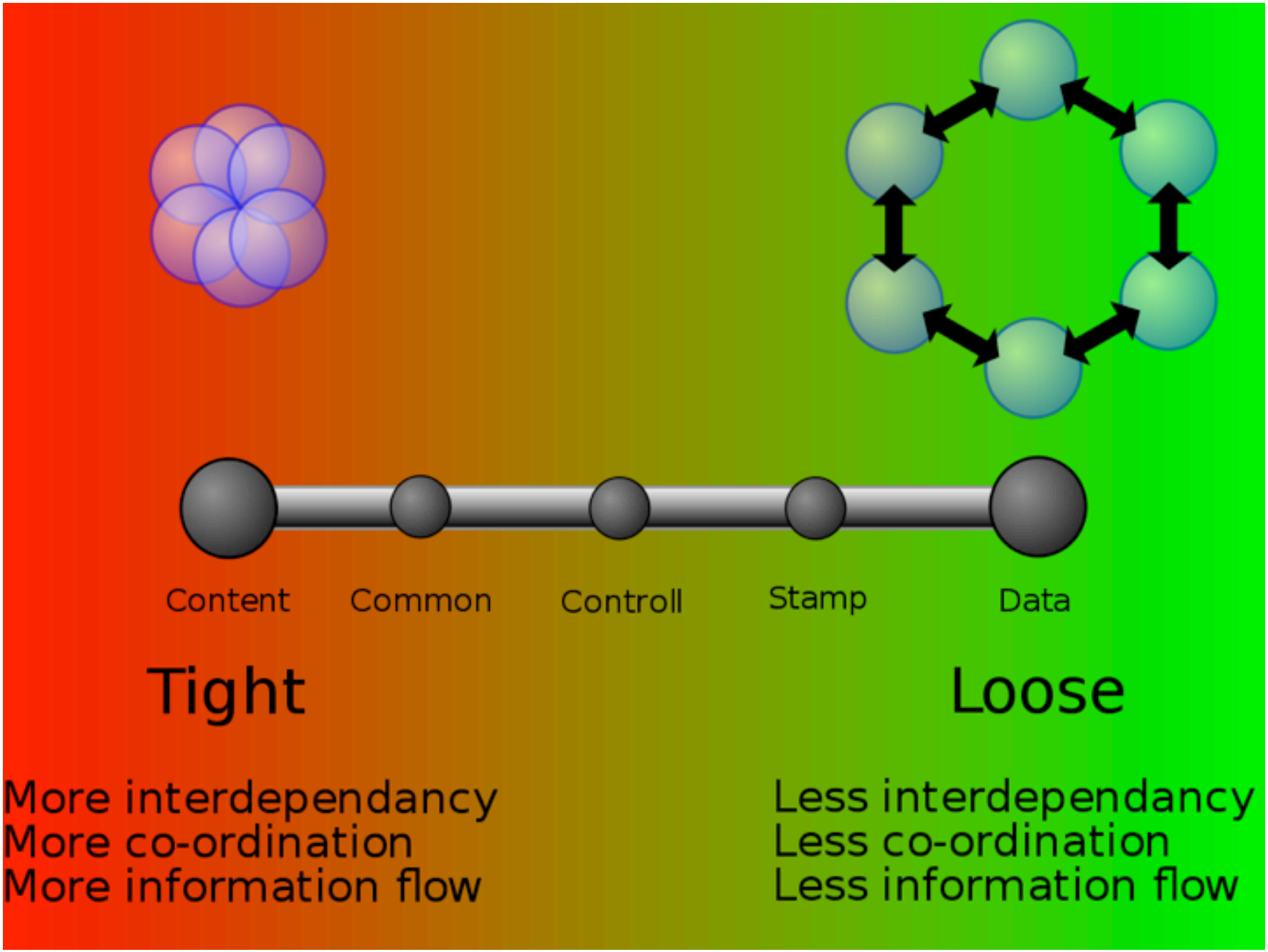
Content coupling (high): one object *a* relies on the internal workings of another object *b*

Problem: changing the way *b* produces or stores data (location, type, timing) will necessitate changing *a*.

Data and Message coupling (low): all communication between objects *a* and *b* relies messaging or passing parameters that are elementary objects

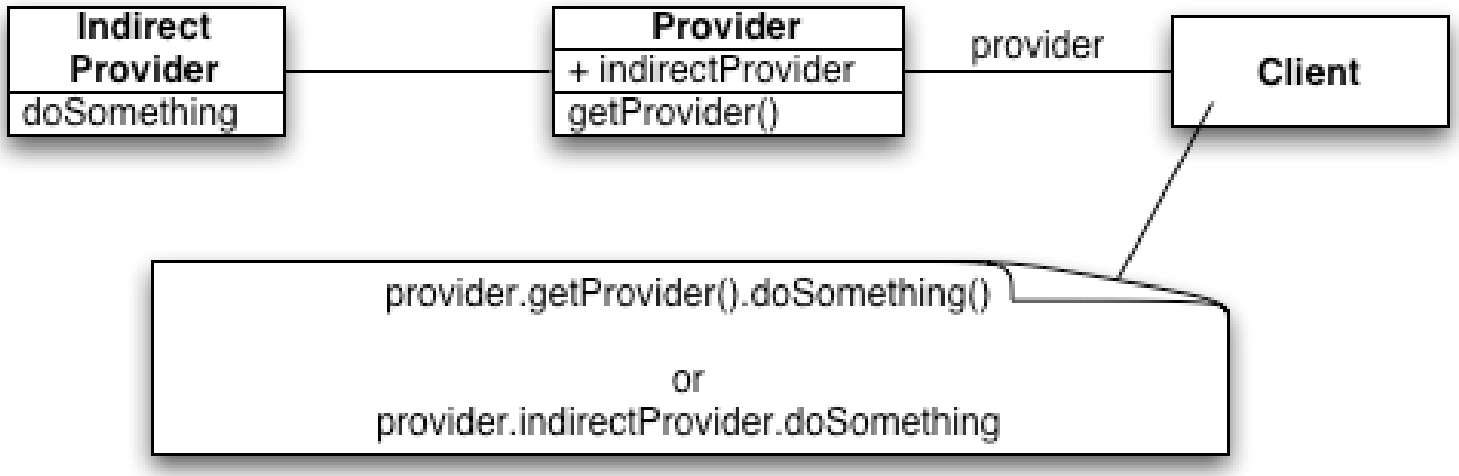
The protocol understood by en elementary object is unlikely to change

➡ Avoid high coupling

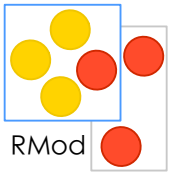


Wikipedia commons

The Core of the Problem



The Law of Demeter



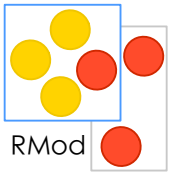
You should send messages only to:

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

Avoid global variables

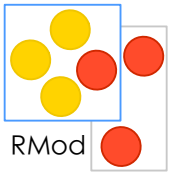
Avoid objects returned from messages sent to others

Correct Messages




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

In other words …



Talk only to your immediate friends

- Friends of friends are suspect

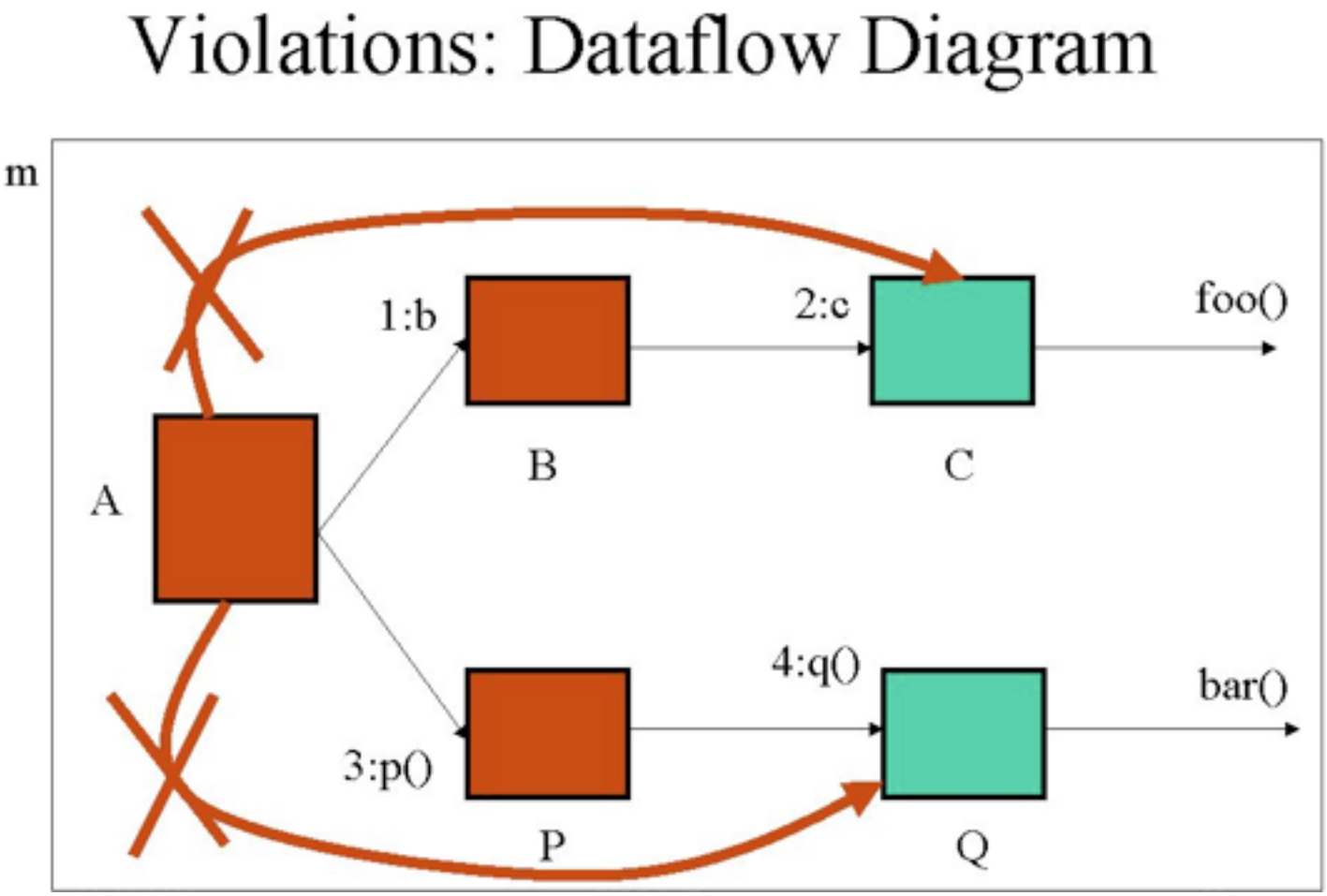
In other words:

- You can play with yourself. this.method()
- You can play with your own toys (but you can't take them apart). field.method(), field.getX()
- You can play with toys that were given to you. arg.method()
- And you can play with toys you've made yourself.
  A a = new A(); a.method()

Halt!

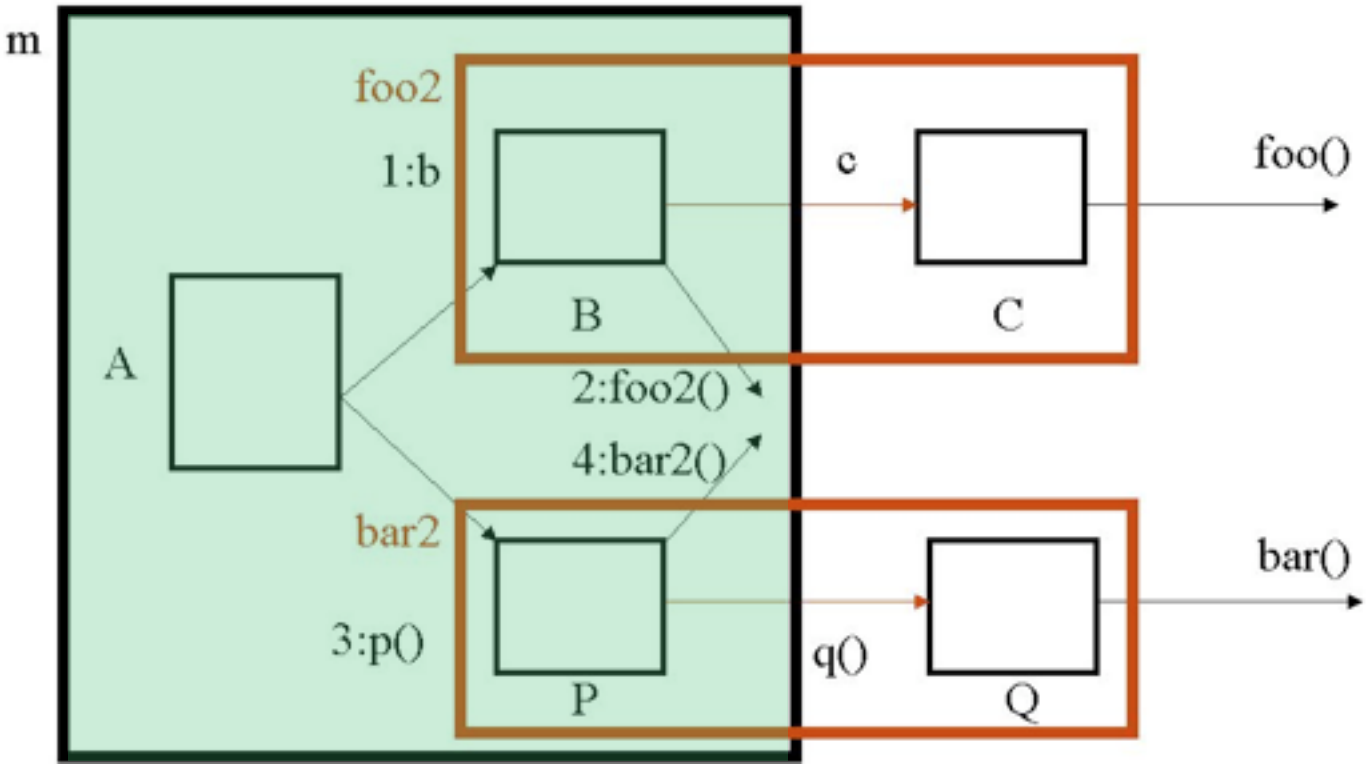
```
class A {public: void m(); P p(); B b; };
class B {public: C c; };
class C {public: void foo(); };
class P {public: Q q(); };
class Q {public: void bar(); };
void A::m() {
    this.b.c.foo(); this.p().q().bar();}
```

Do not skip intermediaries!

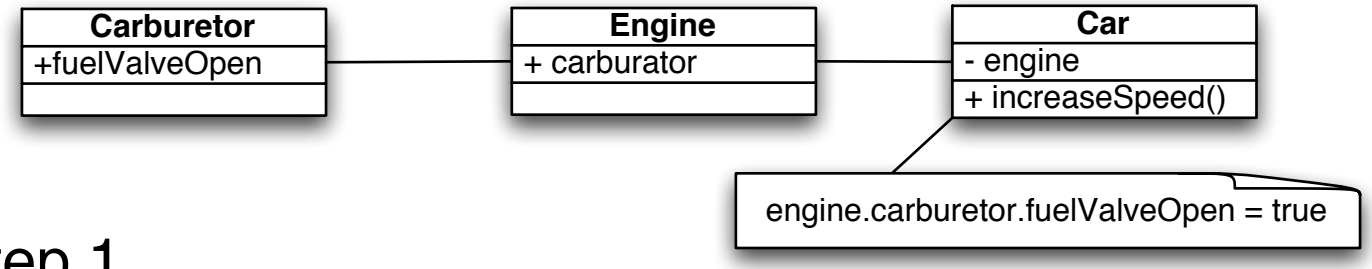


Solution

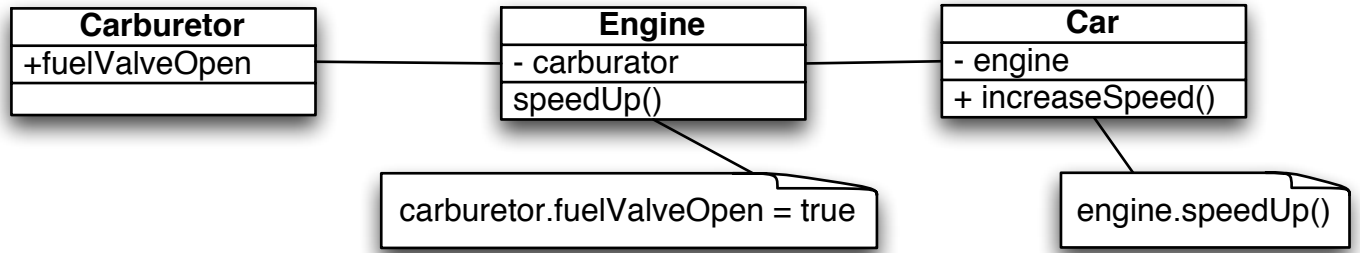
- Follow the Law of Demeter



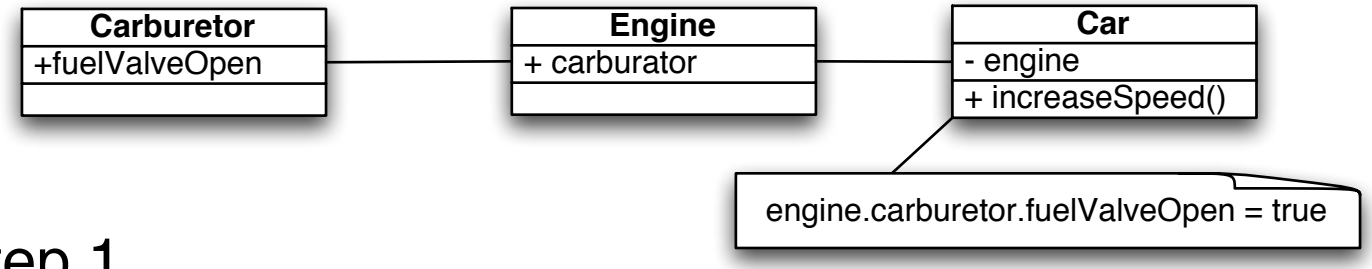
Transformation



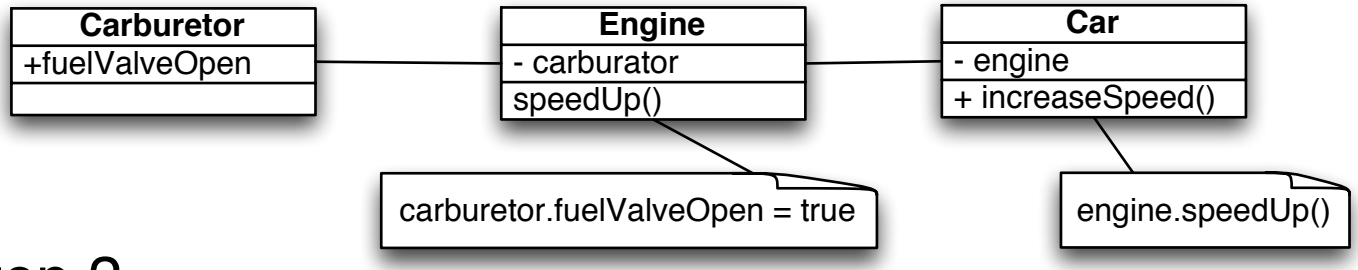
Step 1



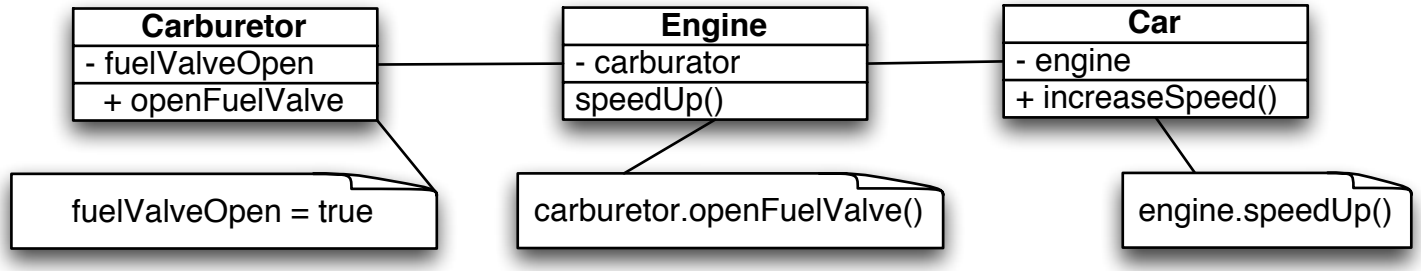
Transformation



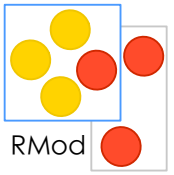
Step 1



Step 2

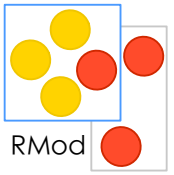


The Dark side of the Law of Demeter




```
Class A
        instVar: myCollection
A » do: aBlock
        myCollection do: aBlock
A » collect: aBlock
        ^ myCollection collect: aBlock
A » select: aBlock
        ^ myCollection select: aBlock
A » detect: aBlock
        ^ myCollection detect: aBlock
A » isEmpty
        ^ myCollection isEmpty
…
```

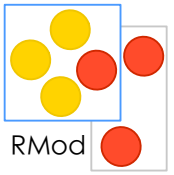
The Dark side of the Law of Demeter




```
Class A
        instVar: myCollection
A » do: aBlock
        myCollection do: aBlock
A » collect: aBlock
        ^ myCollection collect: aBlock
A » select: aBlock
        ^ myCollection select: aBlock
A » detect: aBlock
        ^ myCollection detect: aBlock
A » isEmpty
        ^ myCollection isEmpty
…
```

Each object itself has to provide a complete interface

About Accessor methods

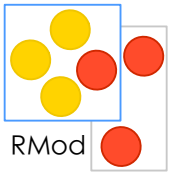


Some schools say: “Access all instance variables using accessor methods”

But

- Be consistent inside a class:
  - do not mix direct access and accessor use
- Initially: think of accessors as protected methods that should not be invoked by clients
  - put them in the “private” protocol
- Put accessors in “accessing” protocol only when necessary

Example

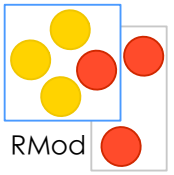



```
Scheduler » initialize
        self tasks: OrderedCollection new.

Scheduler » tasks
        ^ tasks
```

But now everybody can tweak the tasks!

Accessors

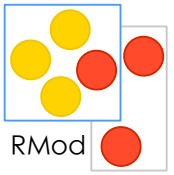


Accessors are good for lazy initialization

```
Scheduler » tasks
    tasks isNil ifTrue: [tasks := ...].
    ^ tasks
```

But: accessors methods should be protected by default, at least at the beginning

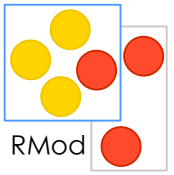
Accessors open Encapsulation



- The fact that accessors are methods doesn't support good data encapsulation.
- You could be tempted to write in a client:

```
SchedulerView » addTaskButton
    ...
    model tasks add: newTask
```

- What will happen if we change the representation of tasks in Scheduler?



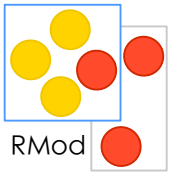
If tasks is now an array it will break

Limit the coupling between your objects!
Provide a good interface for your clients:

```
Scheduler » addTask: aTask
    tasks add: aTask

SchedulerView » addTaskButton
    ...
    model addTask: newTask
```

Copying accessors



Should an accessor copy the structure?

```
Scheduler » tasks
    ^ tasks copy
```

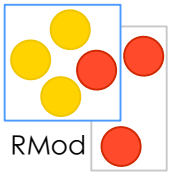
This protects the internals of the scheduler from meddling clients

```
Scheduler uniqueInstance tasks removeFirst
```

does nothing.

- clients can get confused!
- accessor is much more costly to use

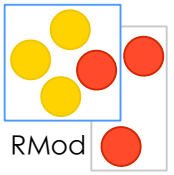
Use intention revealing names



Better

```
Scheduler » copiedTasks
    "returns a copy of the pending tasks"
     ^ task copy
```

Provide a Complete Interface




```
Workstation » accept: aPacket
        aPacket addressee = self address
                ifTrue:[…]
                ifFalse: […]
```

- It is the responsibility of an object to offer a complete interface that protects itself from client intrusion.
- Shift the responsibility to the Packet object

```
Packet » isAddressedTo: aNode
                ^ addressee = aNode address
Workstation » accept: aPacket
        (aPacket isAddressedTo: self)
                ifTrue:[…]
                ifFalse: […]
```