Creating Flexible Interfaces

Based on POODR Chapter 4

Requests, not Objects

- Domain objects correspond to nouns —
  - easy to find, but:
  - *not* at the design center of your application.
  - can be a *trap* for the unwary: important objects may be missing
- Sequence diagrams are a vehicle for exposing, experimenting with, and ultimately defining [the requests that pass between objects, that is,] ... interfaces.
- “I need to send this message, who should respond to it?” is the first step [towards more flexible applications].
- You don’t send messages because you have objects, you have objects because you send messages.

Objects Behave

Focus *Not* on the data in an object, but on

- the requests that are made on it (its *interface*)
- the requests that it makes of its collaborators

- Think about "co-data" rather than data
- *Any* object can be represented without data
  - *except* for the stream of requests that has been made of it (with their arguments)

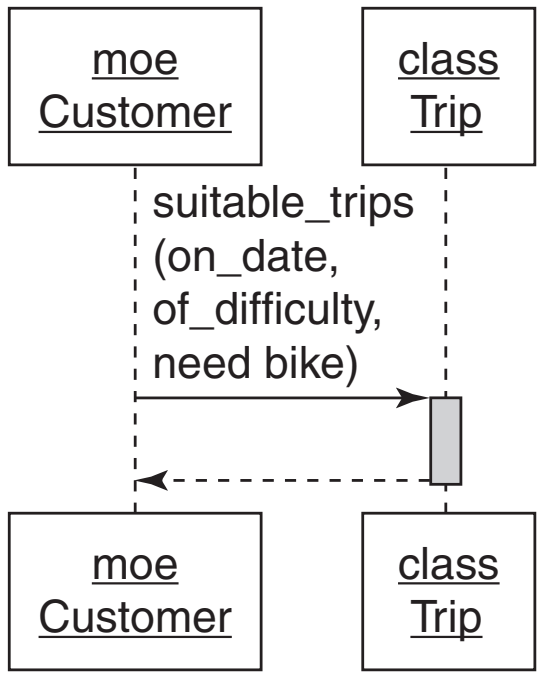


Figure 4.3 A simple sequence diagram.

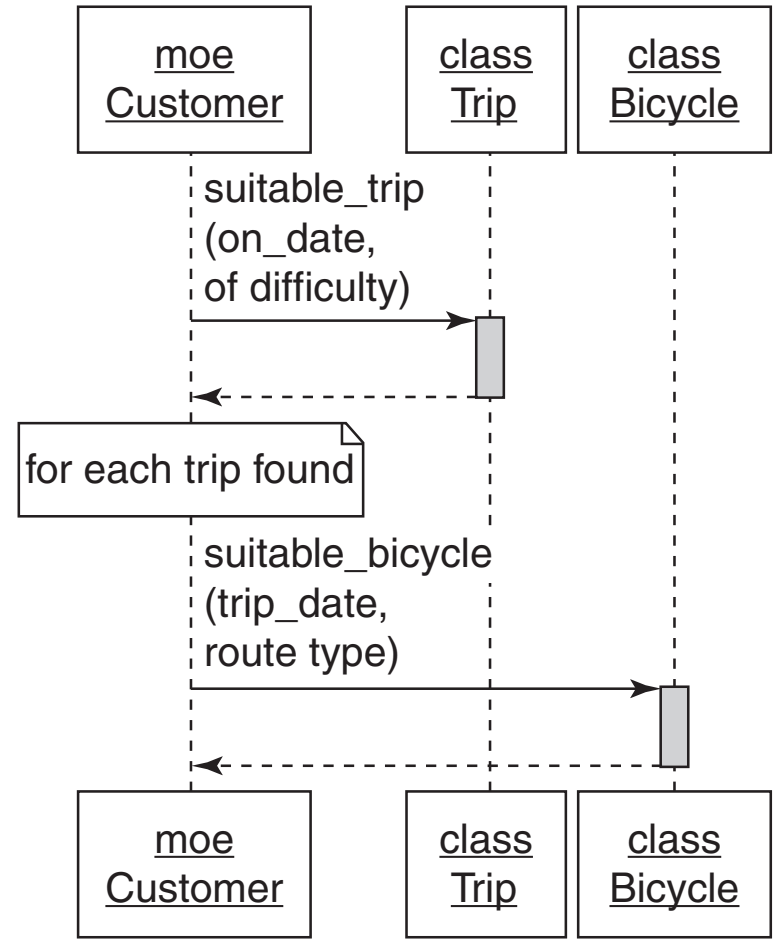


Figure 4.4 Moe talks to trip and bicycle.

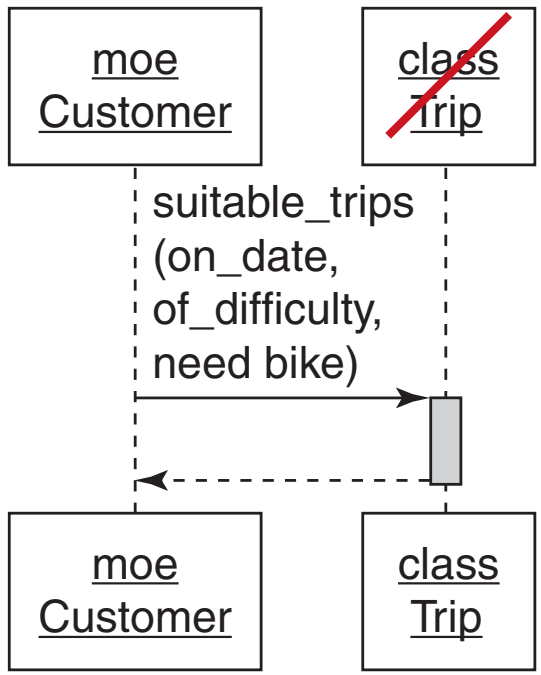


Figure 4.3 A simple sequence diagram.

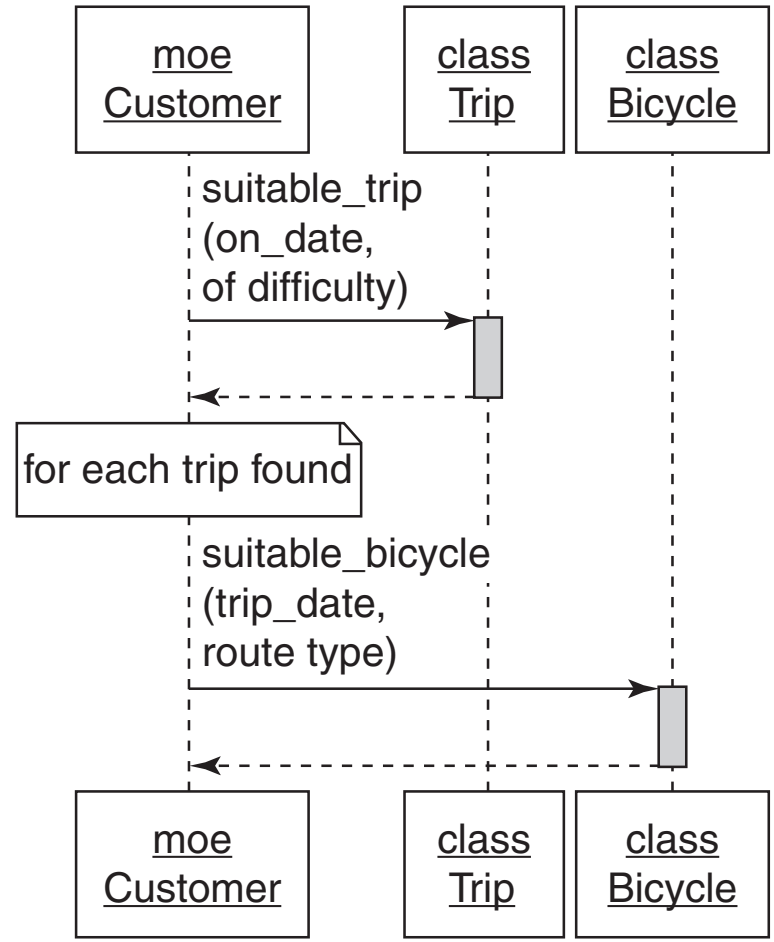


Figure 4.4 Moe talks to trip and bicycle.

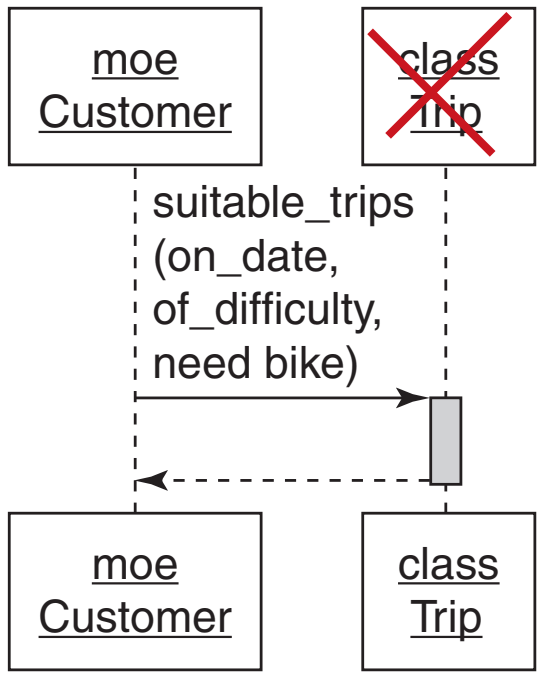


Figure 4.3 A simple sequence diagram.

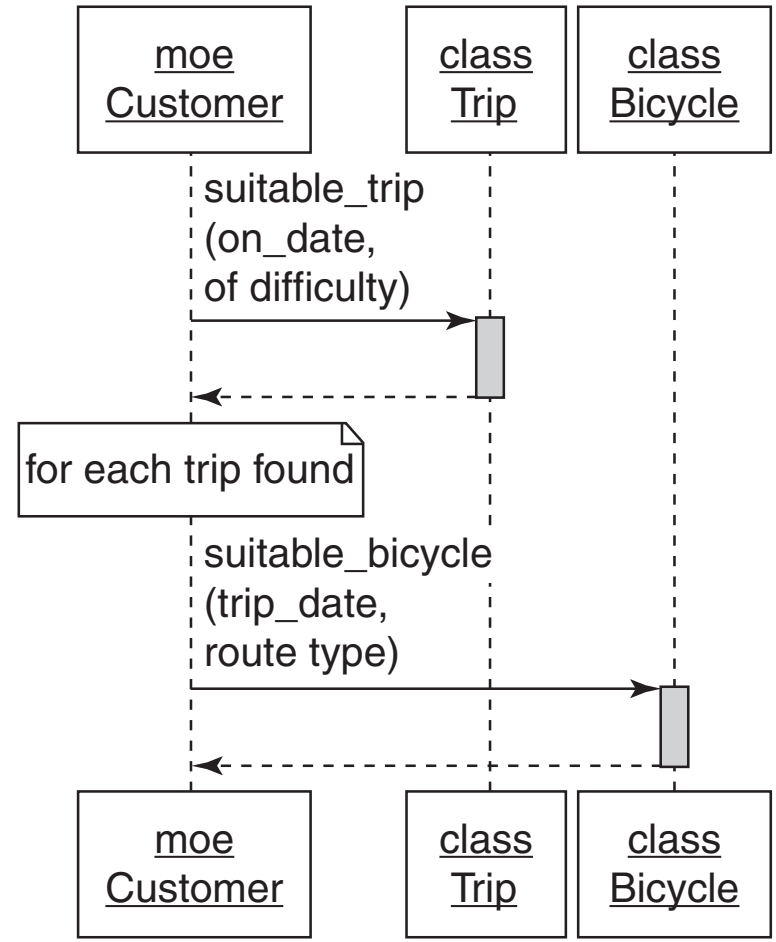


Figure 4.4 Moe talks to trip and bicycle.

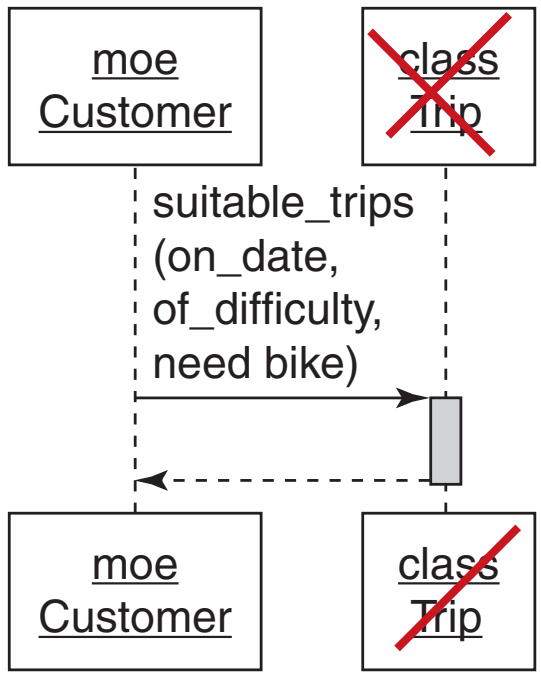


Figure 4.3 A simple sequence diagram.

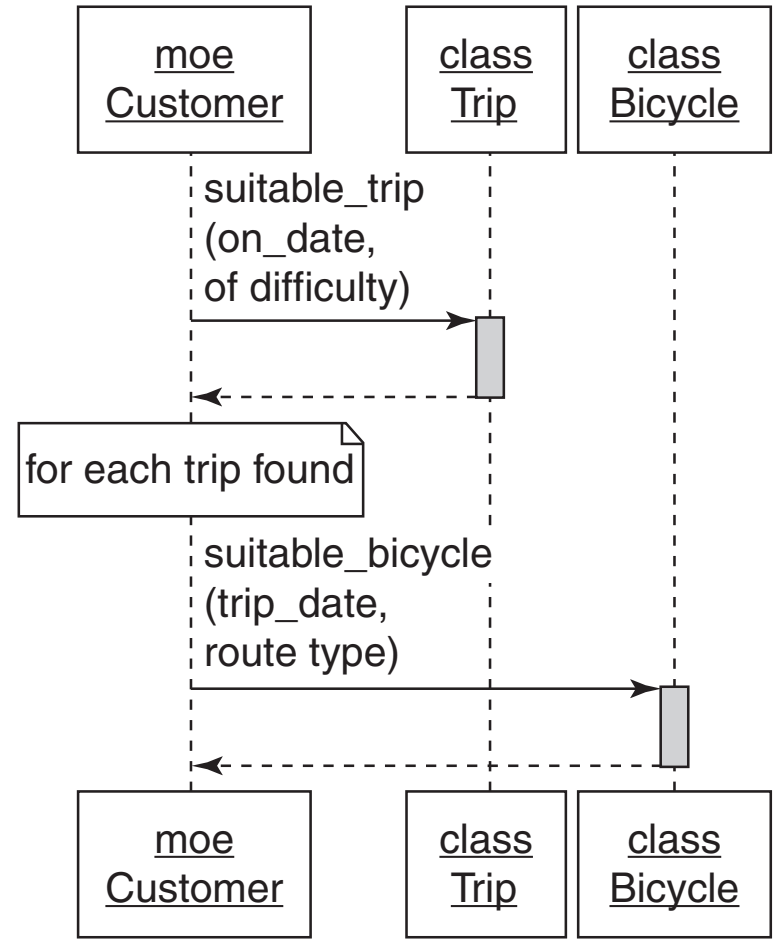


Figure 4.4 Moe talks to trip and bicycle.

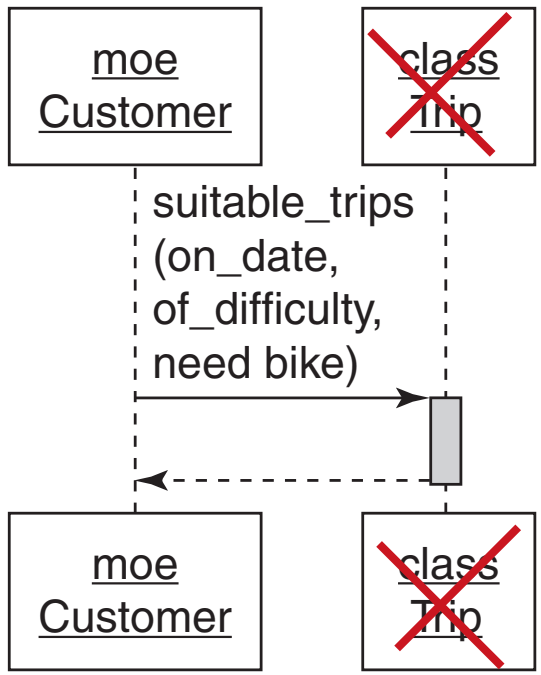


Figure 4.3 A simple sequence diagram.

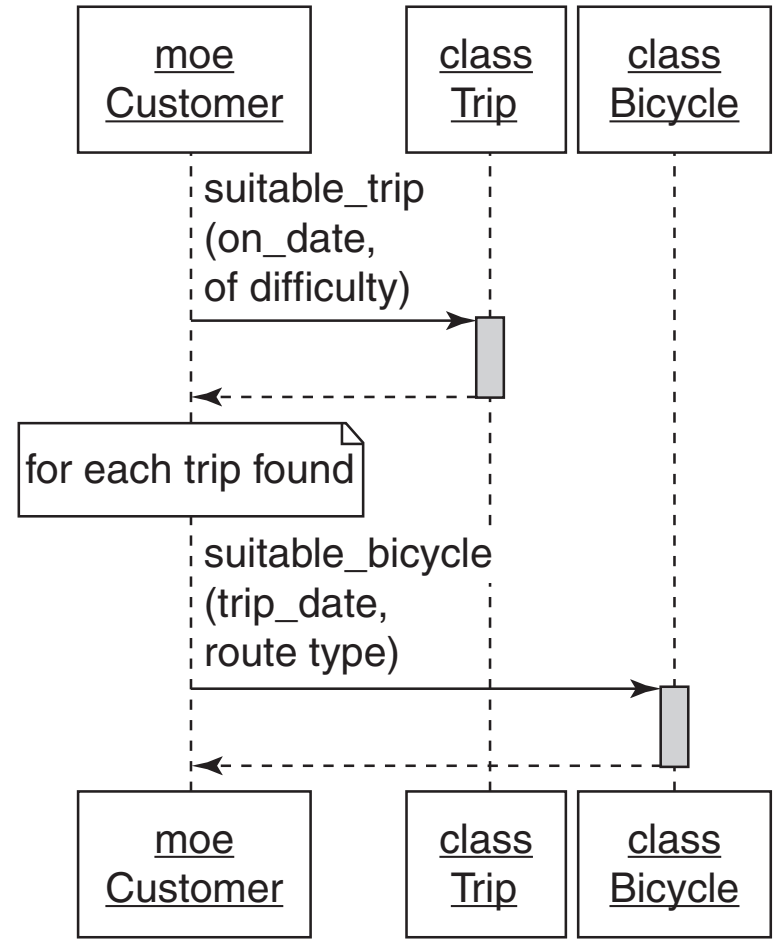


Figure 4.4 Moe talks to trip and bicycle.

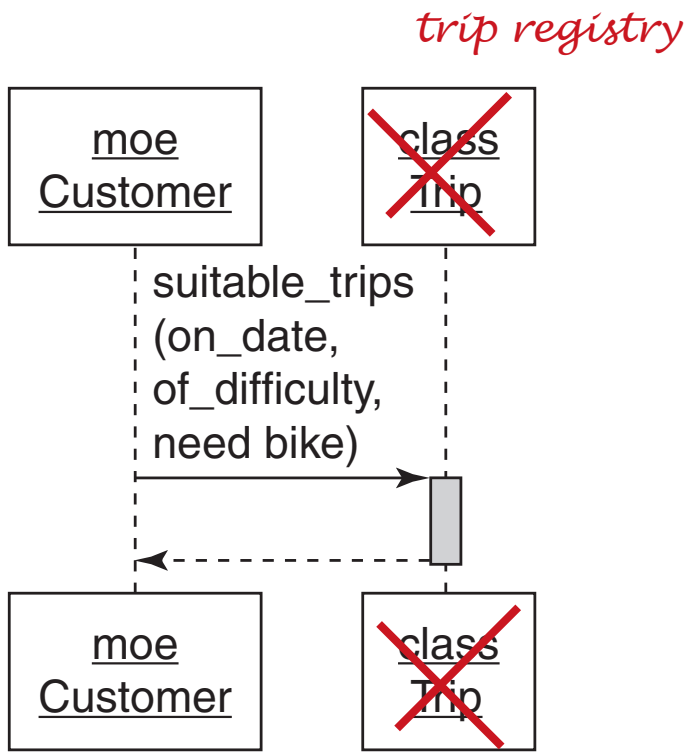


Figure 4.3 A simple sequence diagram.

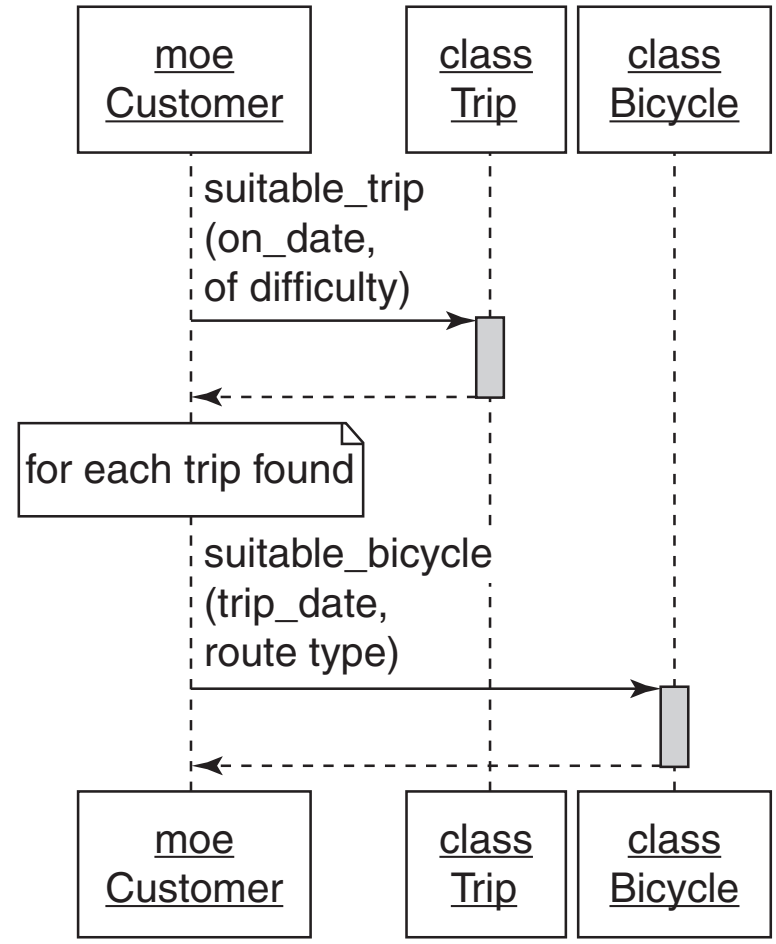


Figure 4.4 Moe talks to trip and bicycle.

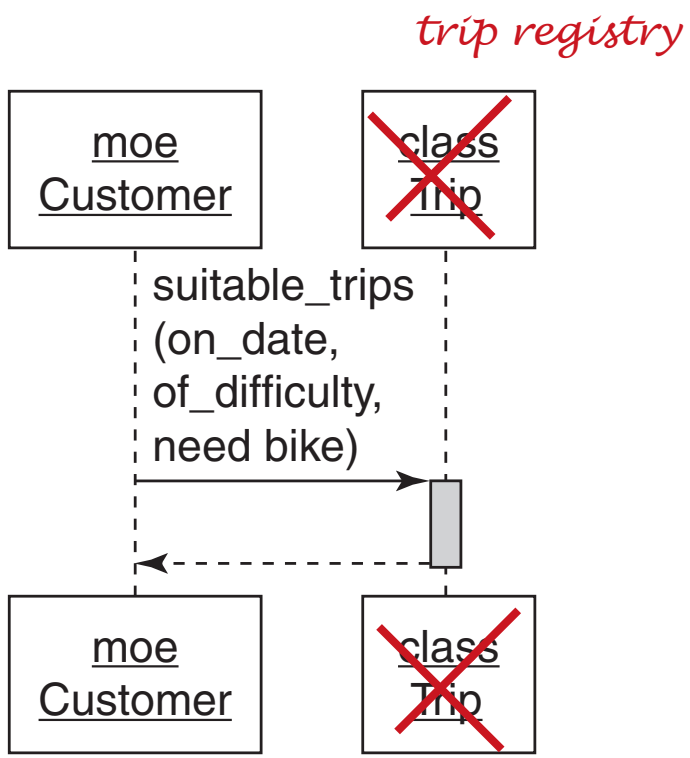


Figure 4.3 A simple sequence diagram.

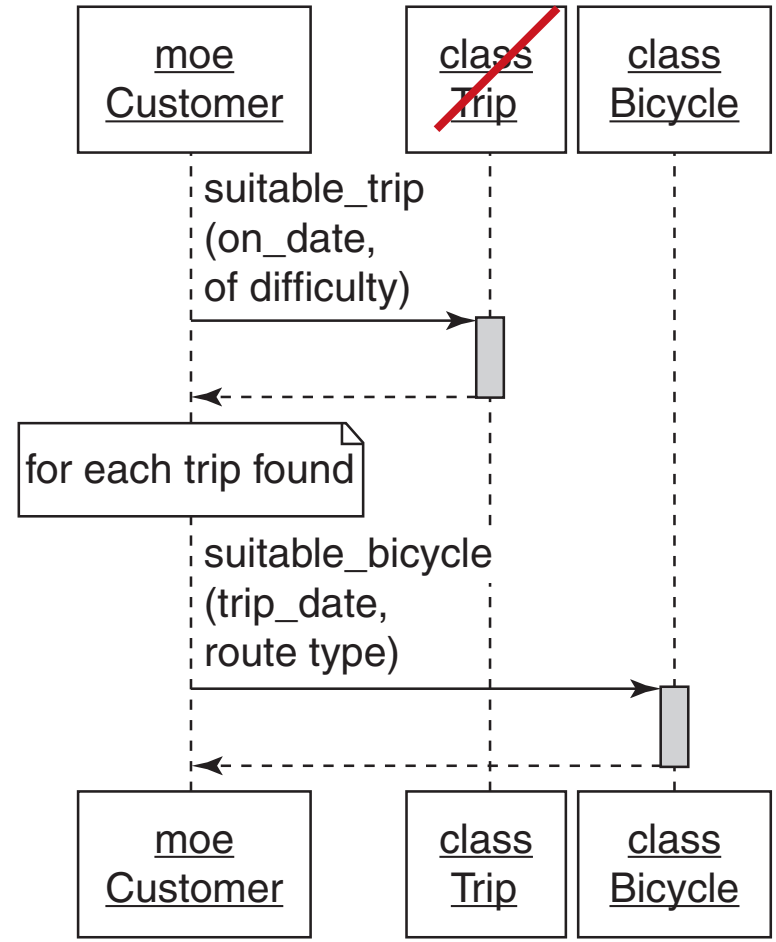


Figure 4.4 Moe talks to trip and bicycle.

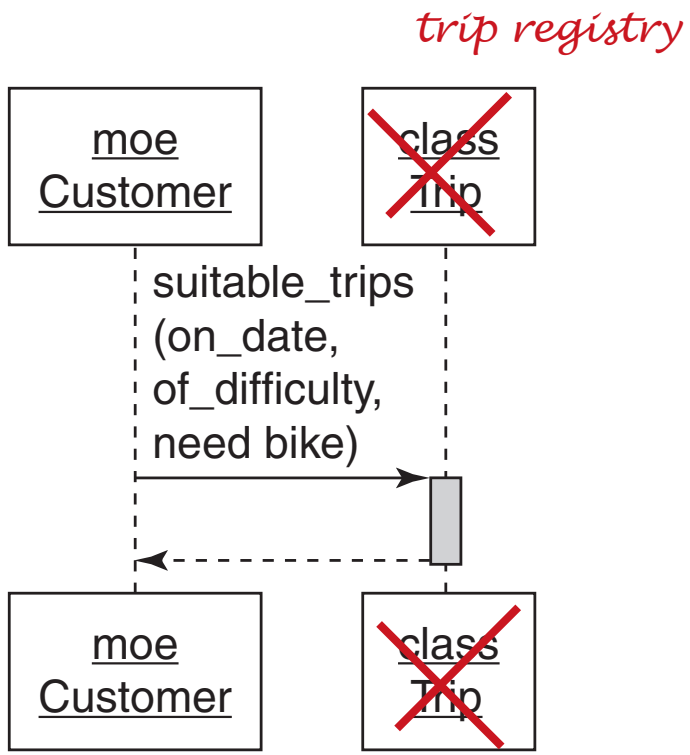


Figure 4.3 A simple sequence diagram.

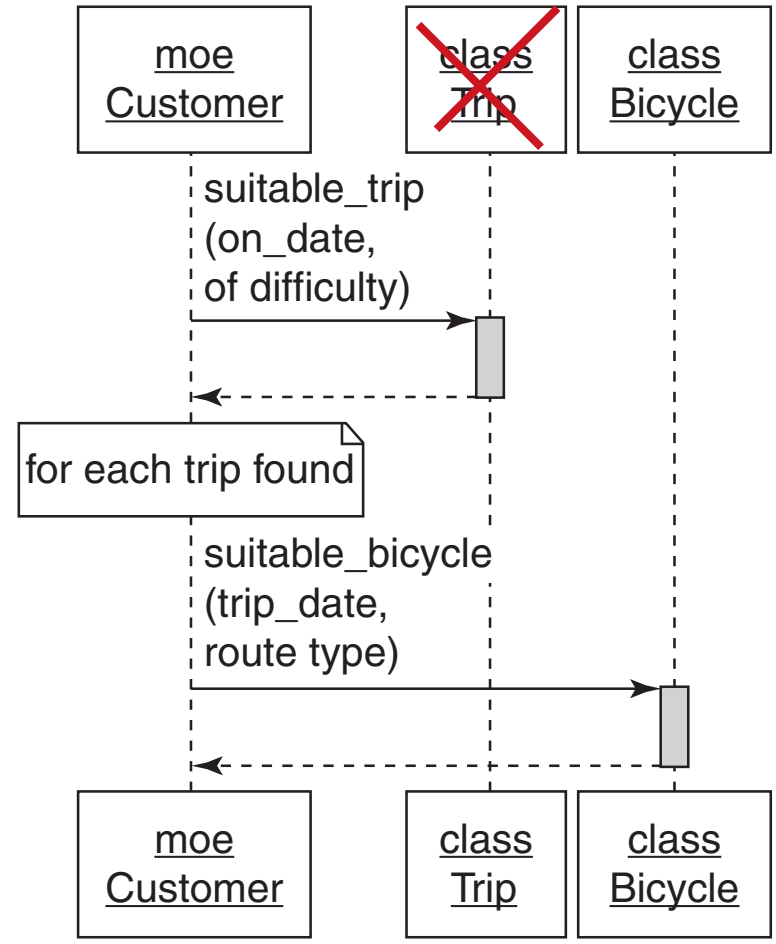


Figure 4.4 Moe talks to trip and bicycle.

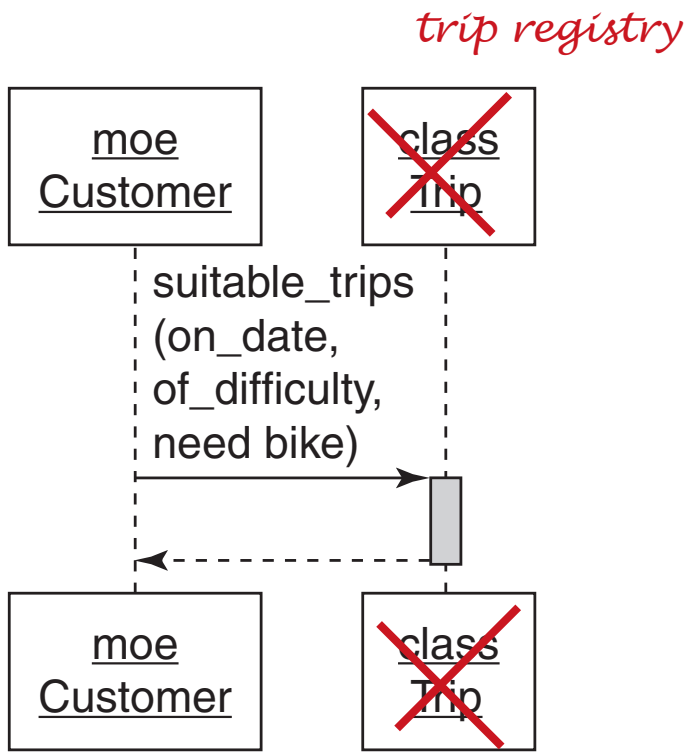


Figure 4.3 A simple sequence diagram.

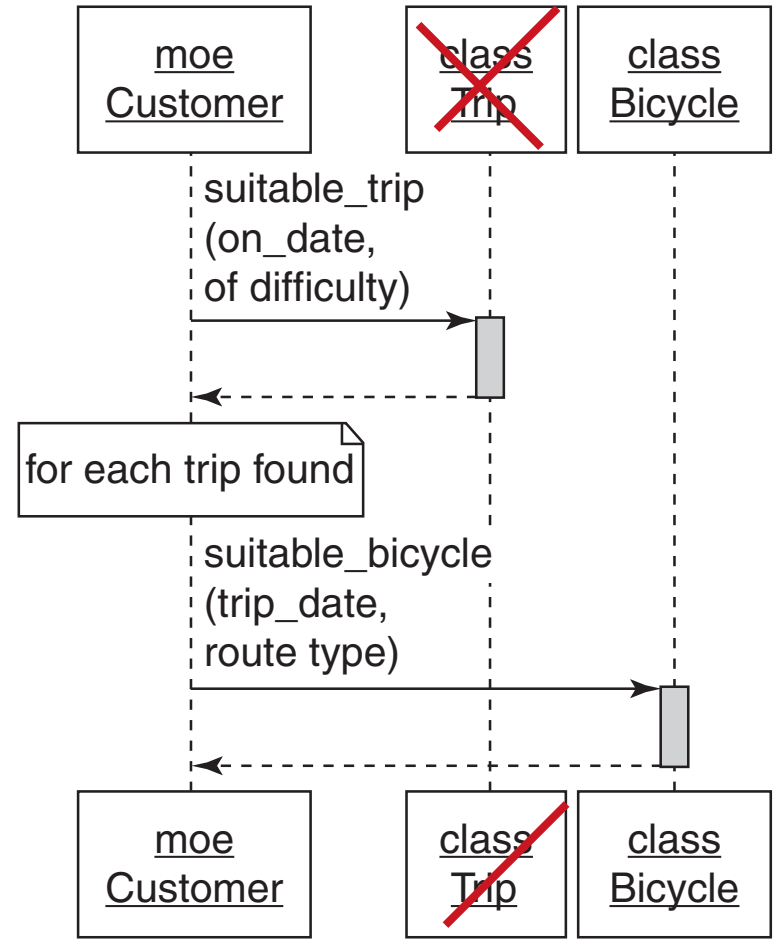


Figure 4.4 Moe talks to trip and bicycle.

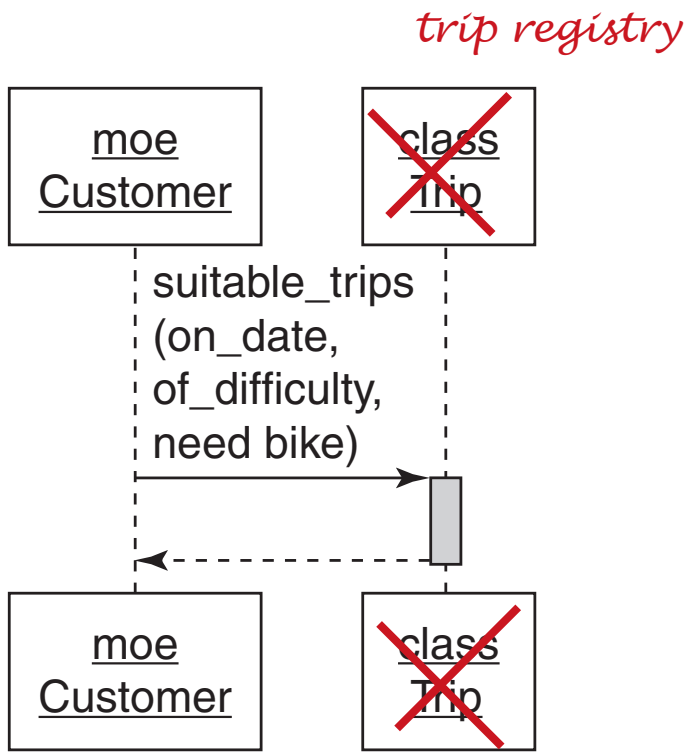


Figure 4.3 A simple sequence diagram.

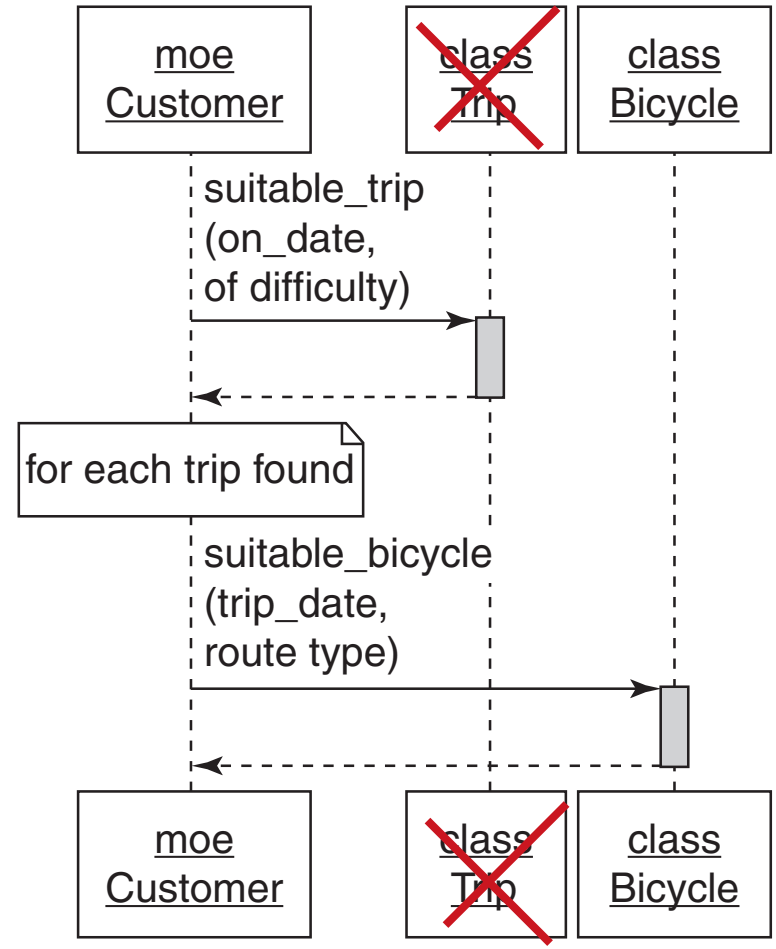


Figure 4.4 Moe talks to trip and bicycle.

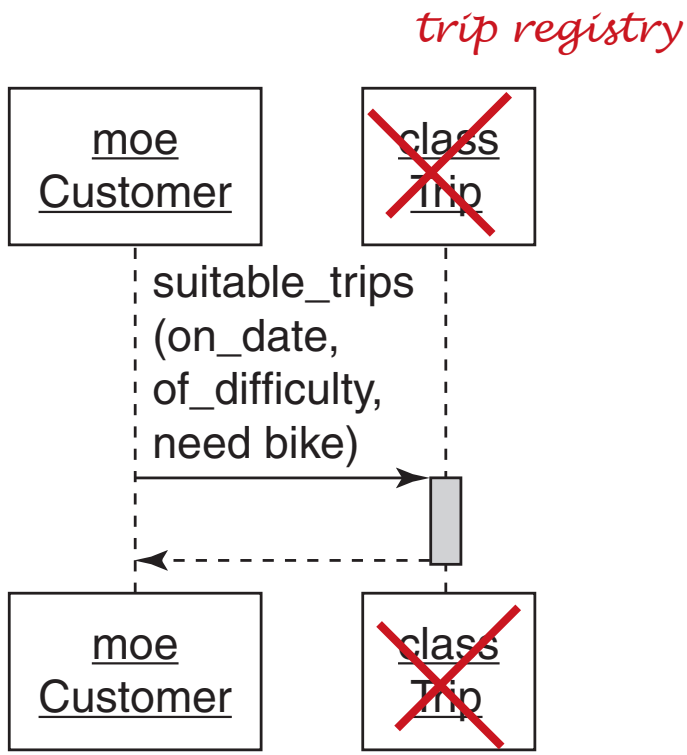


Figure 4.3 A simple sequence diagram.

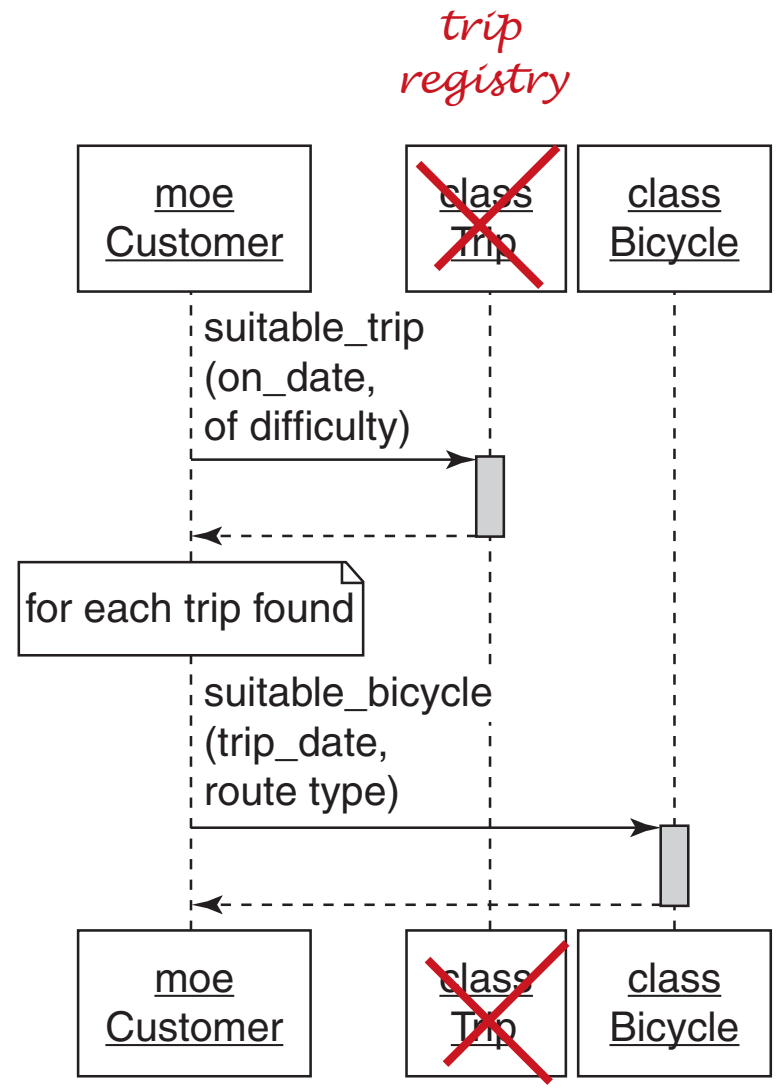


Figure 4.4 Moe talks to trip and bicycle.

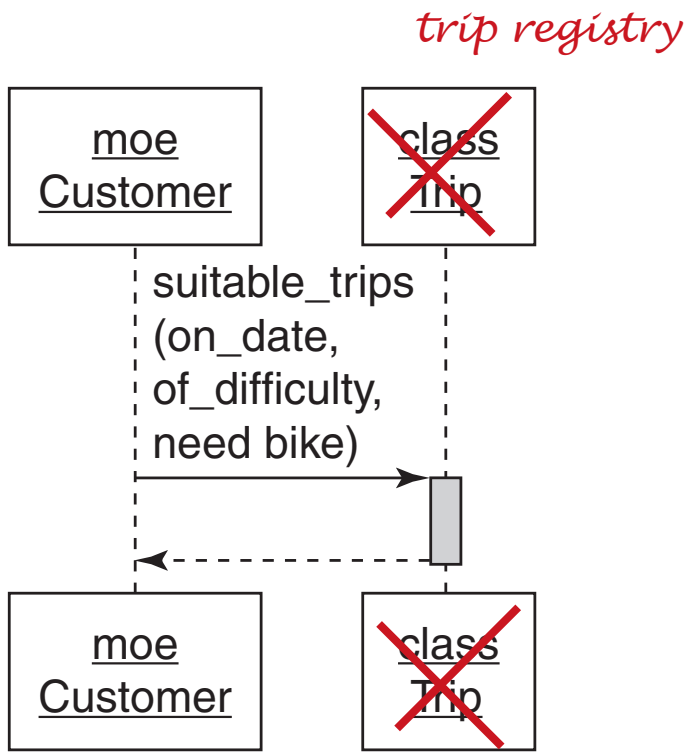


Figure 4.3 A simple sequence diagram.

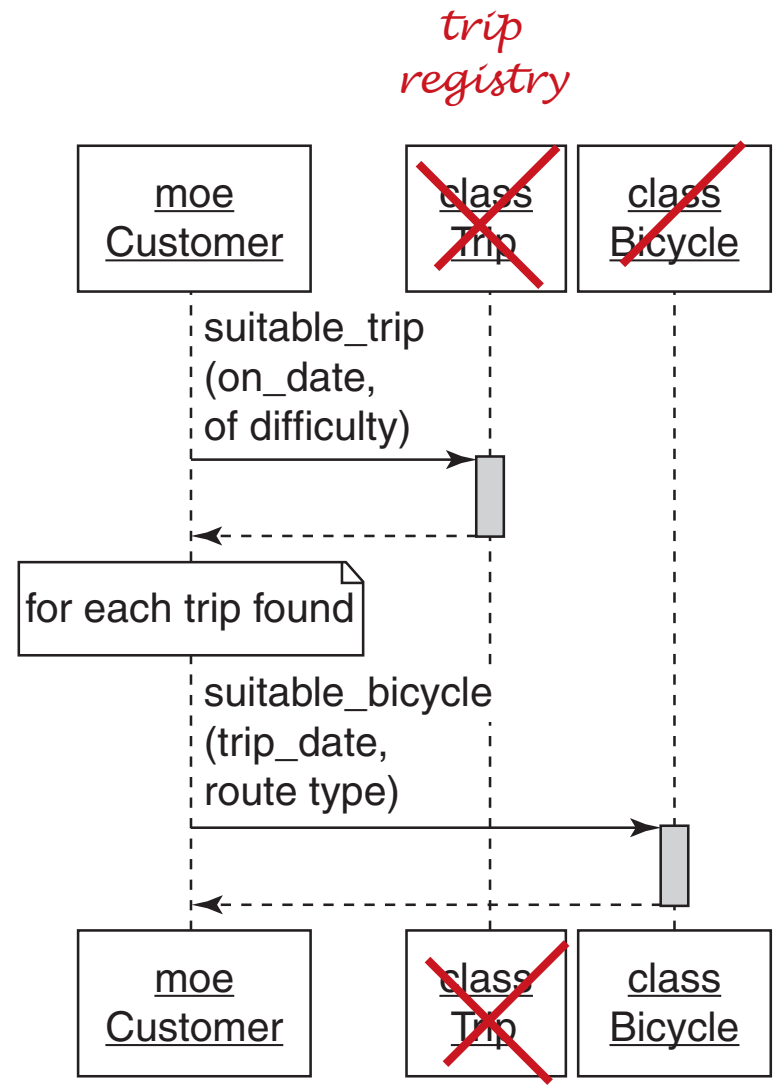


Figure 4.4 Moe talks to trip and bicycle.

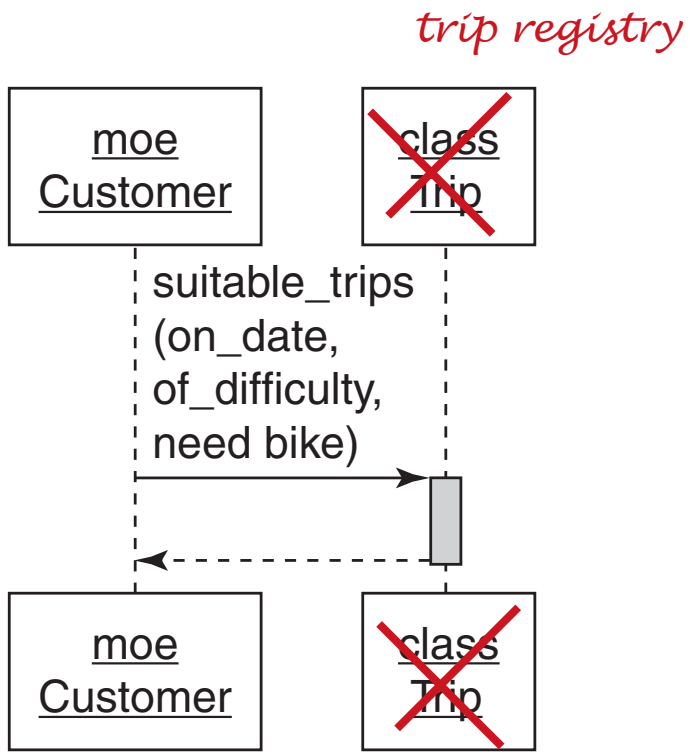


Figure 4.3 A simple sequence diagram.

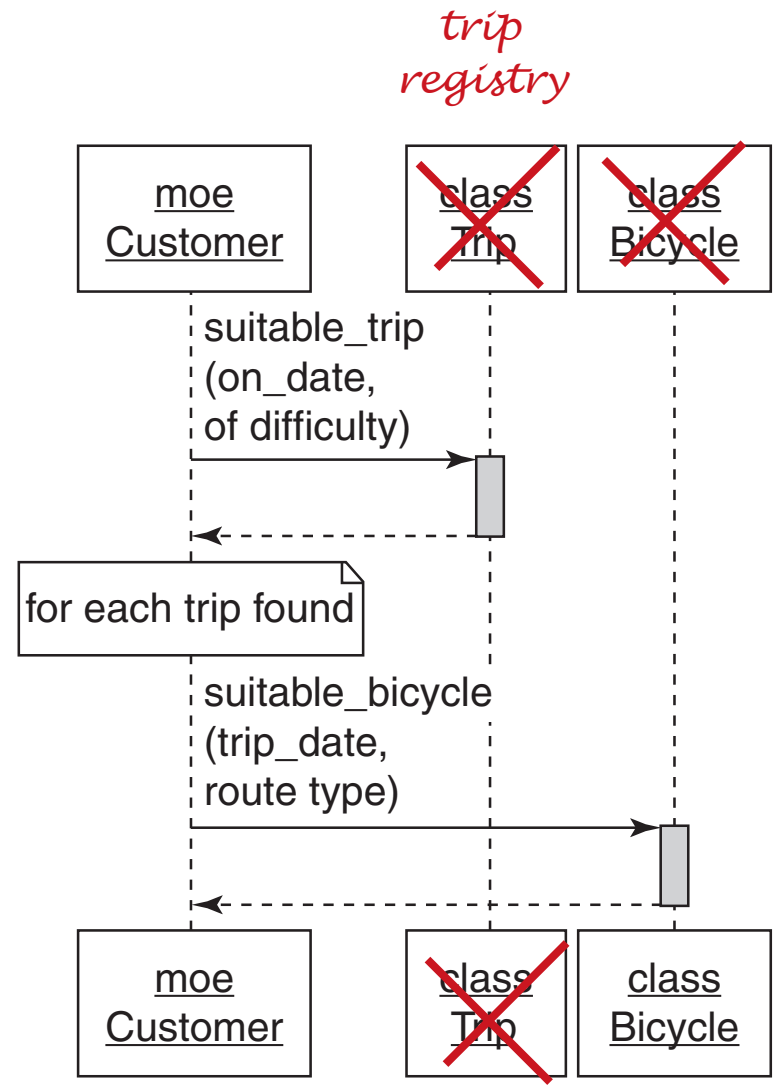


Figure 4.4 Moe talks to trip and bicycle.

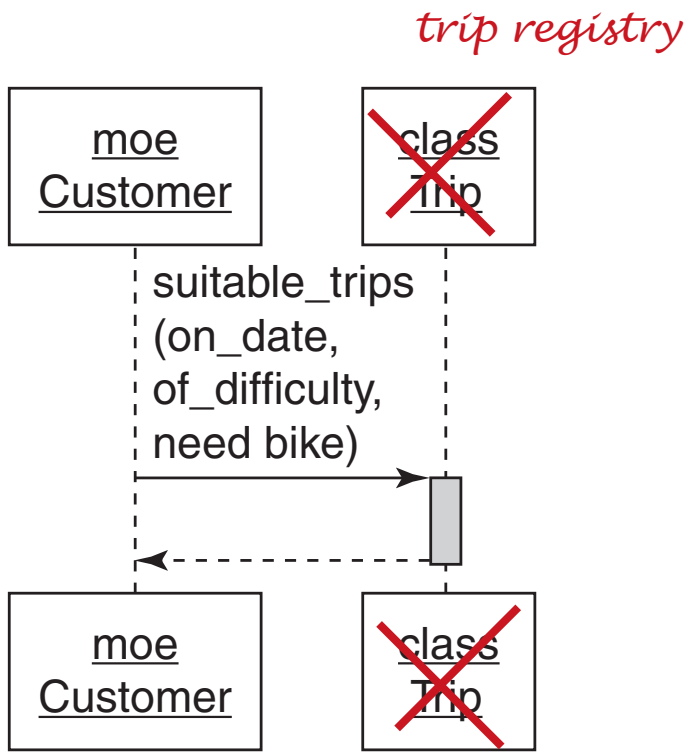


Figure 4.3 A simple sequence diagram.

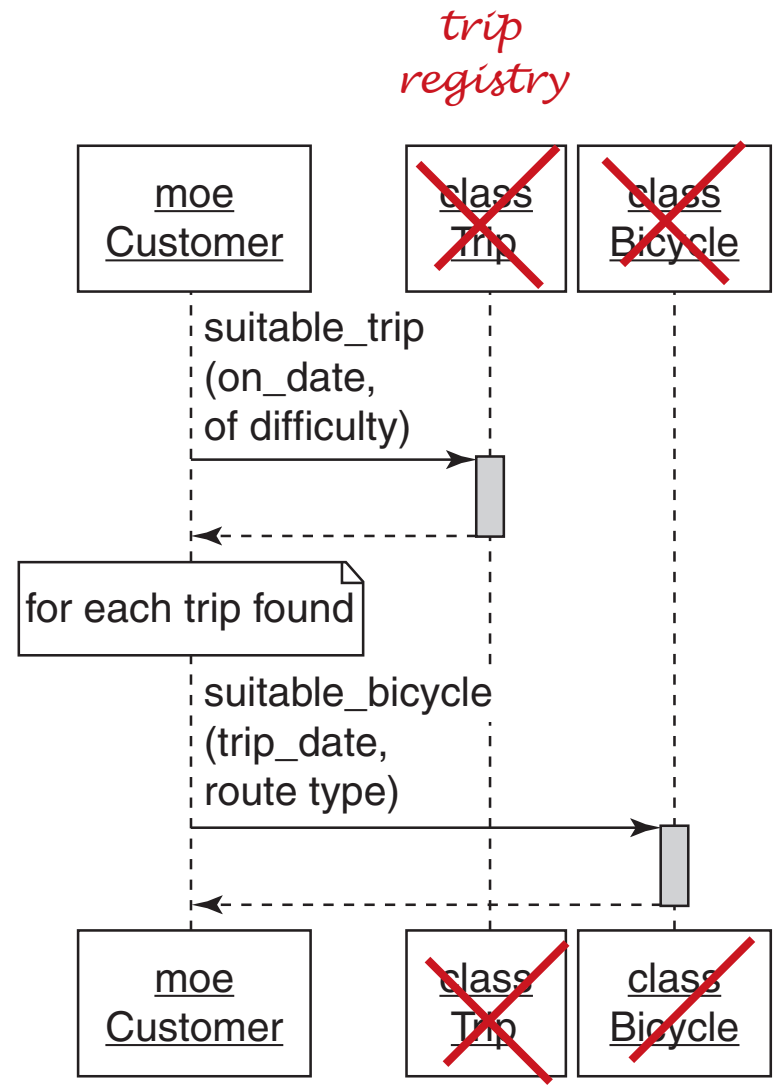


Figure 4.4 Moe talks to trip and bicycle.

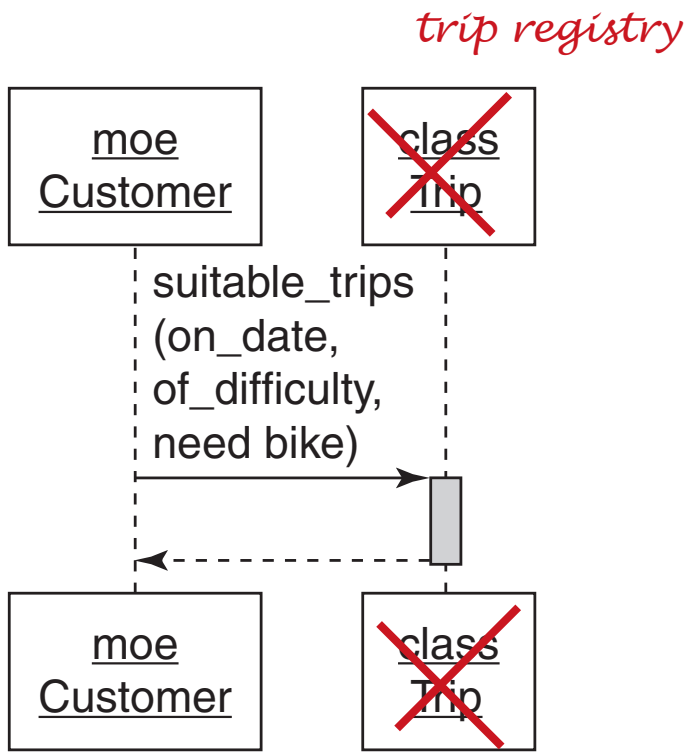


Figure 4.3 A simple sequence diagram.

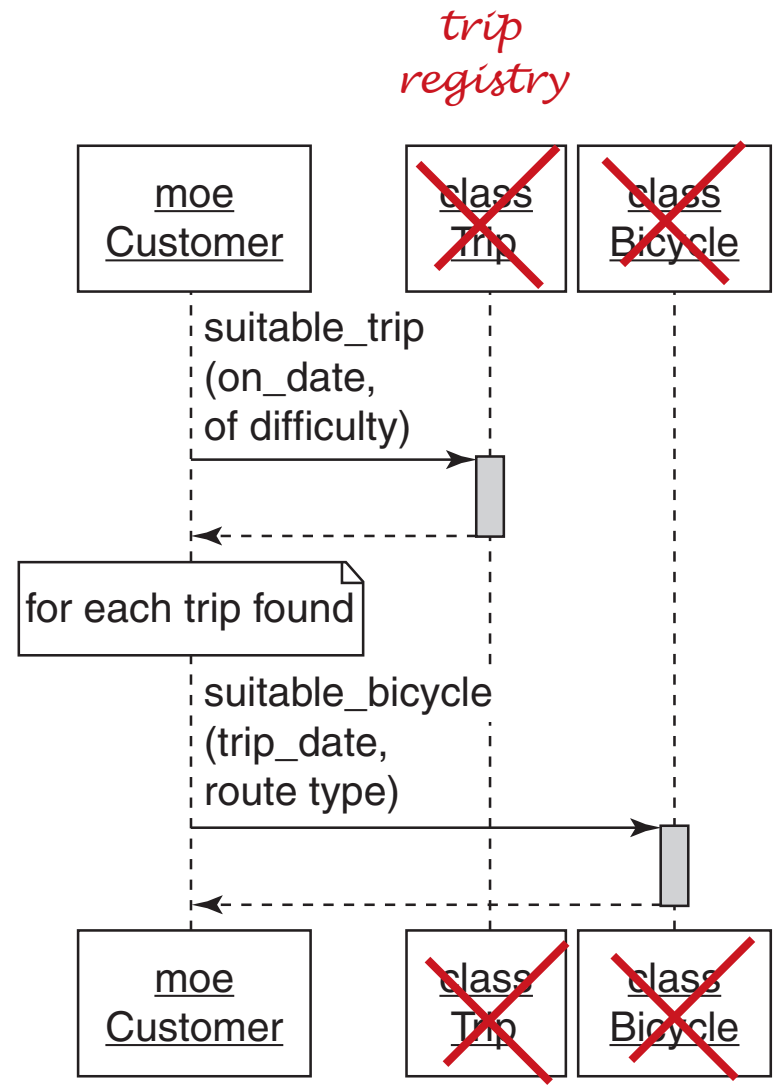


Figure 4.4 Moe talks to trip and bicycle.

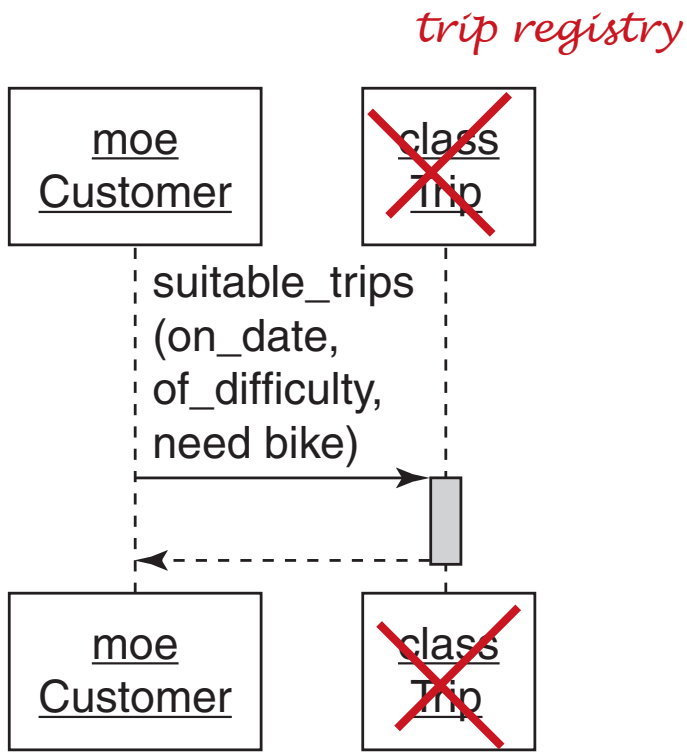


Figure 4.3 A simple sequence diagram.

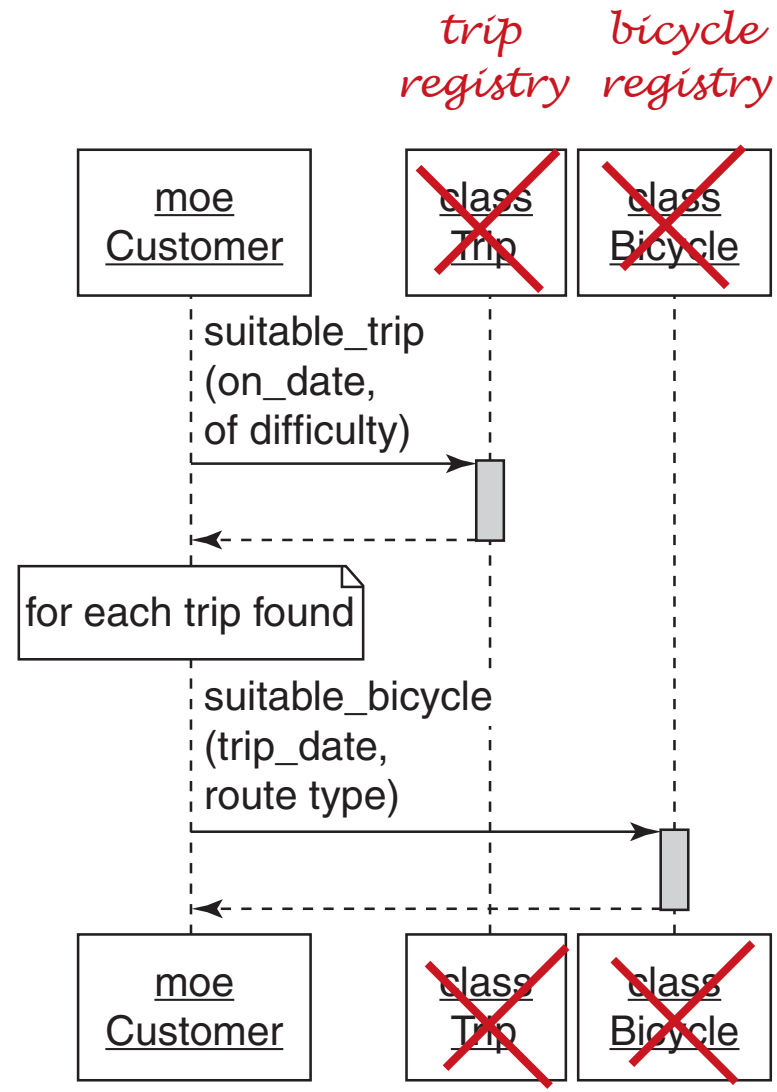


Figure 4.4 Moe talks to trip and bicycle.

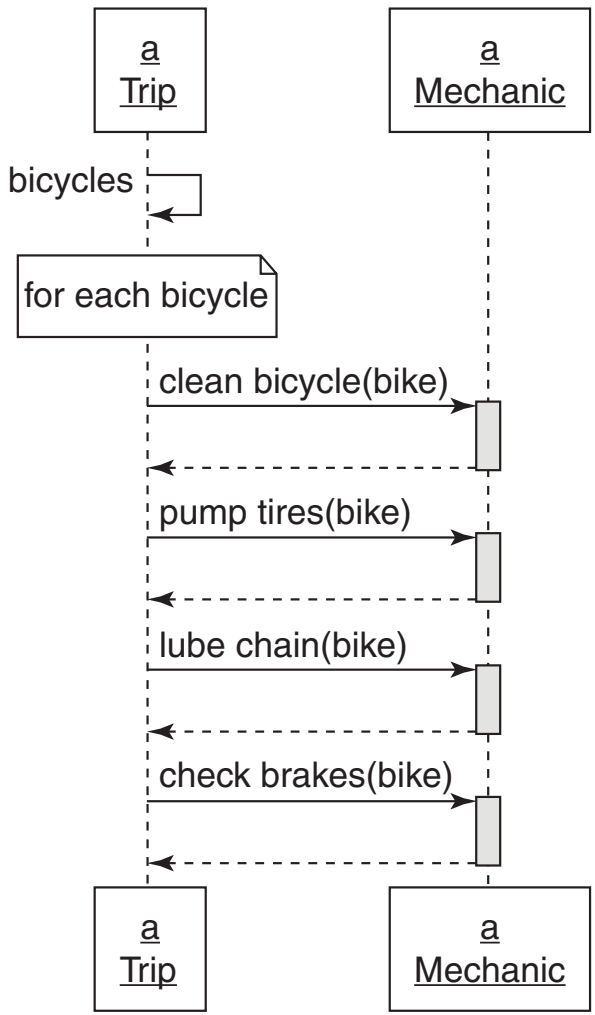


Figure 4.5
A *Trip* tells a *Mechanic* how to prepare each *Bicycle*.

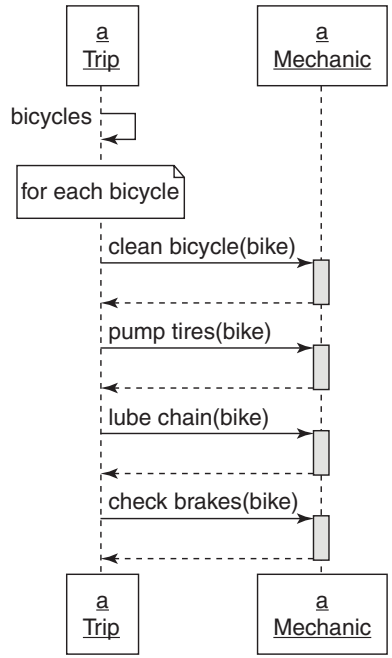


Figure 4.5

Figure 4.5 is quite procedural. A `Trip` tells a `Mechanic` how to prepare a `Bicycle`, almost as if `Trip` were the main program and `Mechanic` a bunch of callable functions. In this design, `Trip` is the only object that knows exactly how to prepare a bike; getting a bike prepared requires using a `Trip` or duplicating the code. `Trip`'s context is large, as is `Mechanic`'s public interface. These two classes are not islands with bridges between them, they are instead a single, woven cloth.

Many new object-oriented programmers start out working just this way, writing procedural code. It's inevitable; this style closely mirrors the best practices of their former procedural languages. Unfortunately, coding in a procedural style defeats the purpose of object orientation. It reintroduces the exact maintenance issues that OOP is designed to avoid.

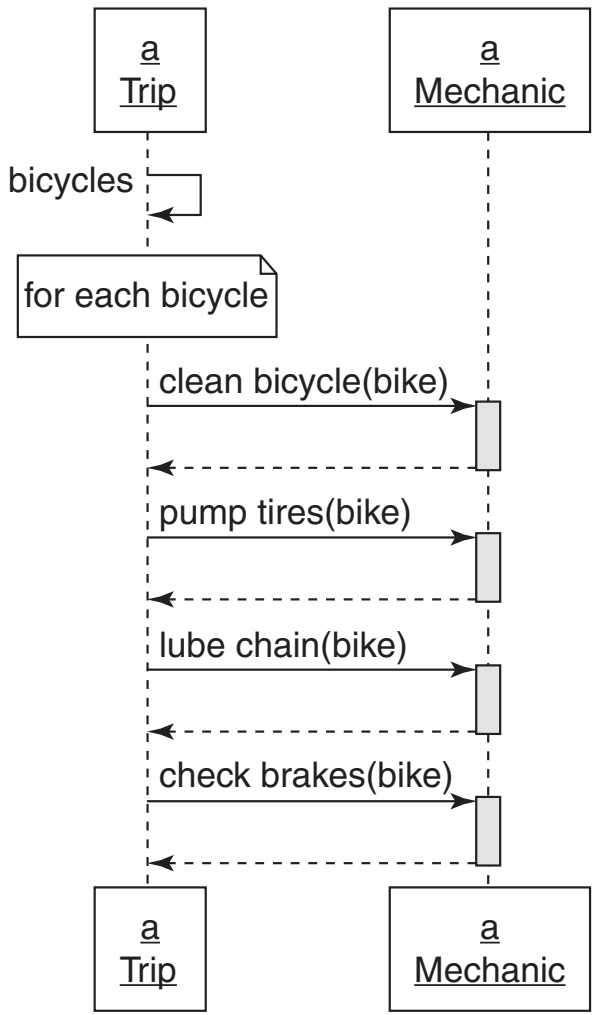


Figure 4.5
A *Trip* tells a *Mechanic* how to prepare each *Bicycle*.

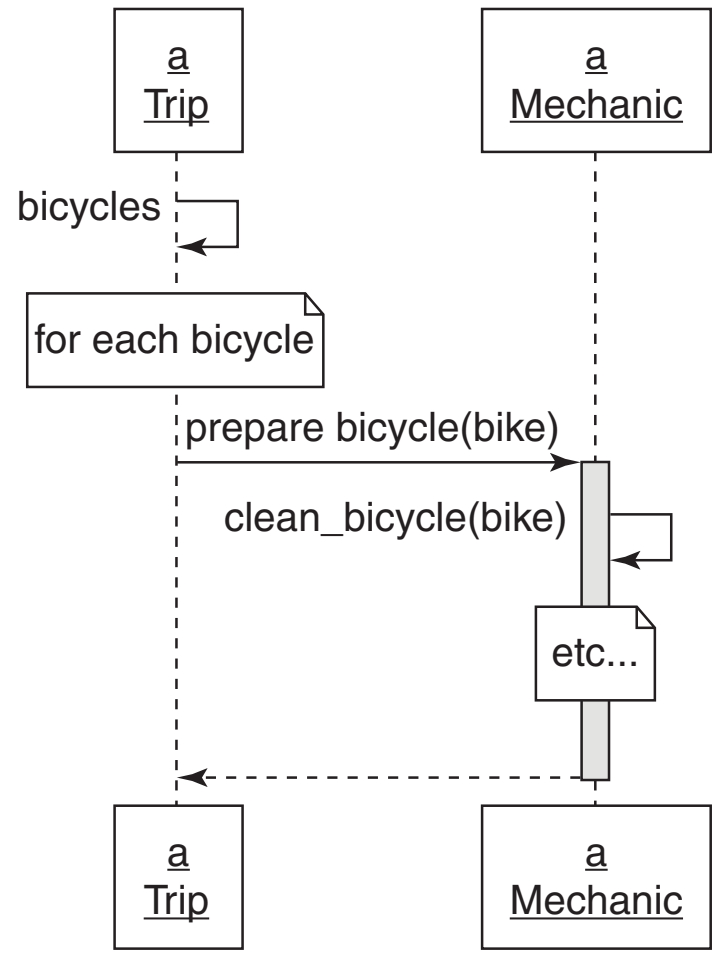
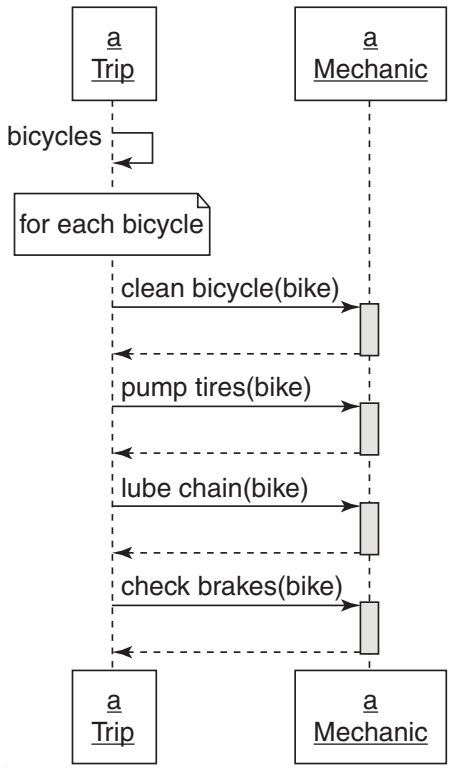


Figure 4.6 A *Trip* asks a *Mechanic* to prepare each *Bicycle*.

Figure 4.6 is more object-oriented. Here, a `Trip` *asks* a `Mechanic` to prepare a `Bicycle`. `Trip`'s context is reduced, and `Mechanic`'s public interface is smaller. Additionally, `Mechanic`'s public interface is now something that any object may profitably use; you don't need a `Trip` to prepare a bike. These objects now communicate in a few well-defined ways; they are less coupled and more easily reusable.

This style of coding places the responsibilities in the correct objects, a great improvement, but continues to require that `Trip` have more context than is necessary. `Trip` still knows that it holds onto an object that can respond to `prepare_bicycle`, and it must *always* have this object.



Trip
tells *Mechanic* how to prepare each *Bicycle*.

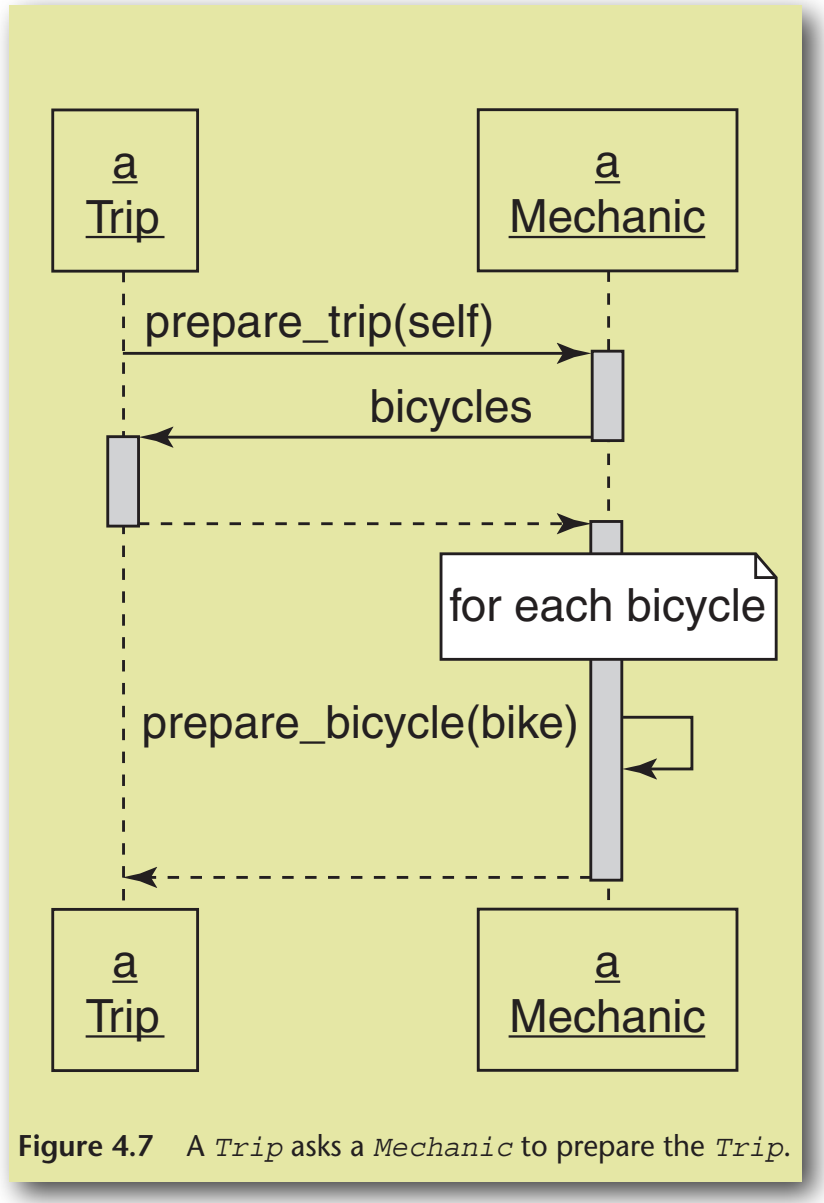


Figure 4.7 A *Trip* asks a *Mechanic* to prepare the *Trip*.

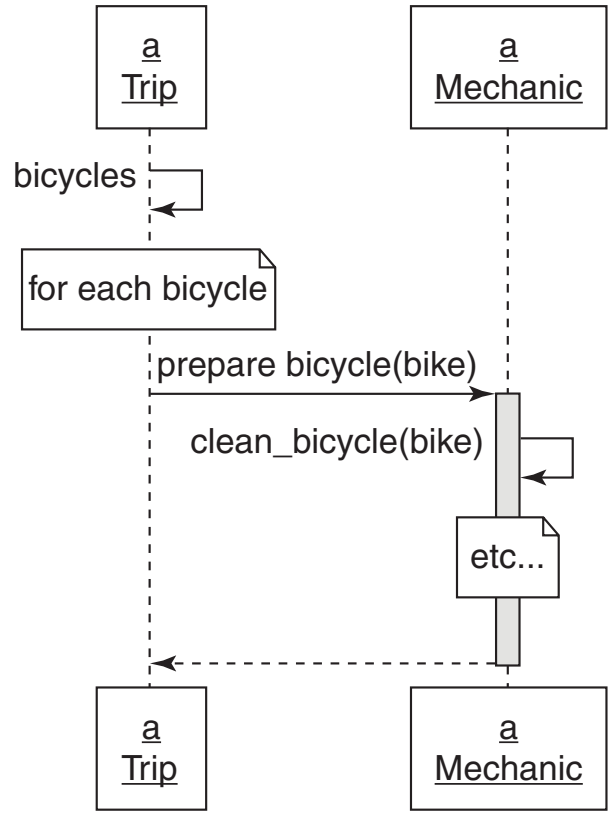
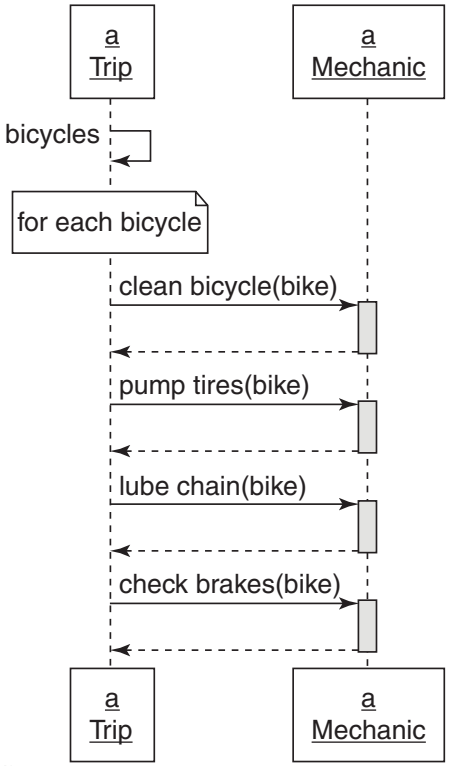


Figure 4.6 A *Trip* asks a *Mechanic* to prepare each *Bicycle*.



Trip tells *Mechanic* how to prepare each *Bicycle*.

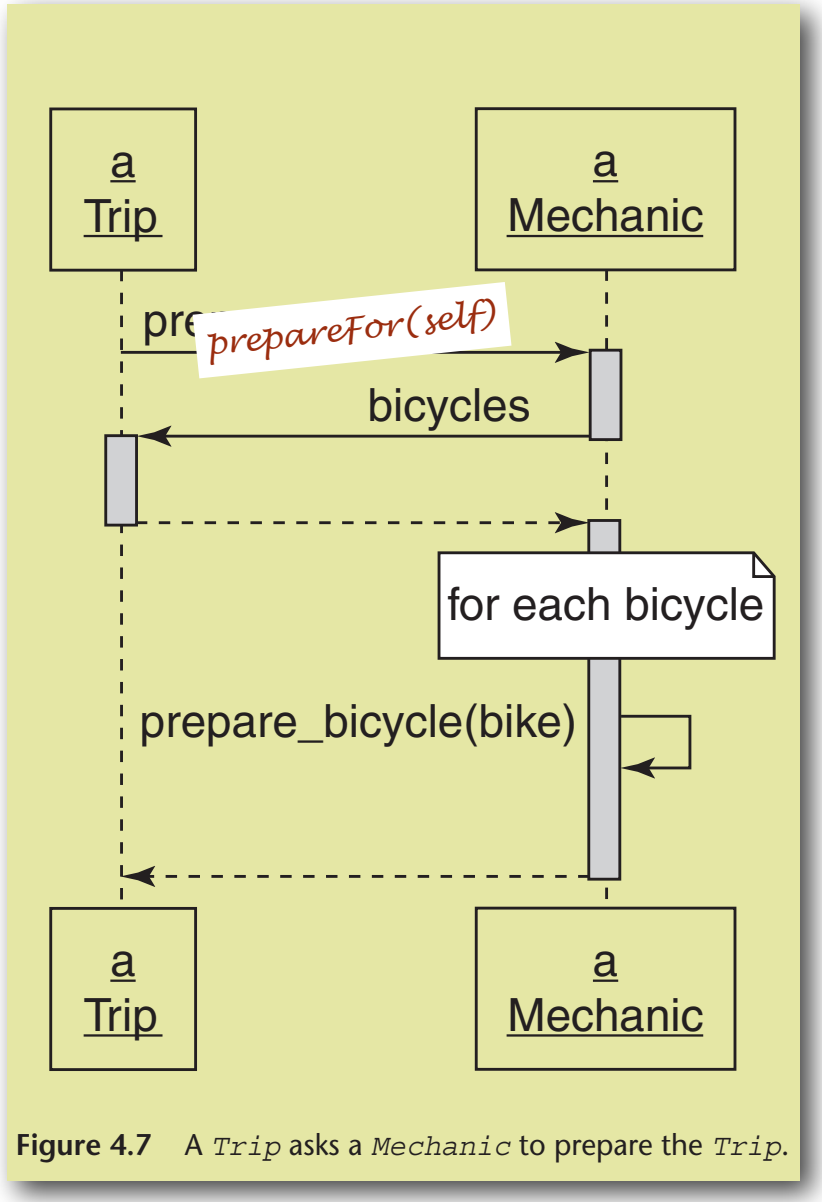


Figure 4.7 A *Trip* asks a *Mechanic* to prepare the *Trip*.

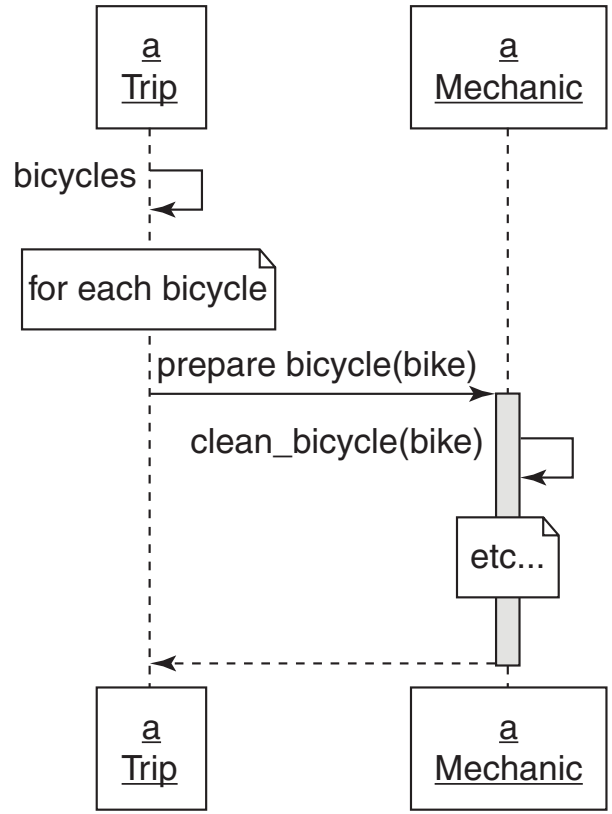


Figure 4.6 A *Trip* asks a *Mechanic* to prepare each *Bicycle*.

Figure 4.7 is far more object-oriented. In this example, `Trip` doesn't know or care that it has a `Mechanic` and it doesn't have any idea what the `Mechanic` will do. `Trip` merely holds onto an object to which it will send `prepare_trip`; it trusts the receiver of this message to behave appropriately.

Figure 4.7 is far more object-oriented. In this example, `Trip` doesn't know or care that it has a `Mechanic` and it doesn't have any idea what the `Mechanic` will do. `Trip` merely holds onto an object to which it will send `prepare_trip`; it trusts the receiver of this message to behave appropriately.

This pattern allows you to add newly-introduced preparers to `Trip` without changing any of its code. You can extend `Trip` without modifying it.

Figure 4.7 is far more object-oriented. In this example, `Trip` doesn't know or care that it has a `Mechanic` and it doesn't have any idea what the `Mechanic` will do. `Trip` merely holds onto an object to which it will send *prepareFor(self)*; it trusts the receiver of this message to behave appropriately.

This pattern allows you to add newly-introduced preparers to `Trip` without changing any of its code. You can extend `Trip` without modifying it.

Figure 4.7 is far more object-oriented. In this example, `Trip` doesn't know or care that it has a `Mechanic` and it doesn't have any idea what the `Mechanic` will do. `Trip` merely holds onto an object to which it will send *prepareFor(self)*; it trusts the receiver of this message to behave appropriately.

This pattern allows you to add newly-introduced preparers to `Trip` without changing any of its code. You can extend `Trip` without modifying it.

The **open/closed principle** states that
software entities should be open for extension, but closed for modification

If objects were human and could describe their own relationships, in Figure 4.5 `Trip` would be telling `Mechanic`: "I know what I want and I know how you do it;" in Figure 4.6: "I know what I want and I know what you do" and in Figure 4.7: "I know what I want and *I trust you to do your part.*"

If objects were human and could describe their own relationships, in Figure 4.5 `Trip` would be telling `Mechanic`: "I know what I want and I know how you do it;" in Figure 4.6: "I know what I want and I know what you do" and in Figure 4.7: "I know what I want and *I trust you to do your part.*"

This blind trust is a keystone of object-oriented design. It allows objects to collaborate without binding themselves to context and is necessary in any application that expects to grow and change.

Forwarding ≠ Delegation

- *Forwarding* a request means asking some other object *assist* to handle the request on your behalf
  - *assist* answers to you, you answer to the original requestor
- *Delegation* means asking another object, *delegate*, to act as *if it were you*
  - *delegate* answers to the original requestor
  - self-request by *delegate* are treated *as if made by you*
- Metz should be arguing to use *forwarding* to avoid train wrecks

Grace note:

- In Grace, methods are by default *public*, and fields are by default *confidential*
- *confidential* means accessible to the object itself, and to the objects that inherit from it.
- you can change the defaults using annotations, *e.g.:*
  - **def** extent **is** *public* = 200@200
    var partner **is** *readable* := nobody
- vars can be readable or writable, as well as fully public
- methods that are not in your interface should be annotated as *confidential*

Types as Interfaces

Types as Interfaces

Grace note:

Types as Interfaces

Grace note:

- you can define a type as an interface

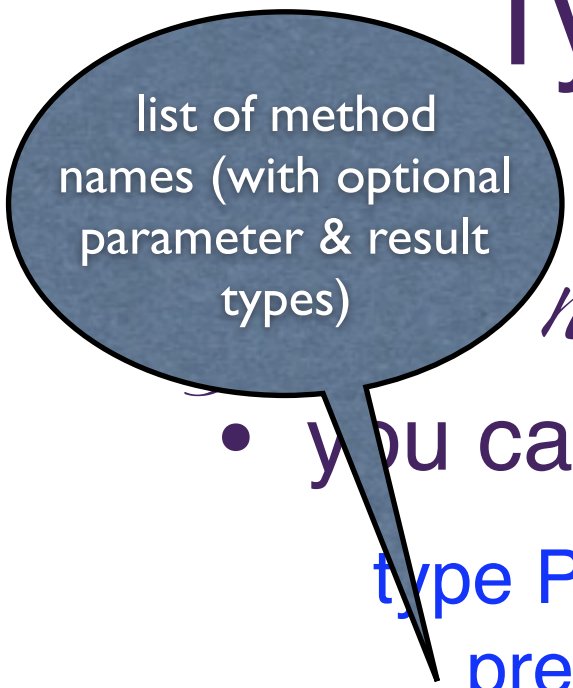
Types as Interfaces

Grace note:

- you can define a type as an interface

```
type Preparer = interface {
    prepareFor(aTrip) → Done
    ...
}
```

Types as Interfaces



note:

- you can define a type as an interface

```
type Preparer = interface {
    prepareFor(aTrip) → Done
    ...
}
```

Types as Interfaces

Grace note:

- you can define a type as an interface

```
type Preparer = interface {
    prepareFor(aTrip) → Done
    ...
}
```

- and use it to check your class:

Types as Interfaces

Grace note:

- you can define a type as an interface

```
type Preparer = interface {
    prepareFor(aTrip) → Done
    ...
}
```

- and use it to check your class:

```
class mechanic →Preparer { ... }
```

Types as Interfaces

Grace note:

- you can define a type as an interface

```
type Preparer = interface {
    prepareFor(aTrip) → Done
    ...
}
```

- and use it to check your class:

```
class mechanic→Preparer{ ... }
```

Types as Interfaces

Grace note:

- you can define a type as an interface

```
type Preparer = interface {
    prepareFor(aTrip) → Done
    ...
}
```

- and use it to check your class:

```
class mechanic→Preparer{ ... }
```

will raise a type error (at runtime) if the new object does not have the right methods

Types as Interfaces

Grace note:

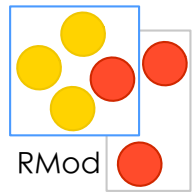
- you can define a type as an interface

```
type Preparer = interface {
    prepareFor(aTrip) → Done
    ...
}
```

- and use it to check your class:

```
class mechanic →Preparer { ... }
```

The Law of Demeter



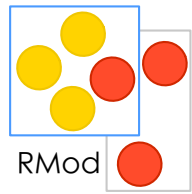
You should send requests only on:

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

Avoid global variables

Avoid objects returned from messages sent to others

Correct Requests

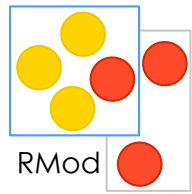



```
method someMethod(aParameter) {
        self.foo
        field.foo
        aParameter.foo
        def myThing := aThing.createNewObject
        myThing.foo
}
```

Talk only to your immediate friends

- Friends of friends are suspect

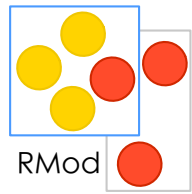
In other words …



- You can play with yourself. this.method(_)
- You can play with your own toys (but you can't take them apart). field.method(_)
- You can play with toys that were given to you. parameter.method(_)
- And you can play with toys you've made yourself.

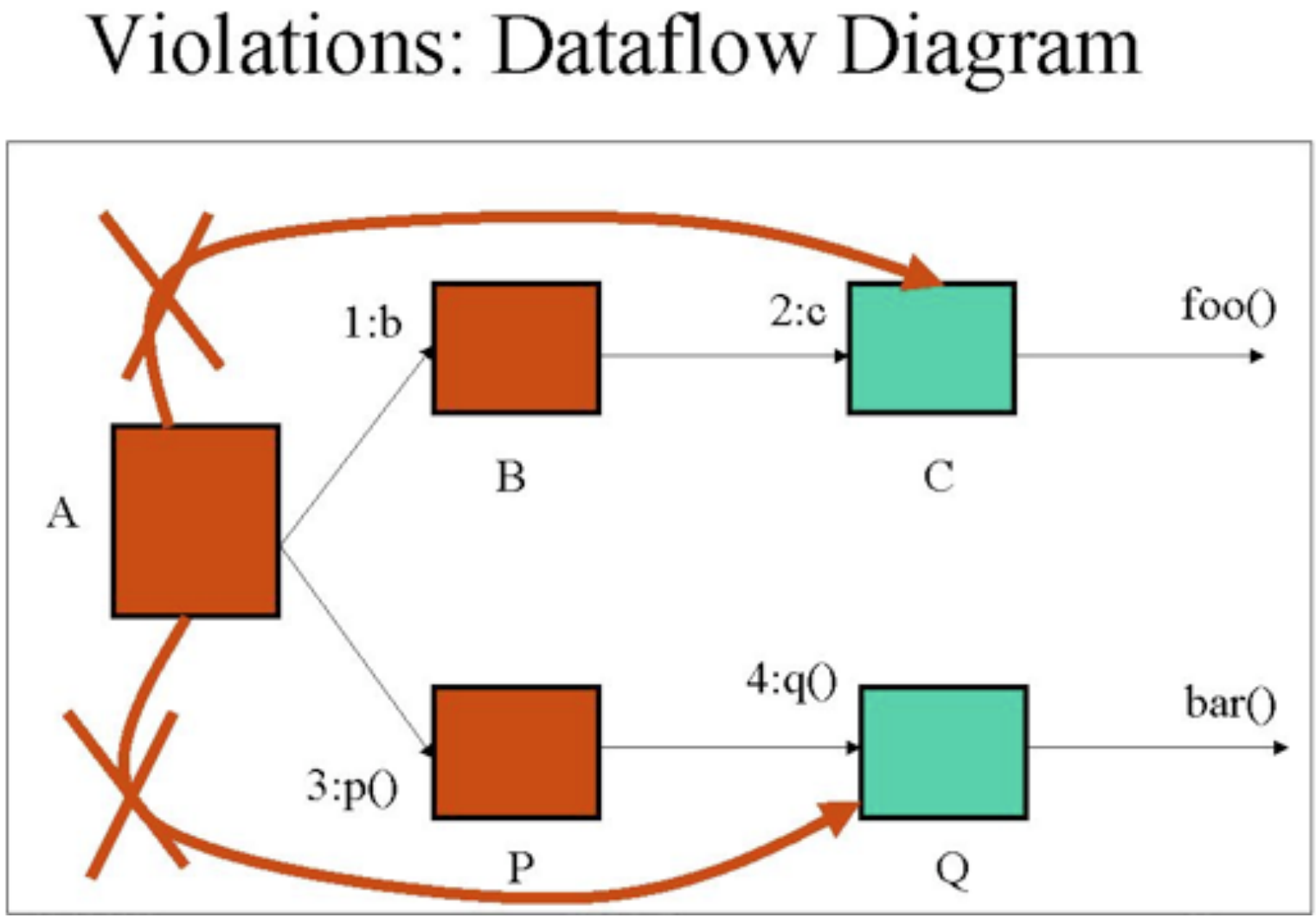
  myA = aFactory.makeA(_); myA.method()

Halt!



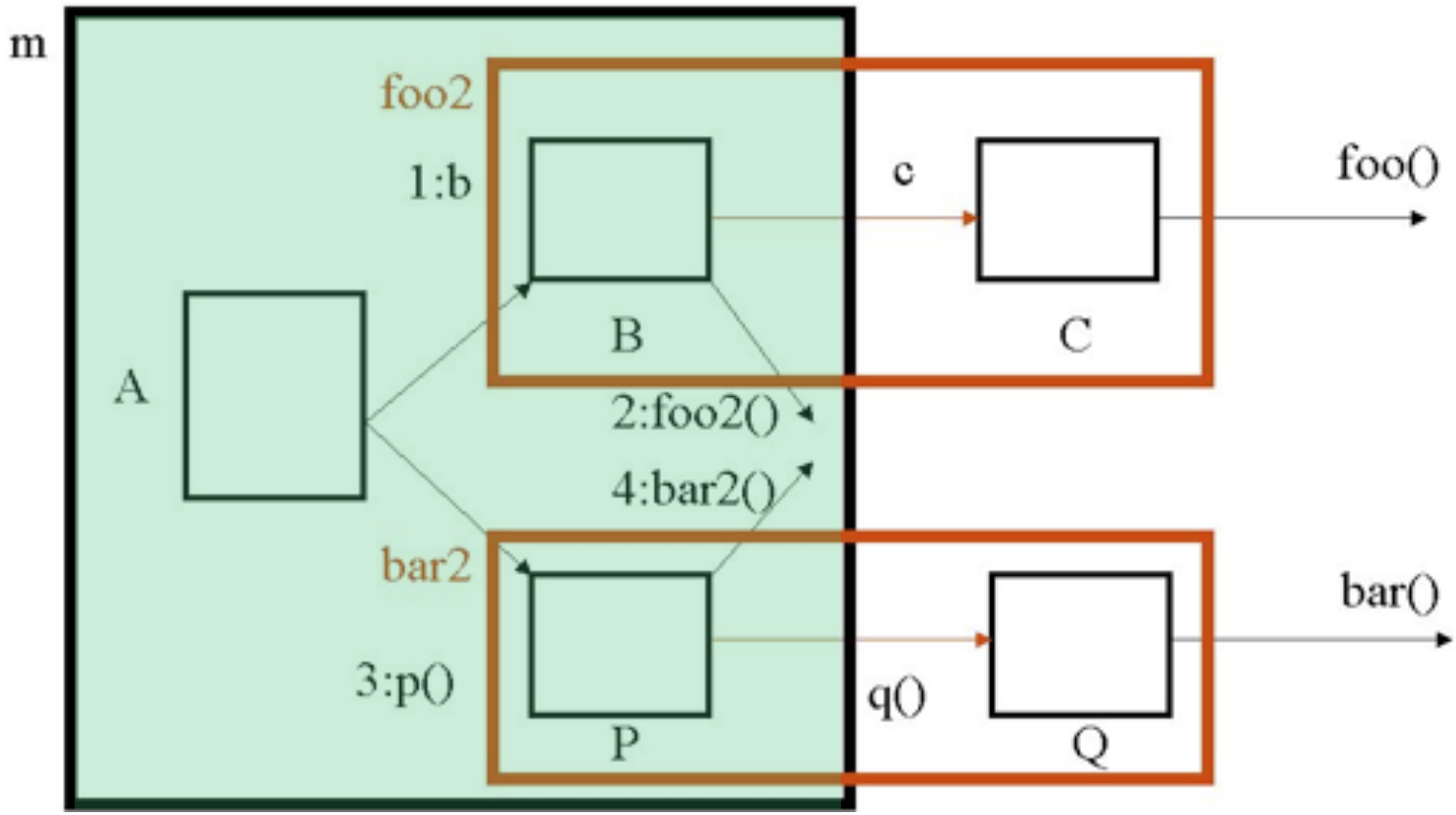

```
class A {public: void m(); P p(); B b; };
class B {public: C c; };
class C {public: void foo(); };
class P {public: Q q(); };
class Q {public: void bar(); };
void A::m() {
    this.b.c.foo(); this.p().q().bar();}
```

Do not skip intermediaries!

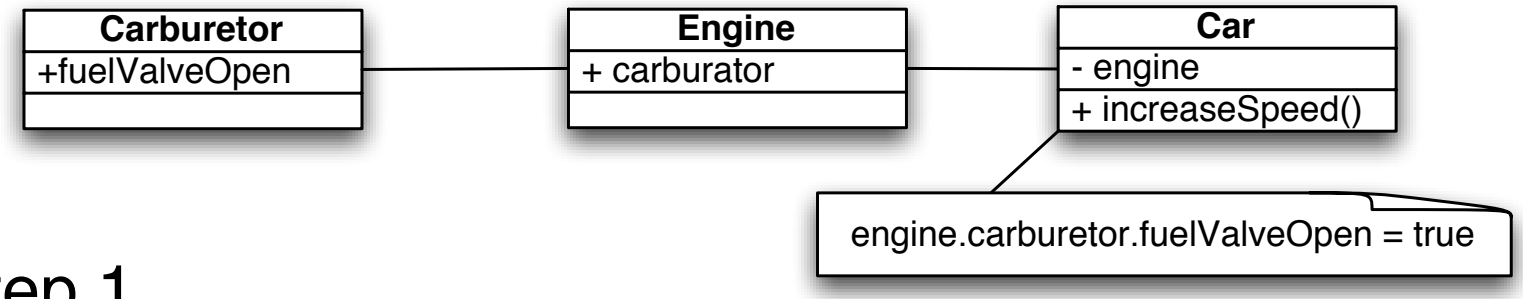


Solution

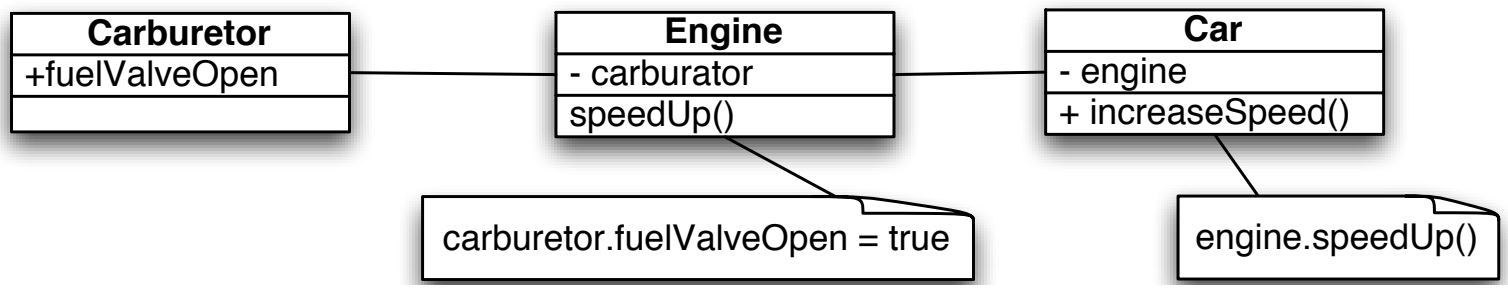
- Follow the Law of Demeter



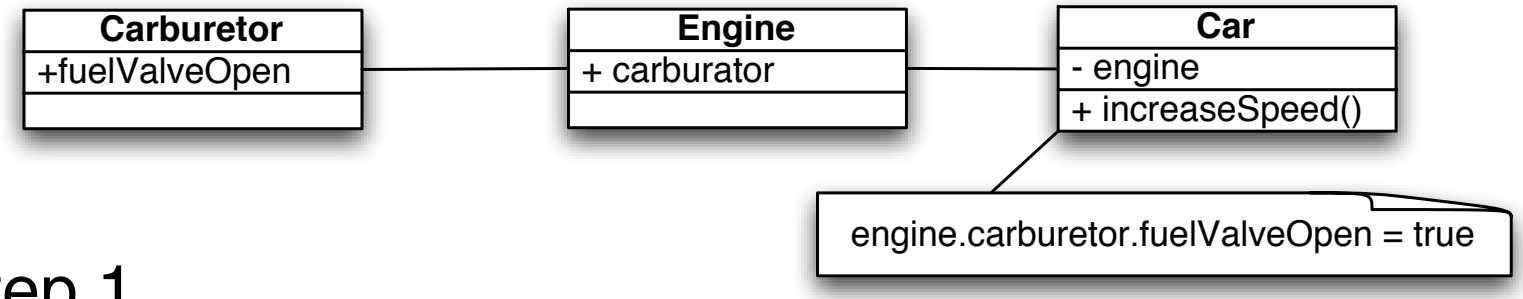
Transformation



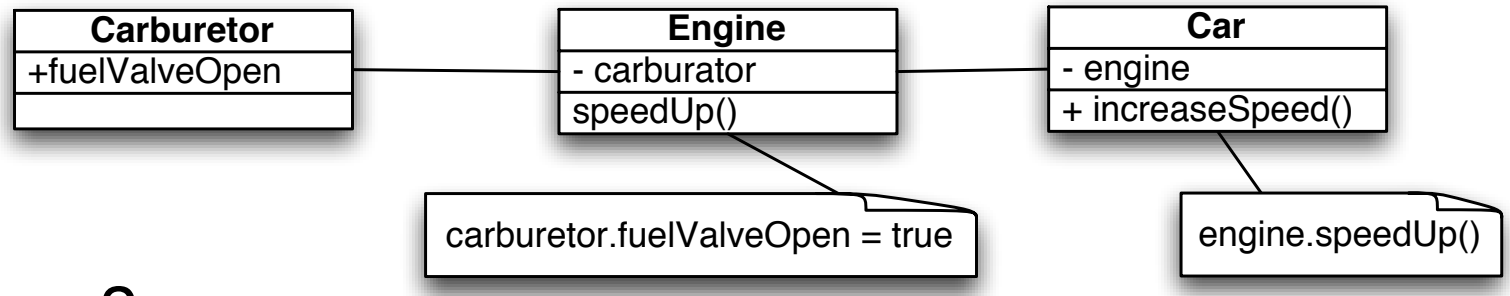
Step 1



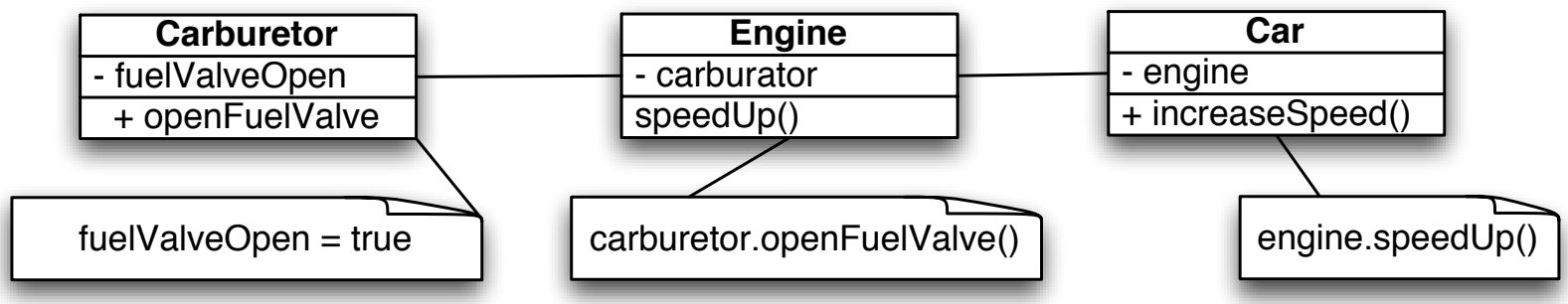
Transformation



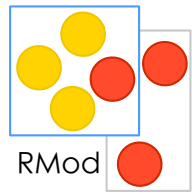
Step 1



Step 2

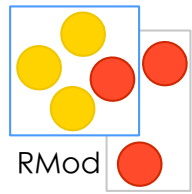


The Dark side of the Law of Demeter




```
class bicycleFleet {
        def bikes = set [ ]
        method do(aBlock) { bikes.do(aBlock) }
        method isEmpty { bikes.isEmpty }
        method map(aBlock) { bikes.map(aBlock) }
        method filter(aBlock) { bikes.filter(aBlock) }
            …
}
```

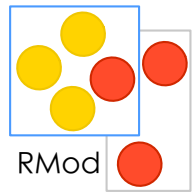
The Dark side of the Law of Demeter




```
class bicycleFleet {
        def bikes = set [ ]
        method do(aBlock) { bikes.do(aBlock) }
        method isEmpty { bikes.isEmpty }
        method map(aBlock) { bikes.map(aBlock) }
        method filter(aBlock) { bikes.filter(aBlock) }
              …
}
```

Each object itself has to provide a complete interface

About Accessor methods



In Grace, we don't have to worry about this.

Accessor methods are indistinguishable from direct field access.

Other languages distinguish.

Some gurus say: “Access all instance variables using accessor methods”. Why?

I say:

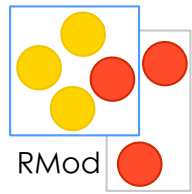
Be consistent inside an object:

do not mix direct access and accessor methods

Initially, make accessors *confidential* or *private*

Make them more *public* as part of designing the interface.

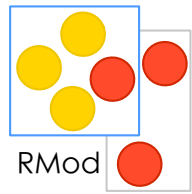
Example




```
def scheduler = object {
        def tasks = priorityQueue
        method suspendedTasks { tasks }
        …
}
```

But now everybody can tweak the task queue!

Accessors

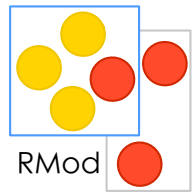


Accessors are good for lazy initialization

```
def suspendedTasks = uninitialized
method tasks {
    if (uninitialized == suspendedTasks) then {
        suspendedTasks := priorityQueue
    }
    suspendedTasks
}
```

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

Provide a Complete Interface




```
class workstation {
    method accept(aPacket) {
        if (aPacket.addressee == self.address) then { … } else { … }
        …
```

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

```
class packet {
    method isAddressedTo(aNode) { addressee == aNode.address }
    …
}
class workstation {
    method accept(aPacket) {
        if (aPacket.isAddressedTo(self)) then { … } else { … }
        …
```

Who is Demeter Anyway?

Who is Demeter Anyway?

Demeter,
Consort of Zeus,
mother of Persephone

Who is Demeter Anyway?

Demeter,
Consort of Zeus,
mother of Persephone

Karl Liebherr,
Professor of Computer Science at Northeastern University