Sharing Behaviour without Inheritance

Based on Metz Chapter 7

Portland State
UNIVERSITY

Why?

- Single inheritance can be used for classification in only one dimension
- Often, we want objects to play multiple roles
- Traits let us implement the role behavior once, reuse it in many places

When to use it?

- When your language supports it!
  - Java interfaces can now include default implementation code — like traits
- Roles often come in pairs
  - Preparable & Preparer, Observable & Observer
  - Sometimes there is no useful code to share
    - iterator in the collections framework

Metz Example

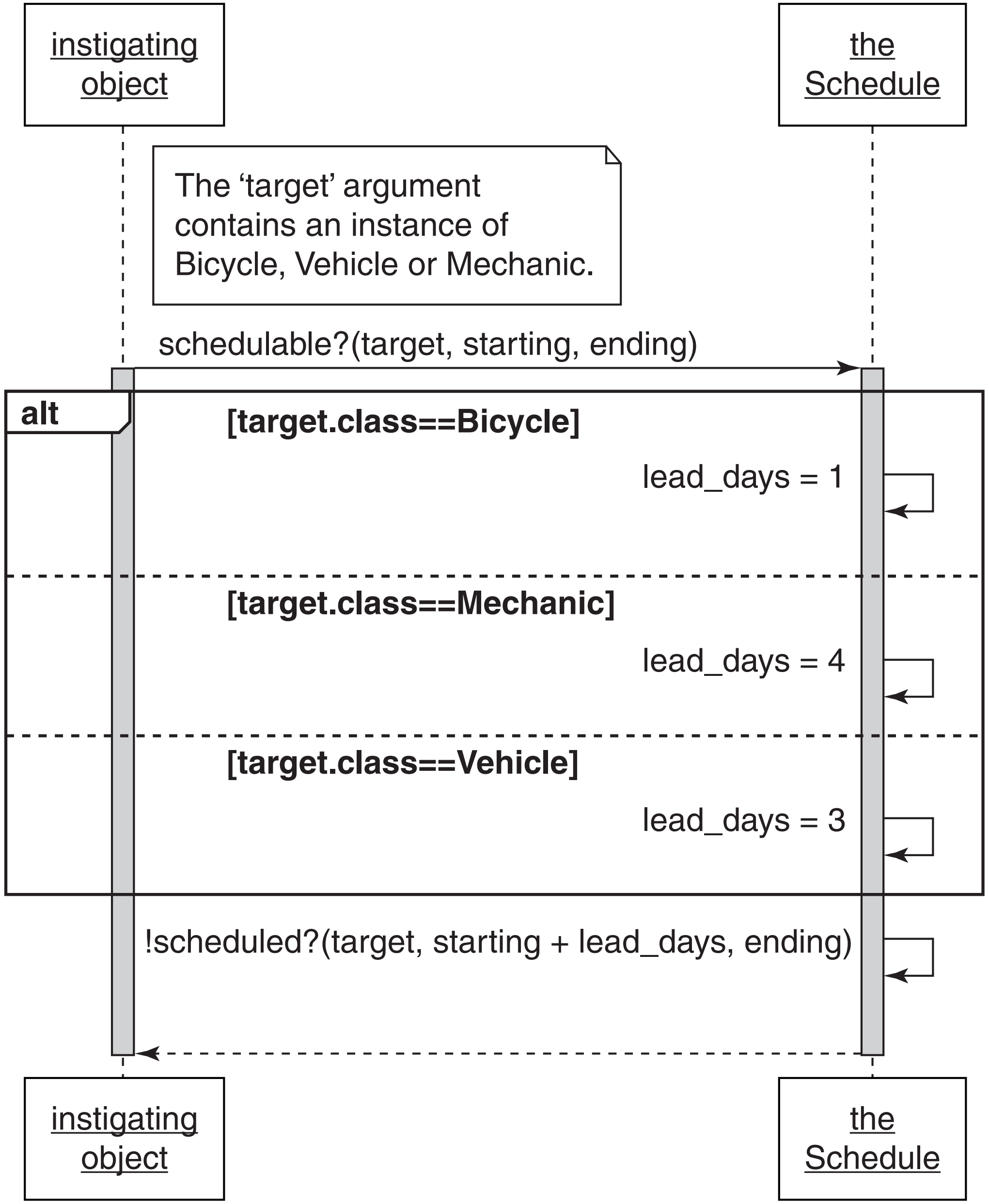


Figure 7.1 The schedule knows the lead time for other objects.

Portland State UNIVERSITY

Better:

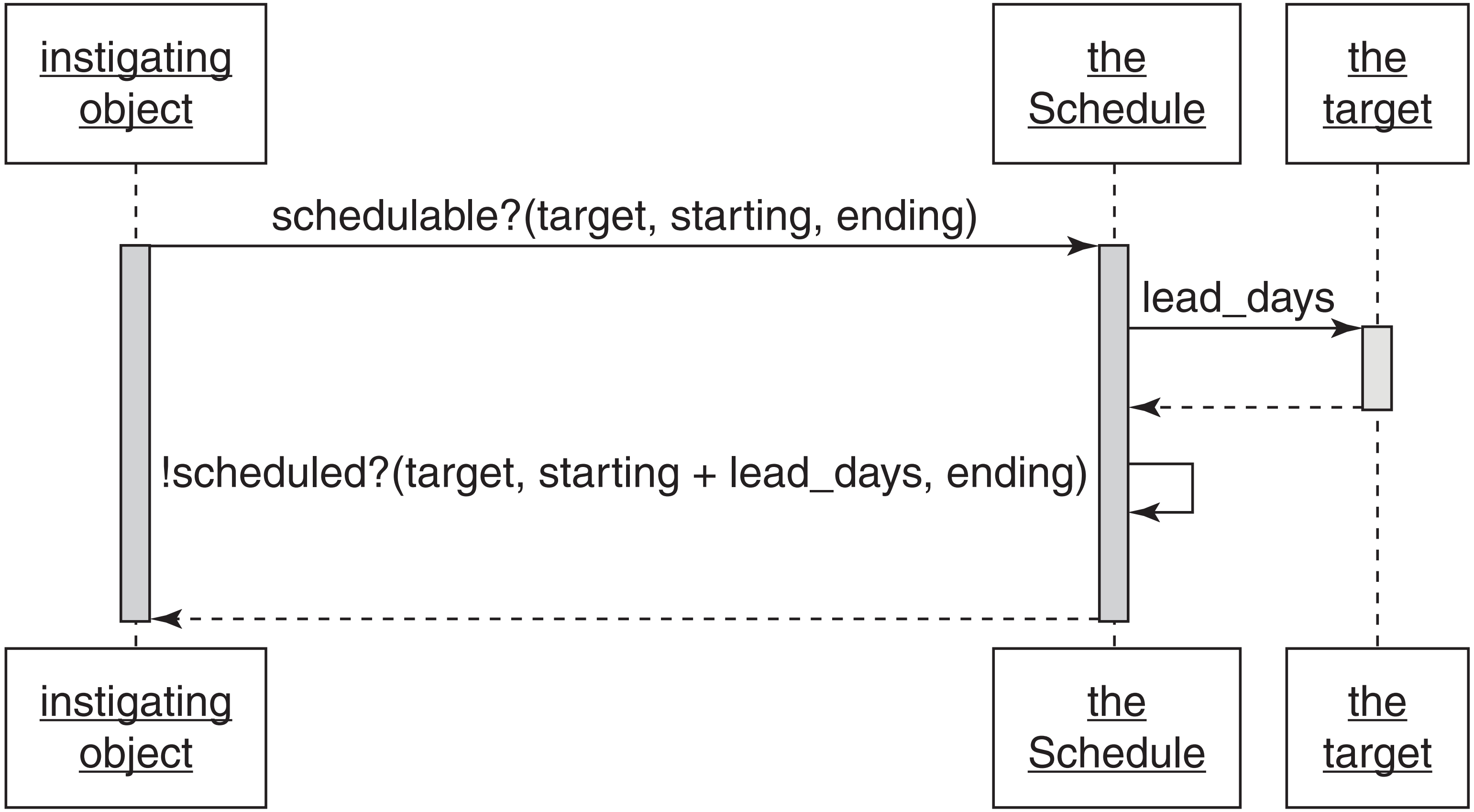


Figure 7.2 The schedule expects targets to know their own lead time.

Better:

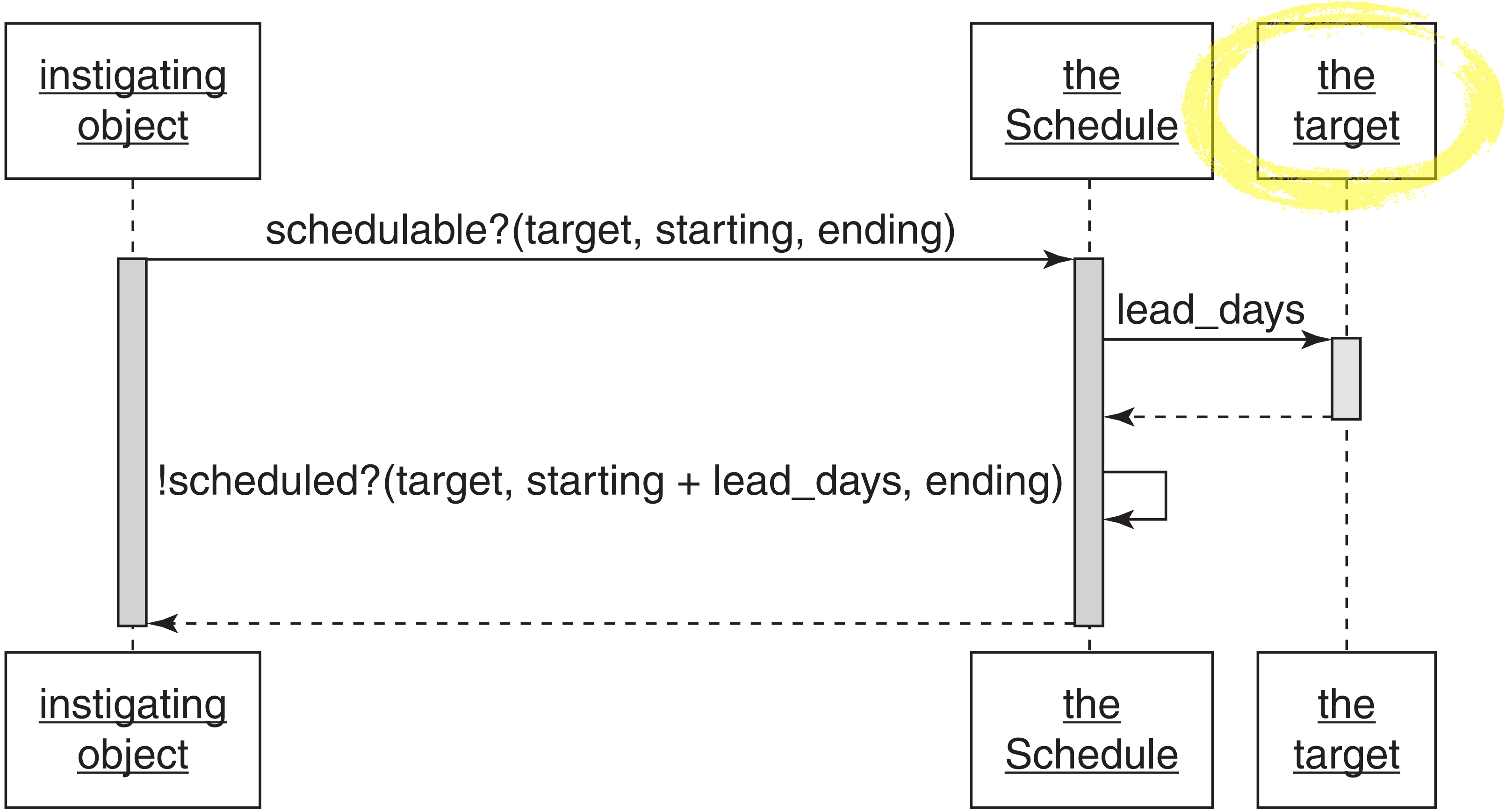


Figure 7.2 The schedule expects targets to know their own lead time.

Minimize dependencies

- objects should manage themselves; they should contain their own behavior
- If your interest is in object B, you should not be forced to know about object A if your only use of it is to find out things about B.

Why ask schedule about target?

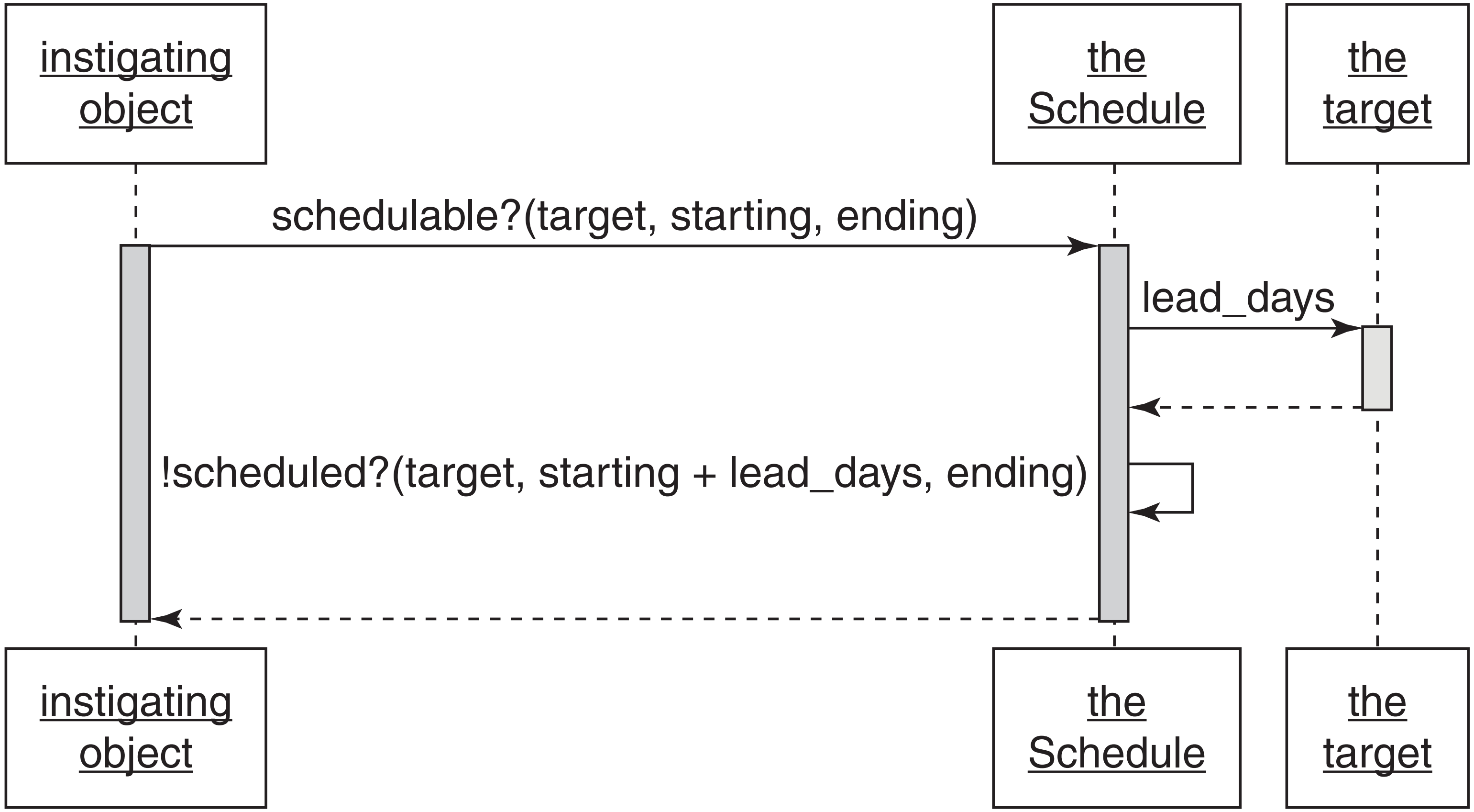


Figure 7.2 The schedule expects targets to know their own lead time.

Why ask schedule about target?

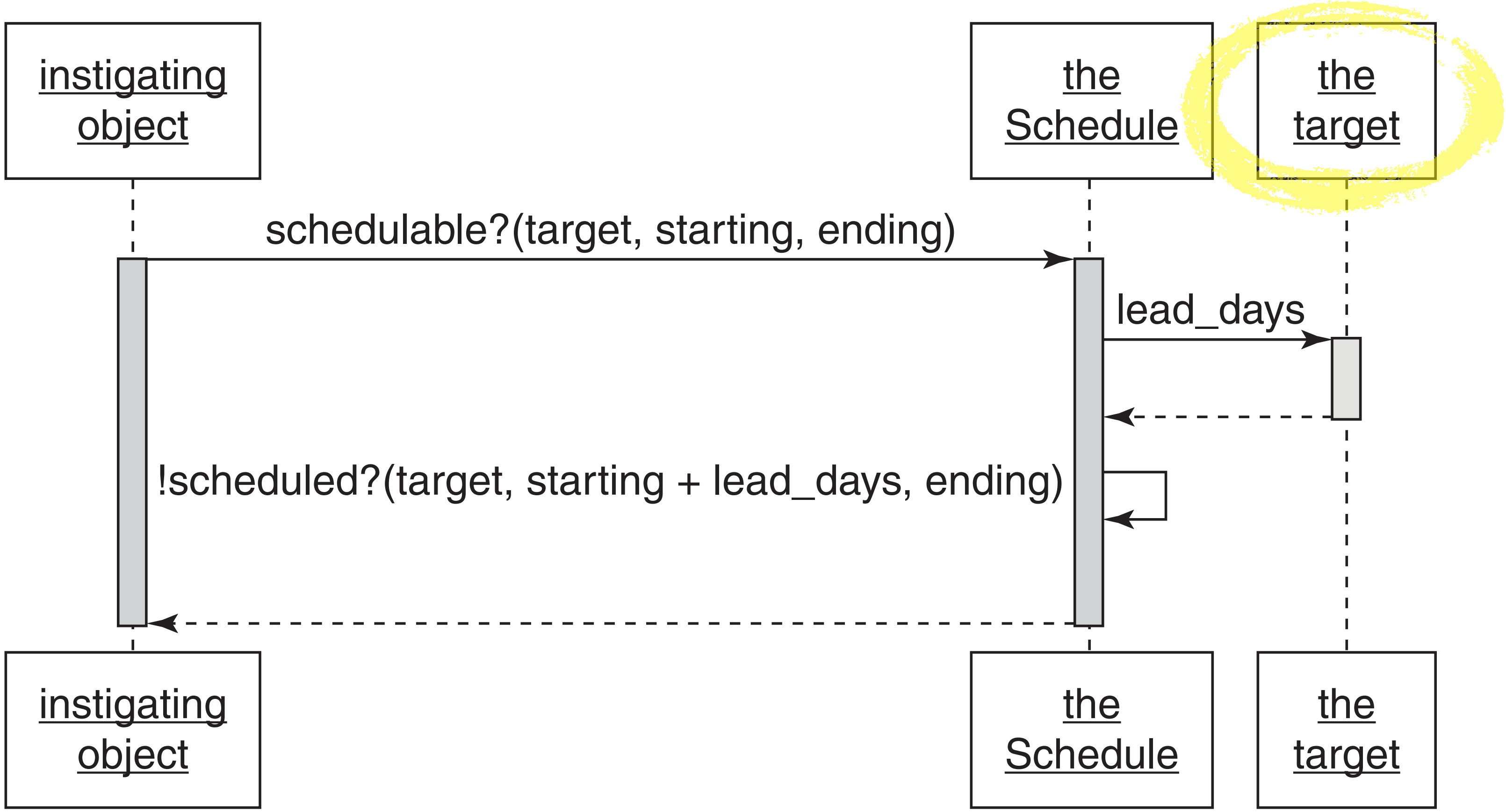


Figure 7.2 The schedule expects targets to know their own lead time.

How to implement traits

- Make the code concrete first
- Make it run green
- *Then* refactor into a trait.
- Why?

Like inheritance…

- Traits can have hook methods, and abstract methods ...
- What about super?
  - depends ...
  - Ruby modules don't change the superclass

Beware!

- Without good tools, understanding code written with traits can be a scary experience.
- The usefulness and maintainability of reuse hierarchies (whether using traits or superclasses) is in direct proportion to the quality of the code.

Insist on the Abstraction

- When an object checks the class of a receiving object to determine what message to send, you have overlooked a “duck type”, a.k.a. an interface
- Define that type!
  - Give its methods intention-revealing names
  - Figure out which objects should implement them

Insist on the Abstraction

Metz says:

- *All* of the code in an abstract superclass should apply to *every* class that inherits it.
- If you cannot correctly identify the abstraction there may not be one!
- If no common abstraction exists then (neither) inheritance (nor trait use) is the solution to your design problem.

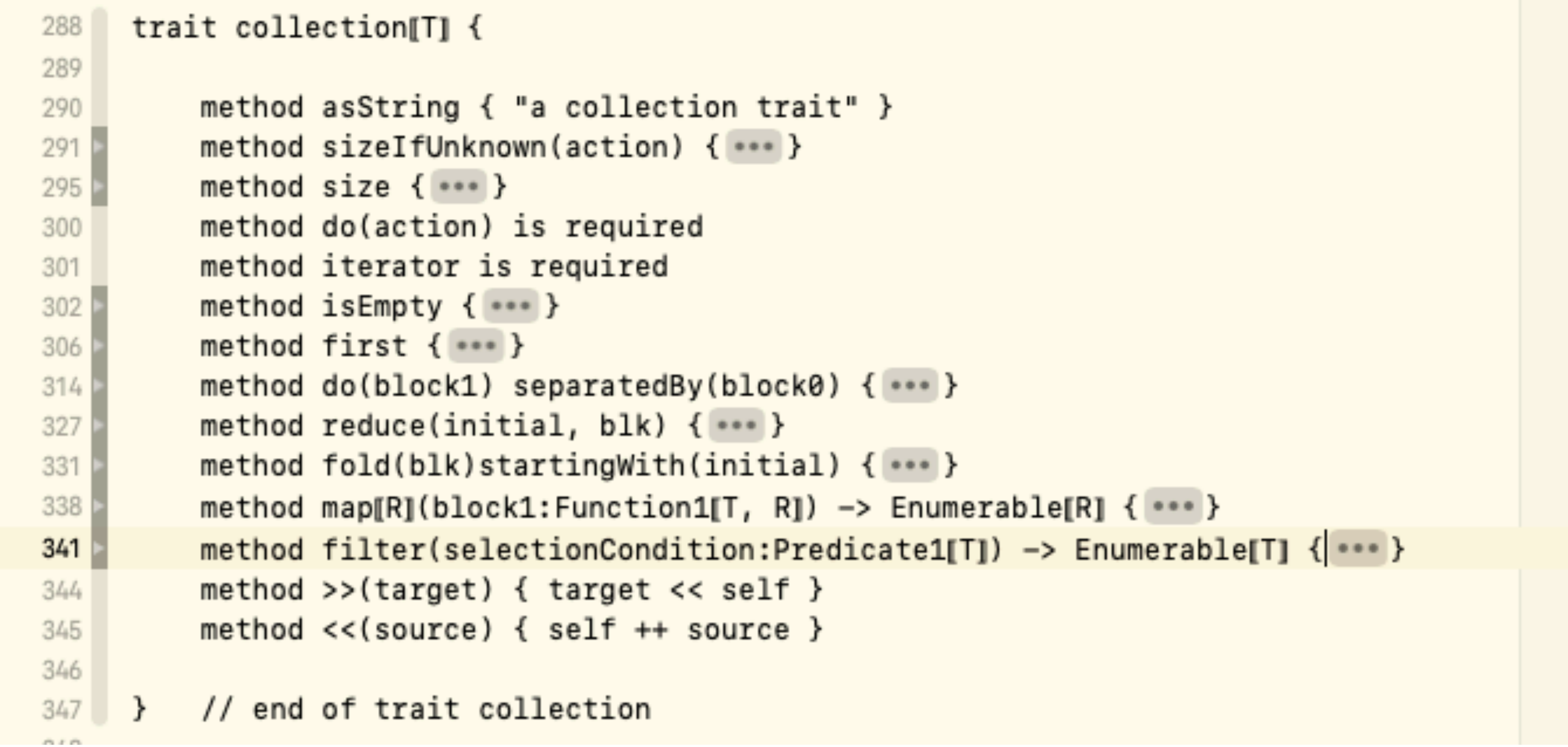
Well, Maybe ...

- I'm not sure of the degree to which I believe that
- Consider:

```
trait emptiness {
    method size is required
    method isEmpty { size == 0 }
    method isNotEmpty { isEmpty.not }
}
```

- Is it useful to factor-out this code?
  - is there an underlying abstraction.?

What about this?



```
trait collection⟦T⟧ {

    method asString { "a collection trait" }
    method sizeIfUnknown(action) {•••}
    method size {•••}
    method do(action) is required
    method iterator is required
    method isEmpty {•••}
    method first {•••}
    method do(block1) separatedBy(block0) {•••}
    method reduce(initial, blk) {•••}
    method fold(blk)startingWith(initial) {•••}
    method map⟦R⟧(block1:Function1⟦T, R⟧) -> Enumerable⟦R⟧ {•••}
    method filter(selectionCondition:Predicate1⟦T⟧) -> Enumerable⟦T⟧ {•••}
    method >>(target) { target << self }
    method <<(source) { self ++ source }

}   // end of trait collection
```

Use the Template Pattern

- The fundamental coding technique for creating inheritable code is the template method.
- This pattern is what allows you to separate the abstract from the concrete.
- The template's requests represent the parts of the algorithm that vary. This forces you to make explicit decisions about what varies and what does not.

Create Shallow Hierarchies

- Easy to understand
  - an object depends on *all* of its ancestors.
- Metz’s template hook pattern works only for one level
  - more than 1 level => back to depending on super