Reducing Costs with Structural Typing

Portland State UNIVERSITY

“Duck Typing”

In computer programming with object-oriented programming languages, duck typing is a layer of programming language and design rules on top of typing. Typing is concerned with vacuous! assigning a type to any object. unnecessary! Duck typing is concerned with establishing the suitability of an object for some purpose. With class typing, suitability is assumed to be determined by an object’s class only.

With nominal typing, suitability is assumed to depend on the claims made when the object was created

In contrast in duck typing, an object's suitability is determined by the presence of certain methods and properties (with appropriate meaning), rather than the actual type of the object.

The name of the concept refers to the duck test, attribut James Whitcomb Riley, which may be phrased as follows:

> When I see a bird that wa like a duck and quacks like a duck, I call that bird

Some confusion present here

Metz:

- Duck types are public interfaces that are not tied to any specific class
- A *Grace* object is like a partygoer at a masquerade ball that changes masks to suit the theme. It can expose a different face to every viewer; it can implement many different interfaces.
- … an object's type is in the eye of the beholder. Users of an object need not, and should not, be concerned about its class.
- It's not what an object is that matters, it's what it does.

Structural *vs.* Nominal

- Structural typing (Duck Typing)
  - a type T describes a set of properties (so, it's like a predicate). An object o has a type T if it satisfies that predicate.
  - Grace has Structural typing.
- Nominal Typing (Class Typing +)
  - a type T is identified with both a set of properties and a name.
  - An object o has a type T if it was made by a class that says that its objects have type T. The language is designed in such a way that this *also* means that it must have the properties of T.
  - Even though p might have all the right properties, it's not of type T unless the class that made it says so.

- Static typing:
  - the type of every expression can be determined before the program starts to execute. The programmer is usually (but not always) required to annotate declarations with types.
- Dynamic typing:
  - no attempt is made to determine the type of an expression until the program is executing.
- Gradual Typing:
  - type annotations are optional; when given, they enable the types of some expressions to be evaluated before the programs starts to execute. Others will be checked at runtime.

Explore consequences of *not* using Duck Types

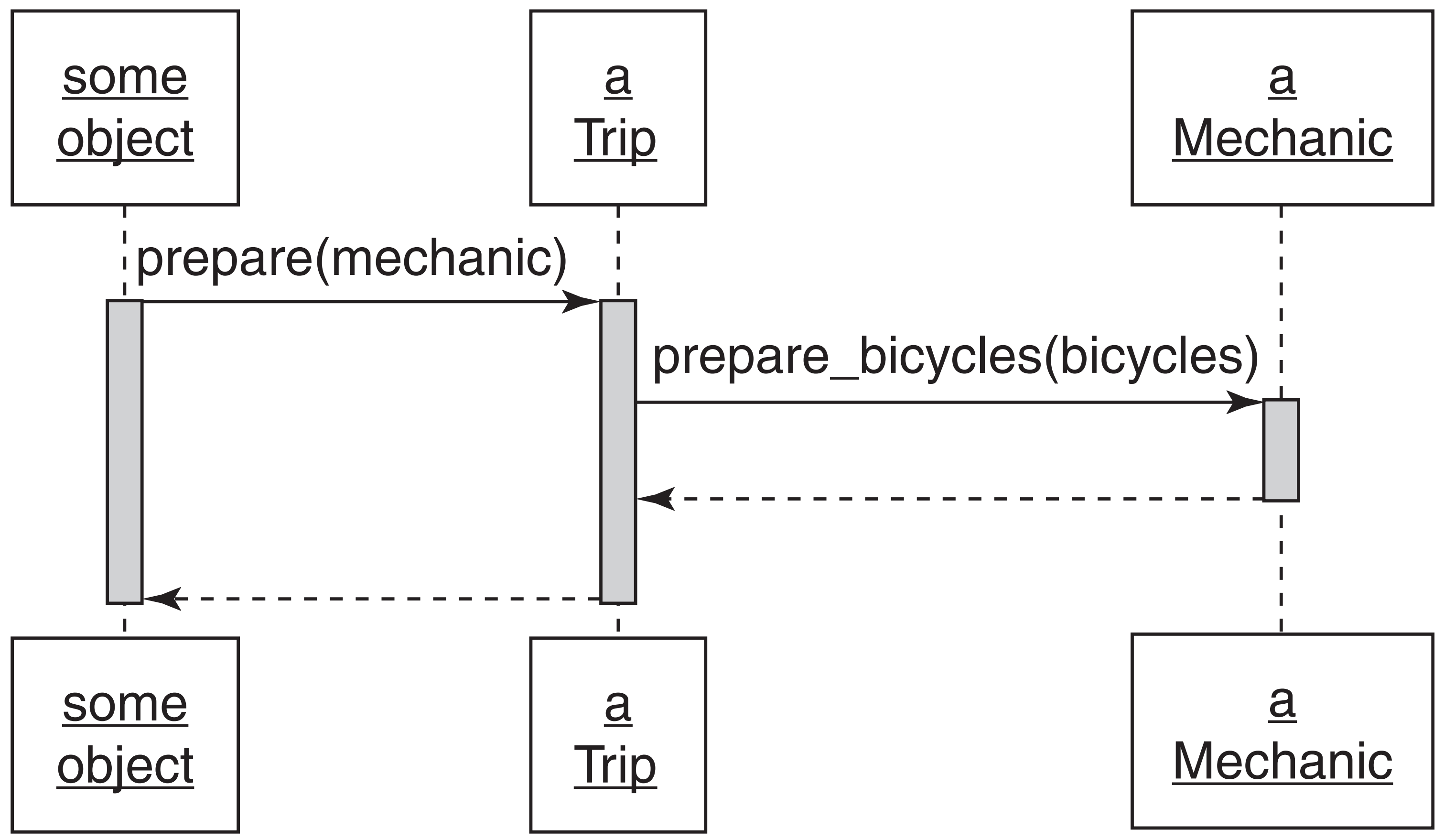


Figure 5.1 Trip prepares itself by asking a mechanic to prepare the bicycles.

- Seems OK, so long as we have *only* mechanics

But when there are *other* preparers…

- type-case,
- instance-of,
- “branding” with strings,

are all equally evil

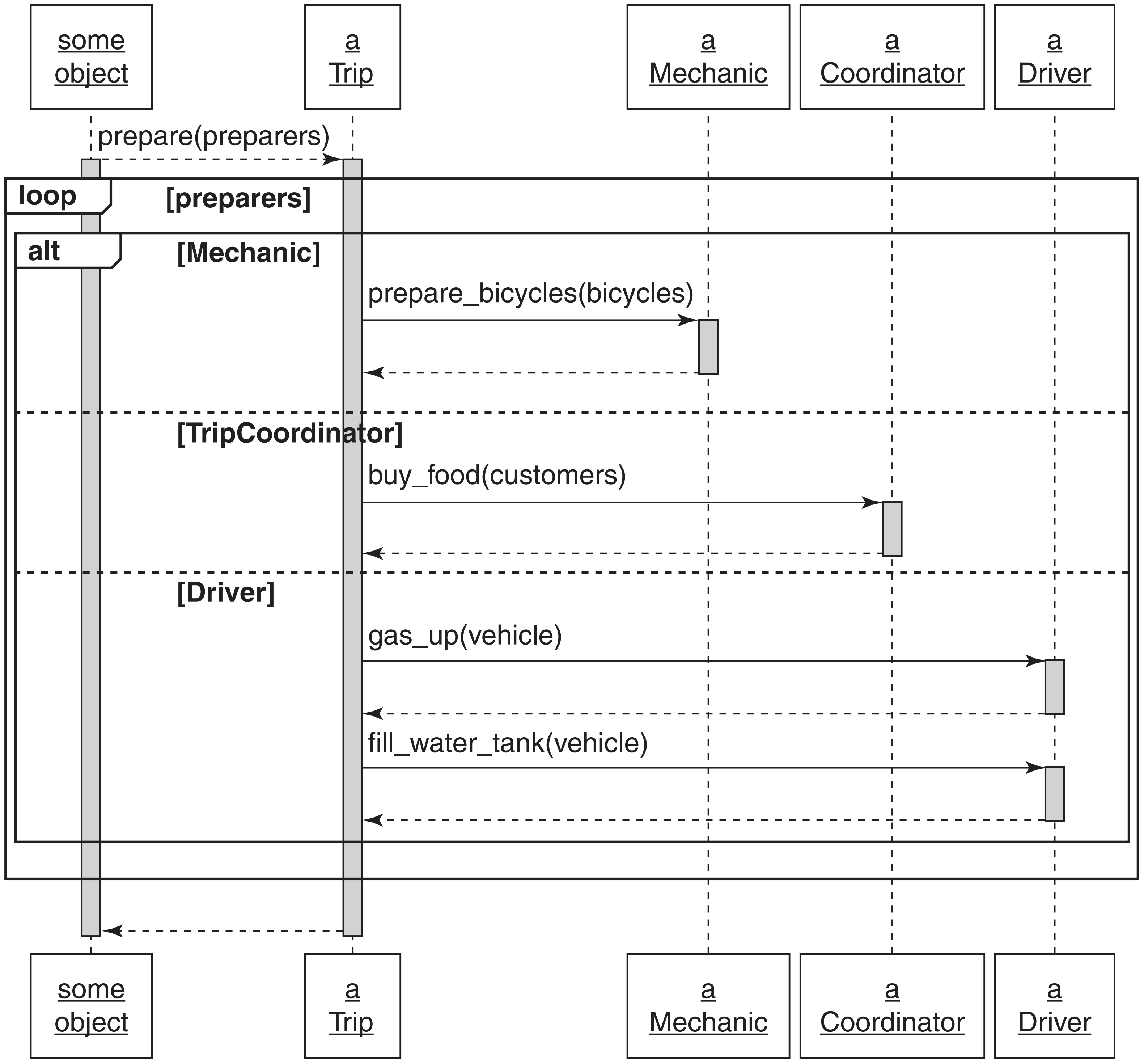


Figure 5.2 Trip knows too many concrete classes and methods.

The Root of the Problem

- "If your design imagination is constrained by class and you find yourself unexpectedly dealing with objects that don't understand the message you are sending, your tendency is to go hunt for messages that these new objects do understand".
- Instead: create new messages that *all* the objects can reasonably understand

What do the targets have in common?

- They all help a trip make preparations

"What kind of thing is `Preparer`? At this point it has no concrete existence; it's an abstraction, an agreement about the public interface on an idea. It's a figment of design."

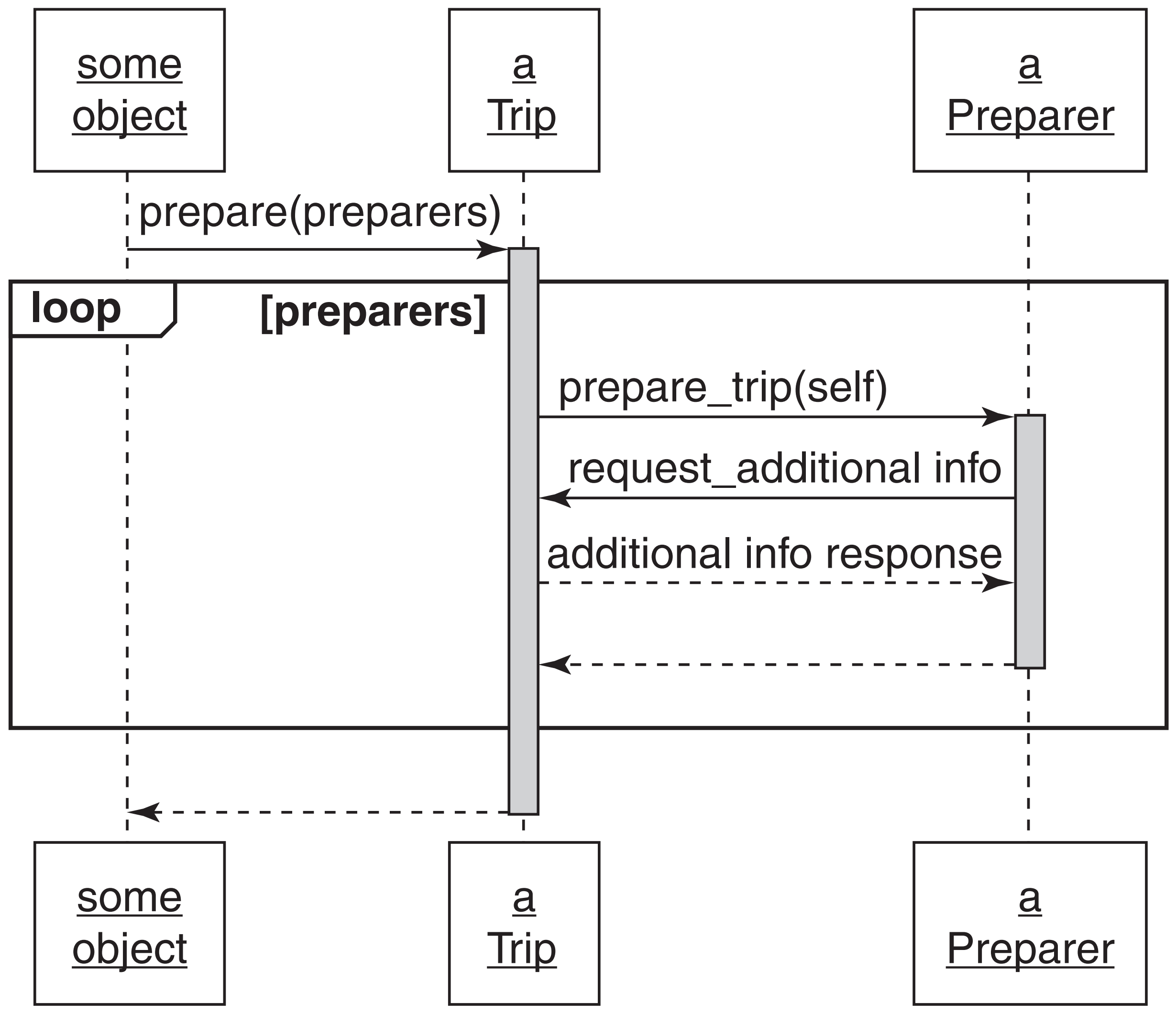


Figure 5.4 Trip collaborates with the preparer duck.

Documenting Duck Types

- “When you create duck types you must both document *and* test their public interfaces.”
- document the existence of the type
- test that certain objects *have that type*

Portland State UNIVERSITY

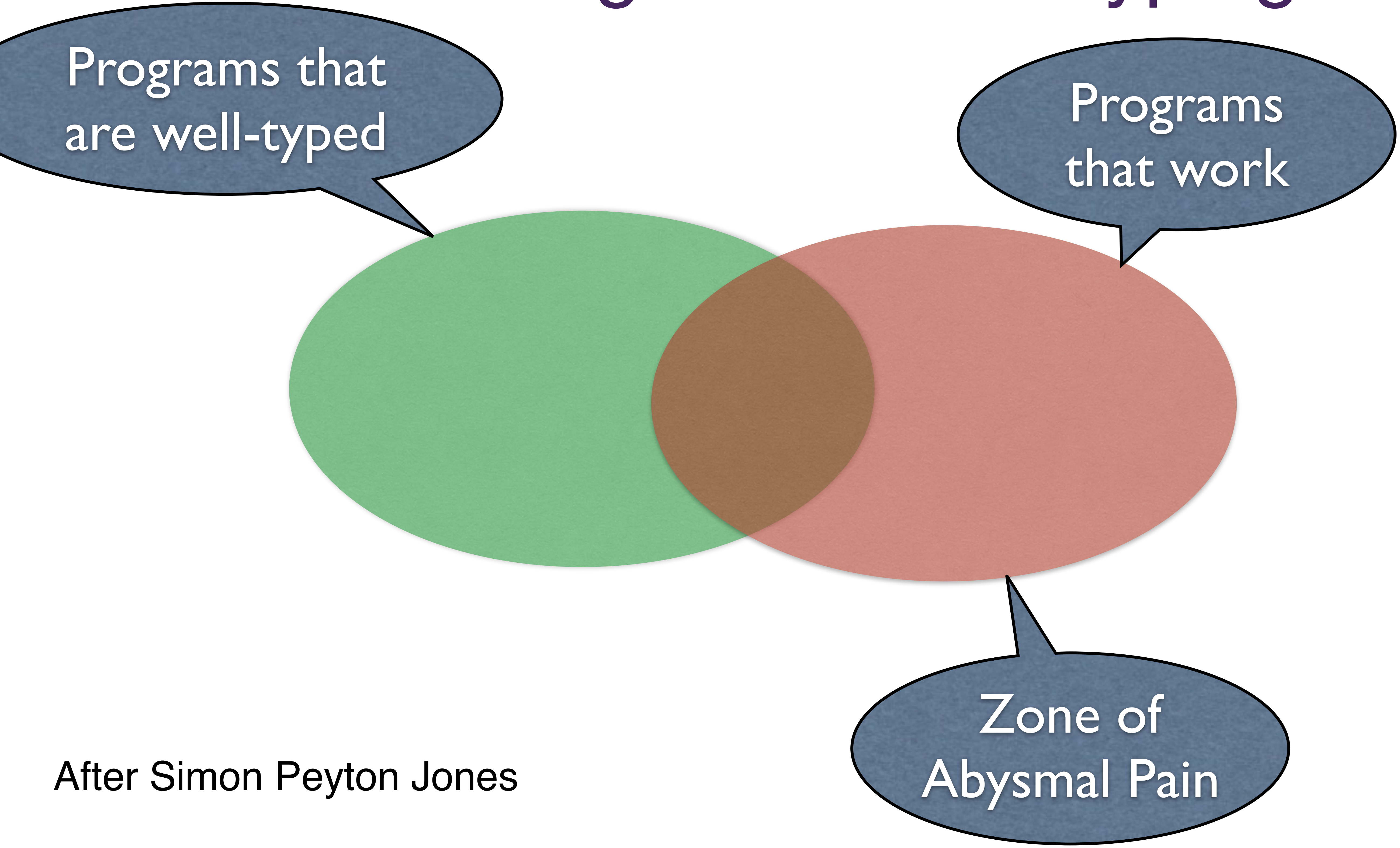
Polymorphism

- Functional programmers: a *function* is (parametrically) polymorphic if it can be applied to arguments of more than one type, and treats them all uniformly
- Metz: a *message* is polymorphic if there are many objects that have a corresponding *method*, and they do related things
- Black: a method is polymorphic if it can be applied to arguments of more than one type. It does this by sending messages that are polymorphic in Metz's sense.

Static typing does *not* prevent Duck typing

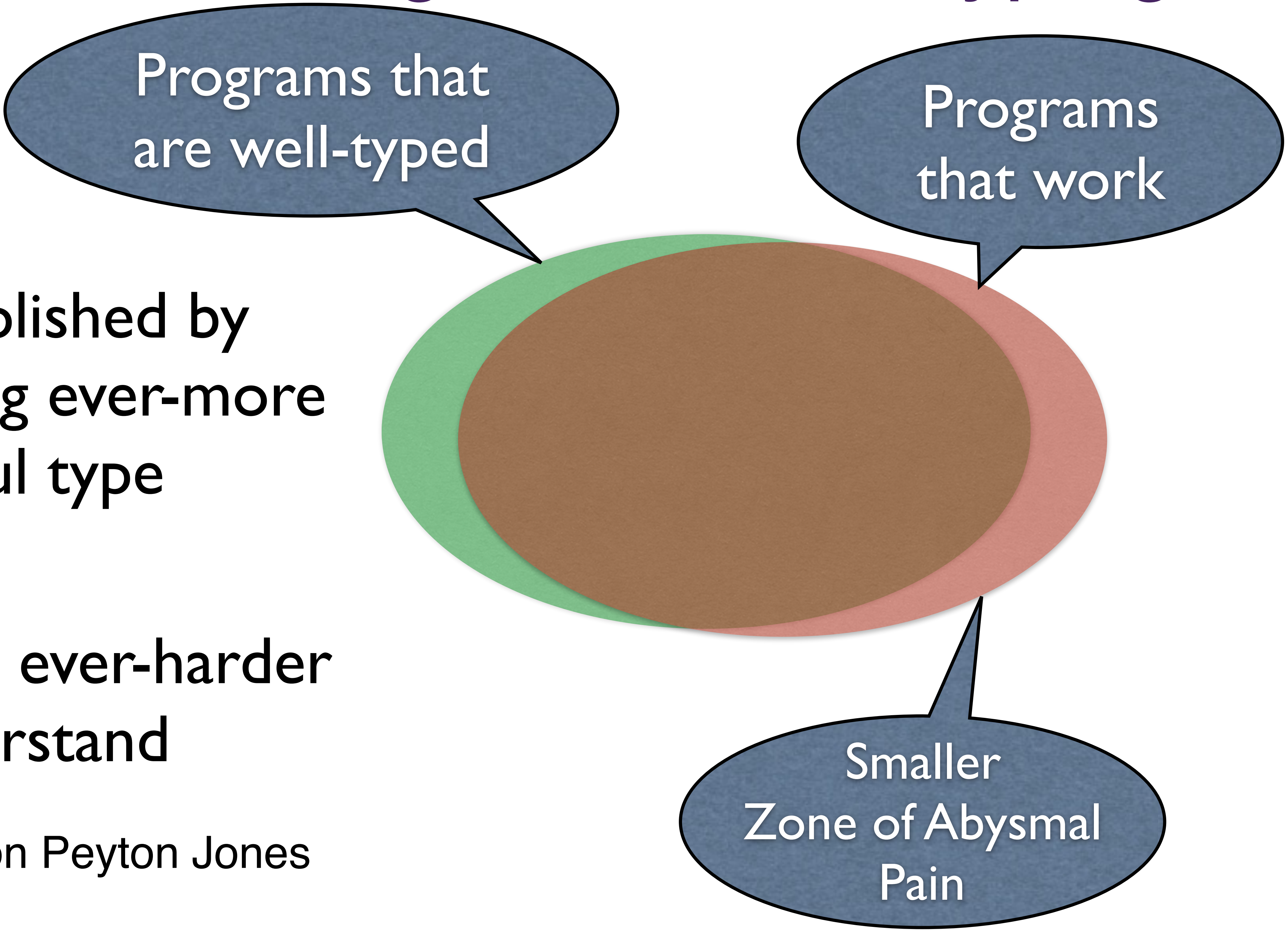
- *Class typing* prevents duck typing
  - Class typing says: I don't care how capable you are, or how good you are at your job …
  - if you don’t come from the right *class*, I won’t even let you *try*
- Even in Java, you *can* do Duck typing if you use *interfaces* as types, not *classes*.

What's Wrong with Static Typing?



After Simon Peyton Jones

What's Wrong with Static Typing?



Accomplished by inventing ever-more powerful type systems

that are ever-harder to understand

After Simon Peyton Jones

Portland State UNIVERSITY