Composition

Based on Metz Chapter 8:
Combining Objects with Composition

Portland State
UNIVERSITY

What is Composition?

- Objects respond to requests
- How?
  - they have their own methods
  - they “pass the buck” to another object: forwarding *to a component*
  - they acquire behavior from another object: delegation

The Gang of Four say:

- The second principle of object-oriented design:
  - *Favor object composition over inheritance*

The Gang of Four say:

- The first principle of object-oriented design:
    - *Program to an interface, not to an implementation*
- The second principle of object-oriented design:
    - *Favor object composition over inheritance*

Inheritance *vs.* Composition

- Inheritance lets us *quickly* create a specialization of an existing object
    - all we need do is program the differences
- But inheritance is not a panacea:
    - the extension must be prepared in advance, as a new class or factory
    - the kind of extension can't be changed at runtime
    - with single inheritance, you have just one shot

Costs of Inheritance

- What happens when you get it wrong?
- Reasonable, usable and Exemplary are coins with two sides!
  - ¬ reasonable: making changes near the top of an incorrectly-modeled hierarchy
  - ¬ usable: recumbentMountainBike (or immutableSet) can’t be built
  - ¬ exemplary: can’t extend an incorrectly-modeled hierarchy

Composition

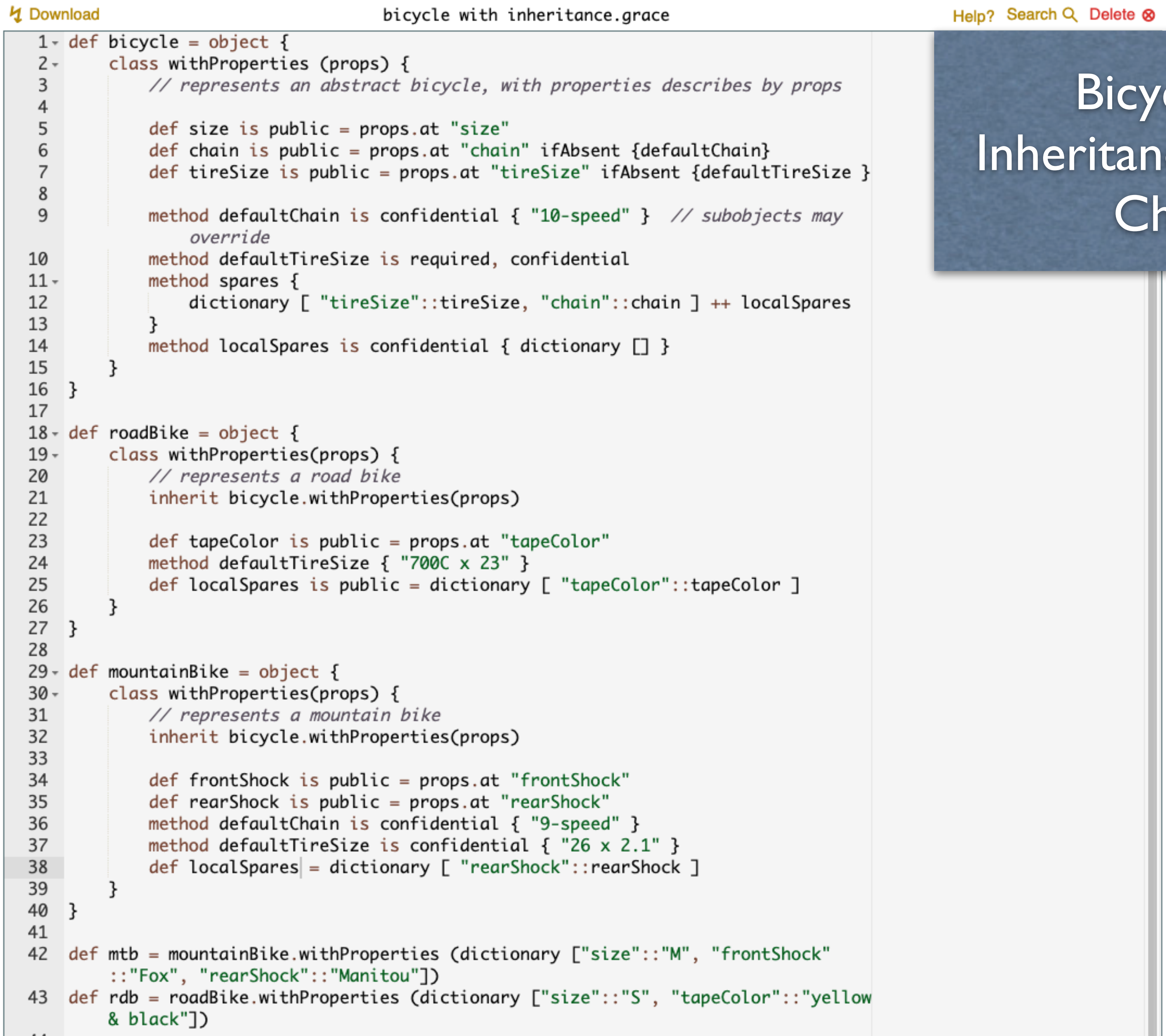
- Pros
    - component can be changed at runtime
        - e.g., state pattern
    - clear separation of responsibilities
        - need know only the interface of the component
- Cons
    - more work
        - define separate classes for part, parts ...
    - delegation not supported by most languages
        - must use self delegation pattern (Beck, p.67)

Metz:

- Inheritance:
  - for the cost of arranging objects in a hierarchy, you get message delegation for free
- Composition:
  - reverses these costs & benefits:
    - not restricted to a hierarchy; objects relationships are explicit
    - delegation of messages must *also* be explicit
- when faced with a problem that composition can solve, you should be biased towards using composition

Composing a Bicycle from Parts

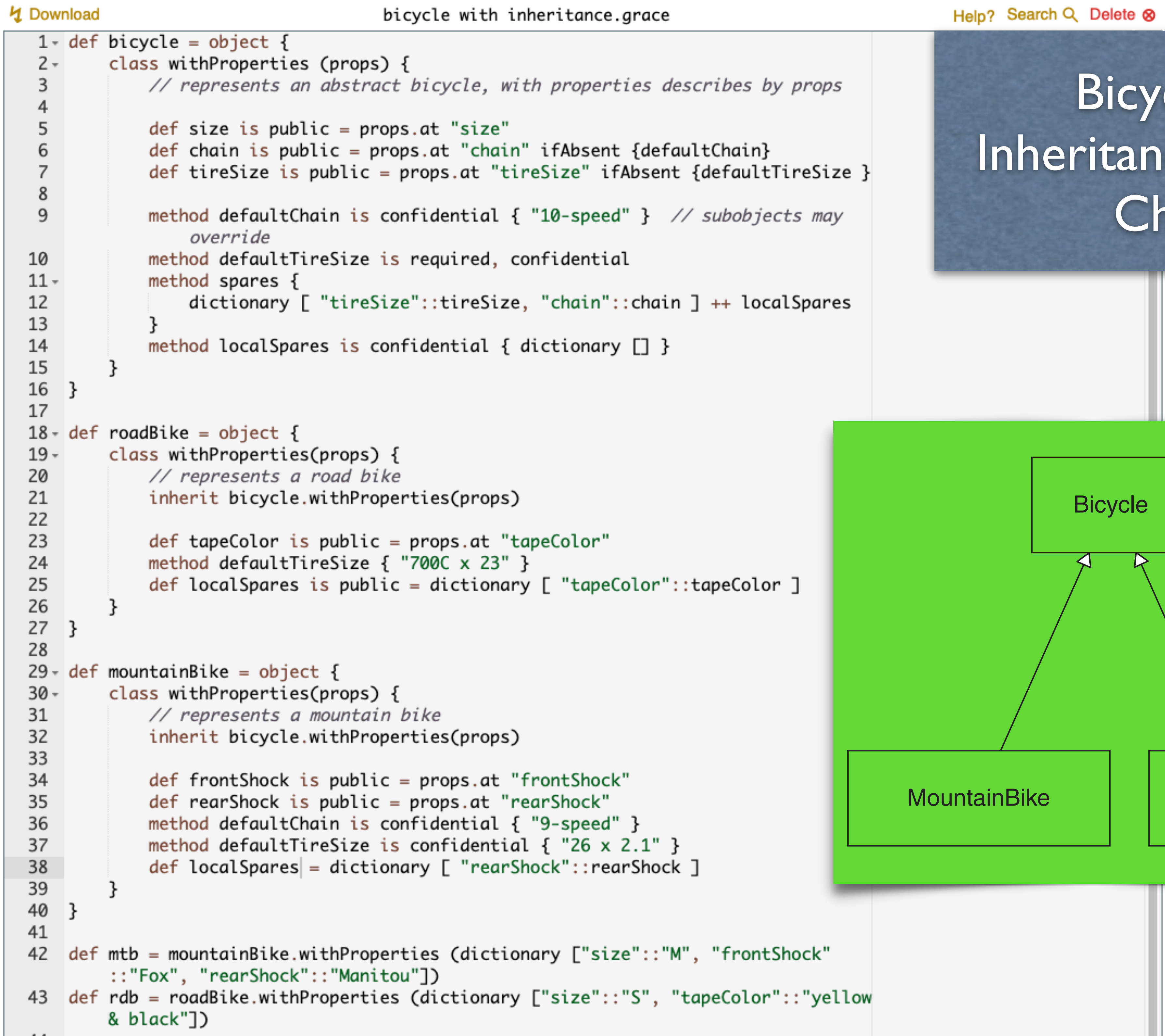
Bicycle with Inheritance from Chapter 6



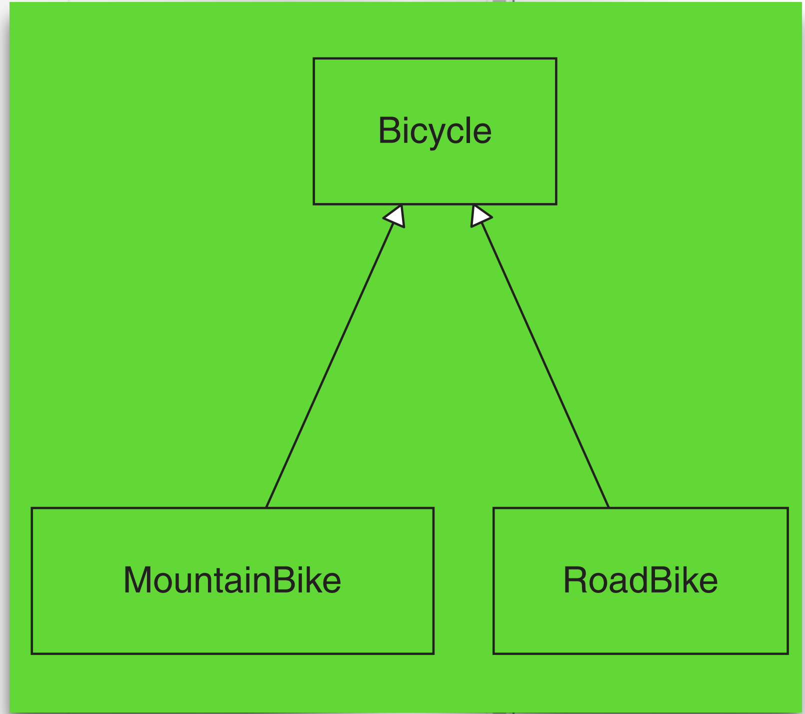
bicycle with inheritance.grace

```
def bicycle = object {
    class withProperties (props) {
        // represents an abstract bicycle, with properties describes by props

        def size is public = props.at "size"
        def chain is public = props.at "chain" ifAbsent {defaultChain}
        def tireSize is public = props.at "tireSize" ifAbsent {defaultTireSize }

        method defaultChain is confidential { "10-speed" }  // subobjects may override
        method defaultTireSize is required, confidential
        method spares {
            dictionary [ "tireSize"::tireSize, "chain"::chain ] ++ localSpares
        }
        method localSpares is confidential { dictionary [] }
    }
}

def roadBike = object {
    class withProperties(props) {
        // represents a road bike
        inherit bicycle.withProperties(props)

        def tapeColor is public = props.at "tapeColor"
        method defaultTireSize { "700C x 23" }
        def localSpares is public = dictionary [ "tapeColor"::tapeColor ]
    }
}

def mountainBike = object {
    class withProperties(props) {
        // represents a mountain bike
        inherit bicycle.withProperties(props)

        def frontShock is public = props.at "frontShock"
        def rearShock is public = props.at "rearShock"
        method defaultChain is confidential { "9-speed" }
        method defaultTireSize is confidential { "26 x 2.1" }
        def localSpares = dictionary [ "rearShock"::rearShock ]
    }
}

def mtb = mountainBike.withProperties (dictionary ["size"::"M", "frontShock"::"Fox", "rearShock"::"Manitou"])
def rdb = roadBike.withProperties (dictionary ["size"::"S", "tapeColor"::"yellow & black"])
```

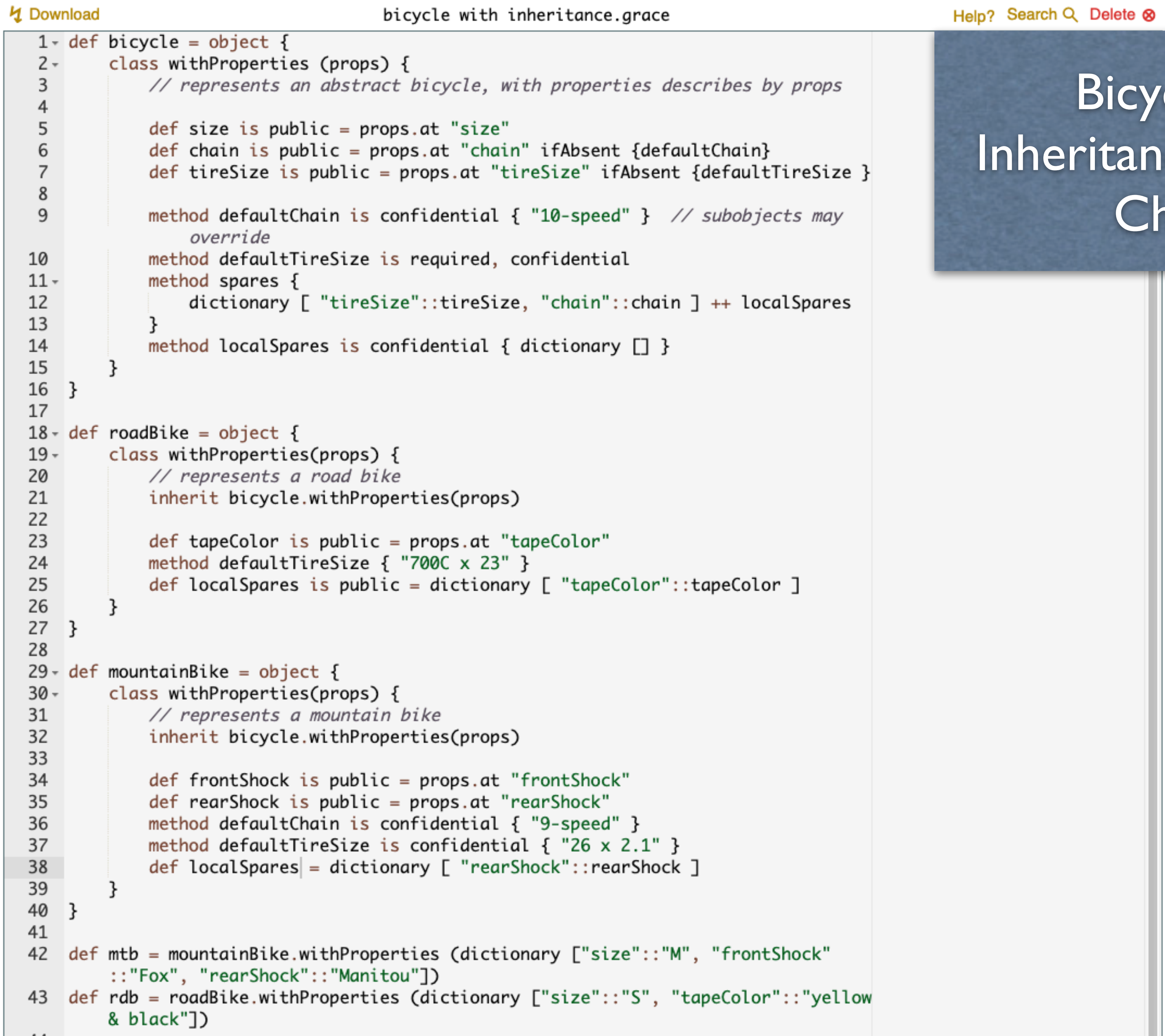
Bicycle with Inheritance from Chapter 6


Download bicycle with inheritance.grace Help? Search Delete

```
def bicycle = object {
    class withProperties (props) {
        // represents an abstract bicycle, with properties describes by props

        def size is public = props.at "size"
        def chain is public = props.at "chain" ifAbsent {defaultChain}
        def tireSize is public = props.at "tireSize" ifAbsent {defaultTireSize }

        method defaultChain is confidential { "10-speed" }  // subobjects may override
        method defaultTireSize is required, confidential
        method spares {
            dictionary [ "tireSize"::tireSize, "chain"::chain ] ++ localSpares
        }
        method localSpares is confidential { dictionary [] }
    }
}

def roadBike = object {
    class withProperties(props) {
        // represents a road bike
        inherit bicycle.withProperties(props)

        def tapeColor is public = props.at "tapeColor"
        method defaultTireSize { "700C x 23" }
        def localSpares is public = dictionary [ "tapeColor"::tapeColor ]
    }
}

def mountainBike = object {
    class withProperties(props) {
        // represents a mountain bike
        inherit bicycle.withProperties(props)

        def frontShock is public = props.at "frontShock"
        def rearShock is public = props.at "rearShock"
        method defaultChain is confidential { "9-speed" }
        method defaultTireSize is confidential { "26 x 2.1" }
        def localSpares = dictionary [ "rearShock"::rearShock ]
    }
}

def mtb = mountainBike.withProperties (dictionary ["size"::"M", "frontShock"::"Fox", "rearShock"::"Manitou"])
def rdb = roadBike.withProperties (dictionary ["size"::"S", "tapeColor"::"yellow & black"])
```

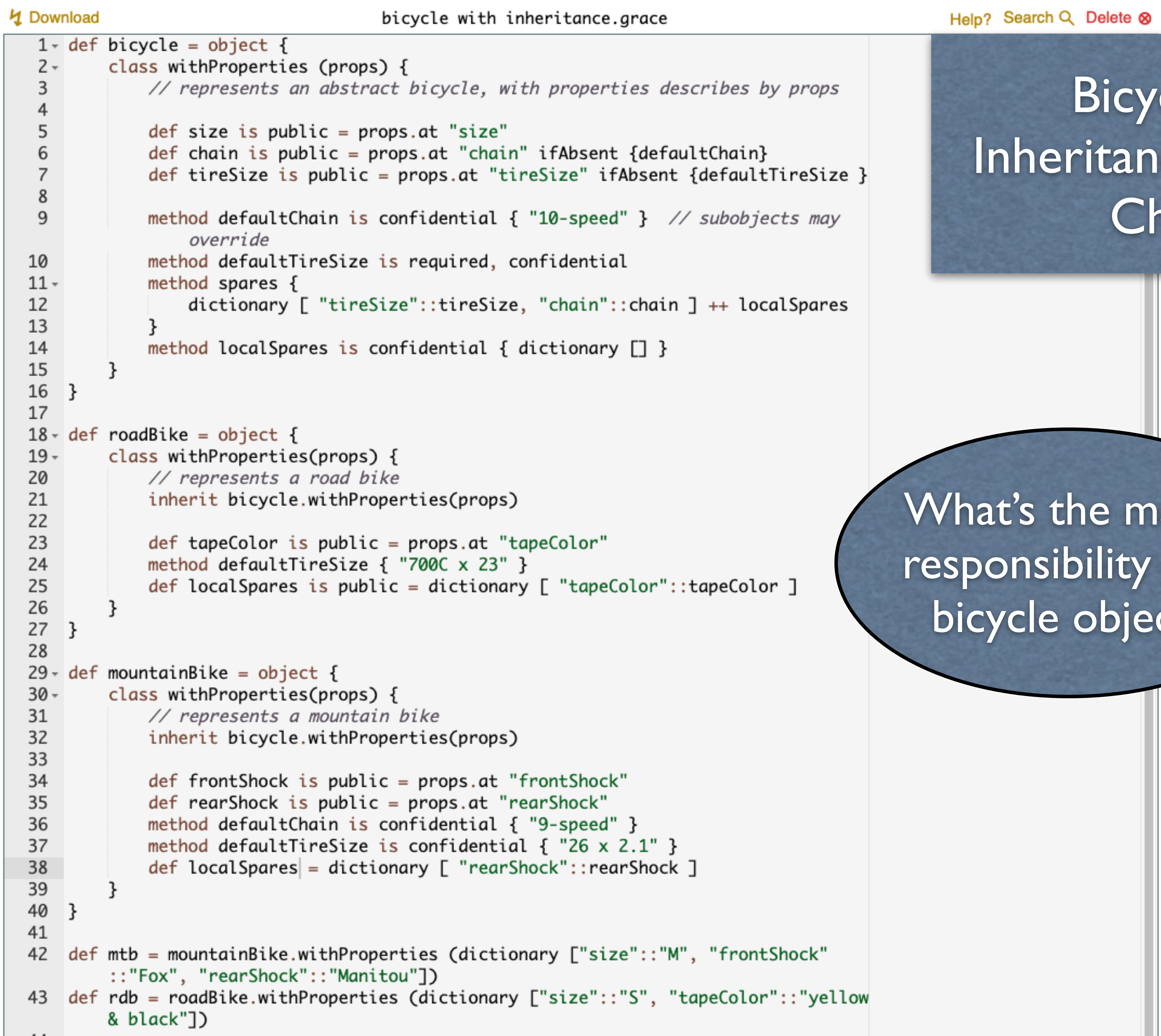


Bicycle with Inheritance from Chapter 6



```
def bicycle = object {
    class withProperties (props) {
        // represents an abstract bicycle, with properties describes by props

        def size is public = props.at "size"
        def chain is public = props.at "chain" ifAbsent {defaultChain}
        def tireSize is public = props.at "tireSize" ifAbsent {defaultTireSize }

        method defaultChain is confidential { "10-speed" }  // subobjects may override
        method defaultTireSize is required, confidential
        method spares {
            dictionary [ "tireSize"::tireSize, "chain"::chain ] ++ localSpares
        }
        method localSpares is confidential { dictionary [] }
    }
}

def roadBike = object {
    class withProperties(props) {
        // represents a road bike
        inherit bicycle.withProperties(props)

        def tapeColor is public = props.at "tapeColor"
        method defaultTireSize { "700C x 23" }
        def localSpares is public = dictionary [ "tapeColor"::tapeColor ]
    }
}

def mountainBike = object {
    class withProperties(props) {
        // represents a mountain bike
        inherit bicycle.withProperties(props)

        def frontShock is public = props.at "frontShock"
        def rearShock is public = props.at "rearShock"
        method defaultChain is confidential { "9-speed" }
        method defaultTireSize is confidential { "26 x 2.1" }
        def localSpares = dictionary [ "rearShock"::rearShock ]
    }
}

def mtb = mountainBike.withProperties (dictionary ["size"::"M", "frontShock"::"Fox", "rearShock"::"Manitou"])
def rdb = roadBike.withProperties (dictionary ["size"::"S", "tapeColor"::"yellow & black"])
```

Bicycle with Inheritance from Chapter 6



```
def bicycle = object {
    class withProperties (props) {
        // represents an abstract bicycle, with properties describes by props

        def size is public = props.at "size"
        def chain is public = props.at "chain" ifAbsent {defaultChain}
        def tireSize is public = props.at "tireSize" ifAbsent {defaultTireSize }

        method defaultChain is confidential { "10-speed" }  // subobjects may override
        method defaultTireSize is required, confidential
        method spares {
            dictionary [ "tireSize"::tireSize, "chain"::chain ] ++ localSpares
        }
        method localSpares is confidential { dictionary [] }
    }
}

def roadBike = object {
    class withProperties(props) {
        // represents a road bike
        inherit bicycle.withProperties(props)

        def tapeColor is public = props.at "tapeColor"
        method defaultTireSize { "700C x 23" }
        def localSpares is public = dictionary [ "tapeColor"::tapeColor ]
    }
}

def mountainBike = object {
    class withProperties(props) {
        // represents a mountain bike
        inherit bicycle.withProperties(props)

        def frontShock is public = props.at "frontShock"
        def rearShock is public = props.at "rearShock"
        method defaultChain is confidential { "9-speed" }
        method defaultTireSize is confidential { "26 x 2.1" }
        def localSpares = dictionary [ "rearShock"::rearShock ]
    }
}

def mtb = mountainBike.withProperties (dictionary ["size"::"M", "frontShock"::"Fox", "rearShock"::"Manitou"])
def rdb = roadBike.withProperties (dictionary ["size"::"S", "tapeColor"::"yellow & black"])
```

What's the major responsibility of a bicycle object?

- What's the major responsibility of a bicycle object?
- To respond to the `spares` request with a collection of spare parts
- Bicycles have parts; this feels like a *bicycle should be composed of parts*
- So, let's create a `parts` object
  - `bicycles` will delegate responsibility for spares to their `parts`

- `bicycles` will delegate responsibility for spares to their `parts`

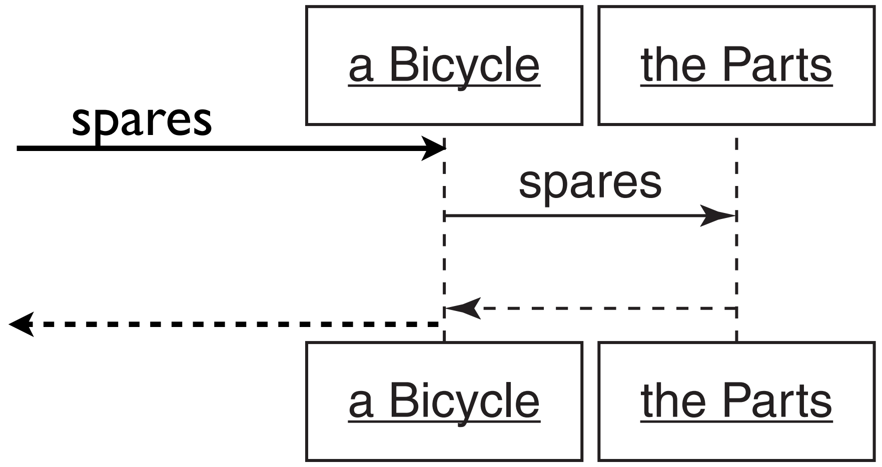


Figure 8.1 A *`Bicycle`* asks *`Parts`* for *`spares`*.

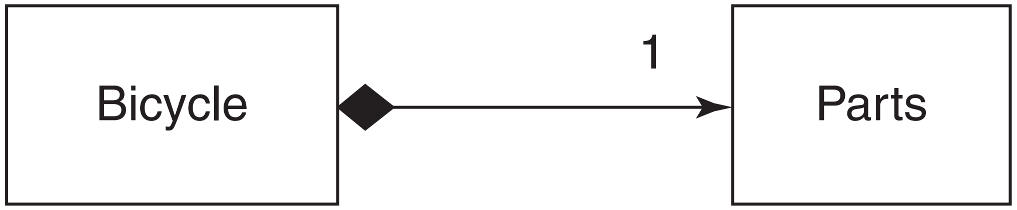


Figure 8.2 A *`Bicycle`* has-a *`Parts`*.

- `bicycles` will delegate responsibility for spares to their `parts`

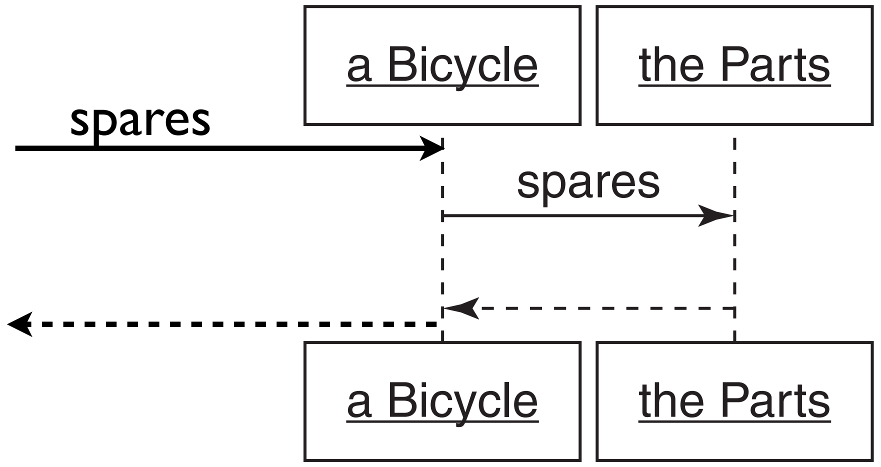


Figure 8.1 A *`Bicycle`* asks *`Parts`* for *`spares`*.

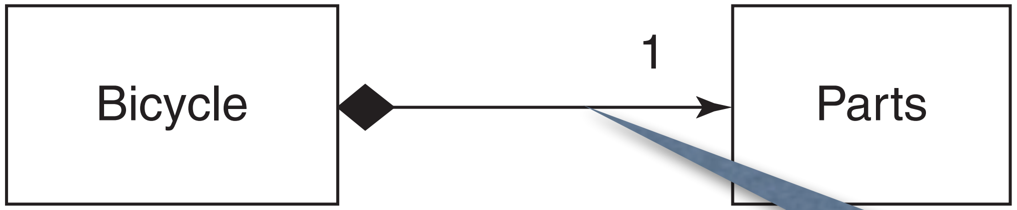


Figure 8.2 A *`Bicycle`* has-a *`Parts`*.

relationship

- `bicycles` will delegate responsibility for spares to their `parts`

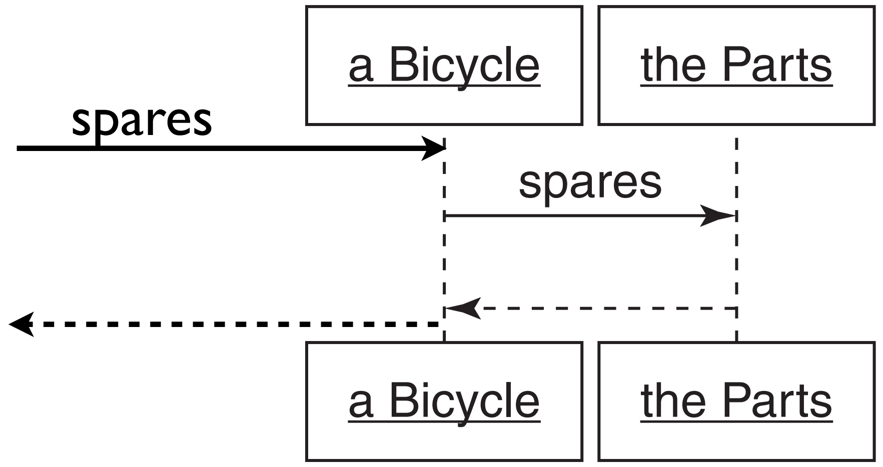


Figure 8.1 A *`Bicycle`* asks *`Parts`* for *`spares`*.

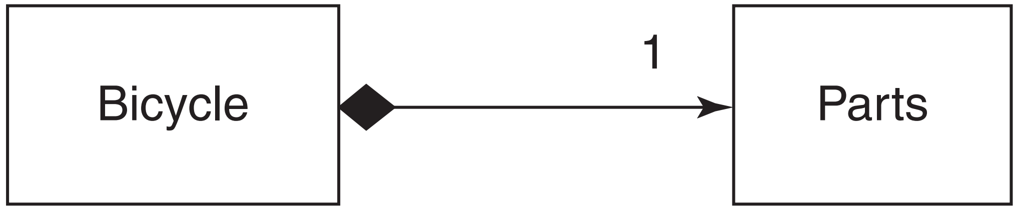


Figure 8.2 A *`Bicycle`* has-a *`Parts`*.

- `bicycles` will delegate responsibility for spares to their `parts`

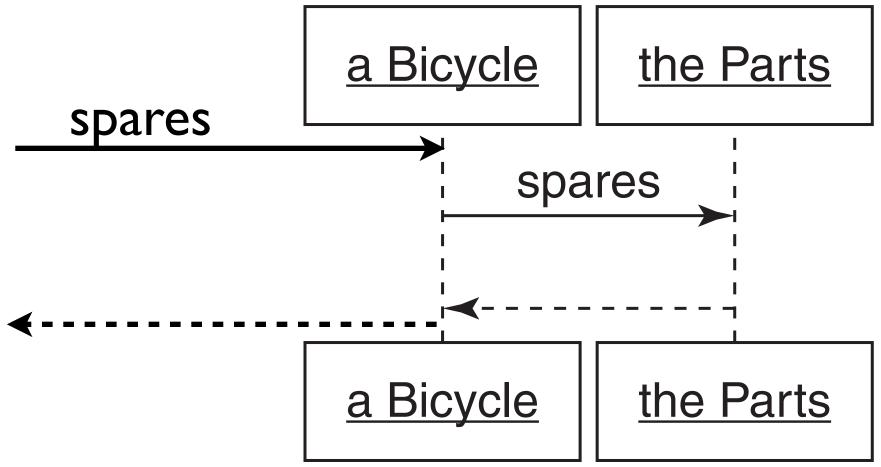


Figure 8.1 A *`Bicycle`* asks *`Parts`* for *`spares`*.

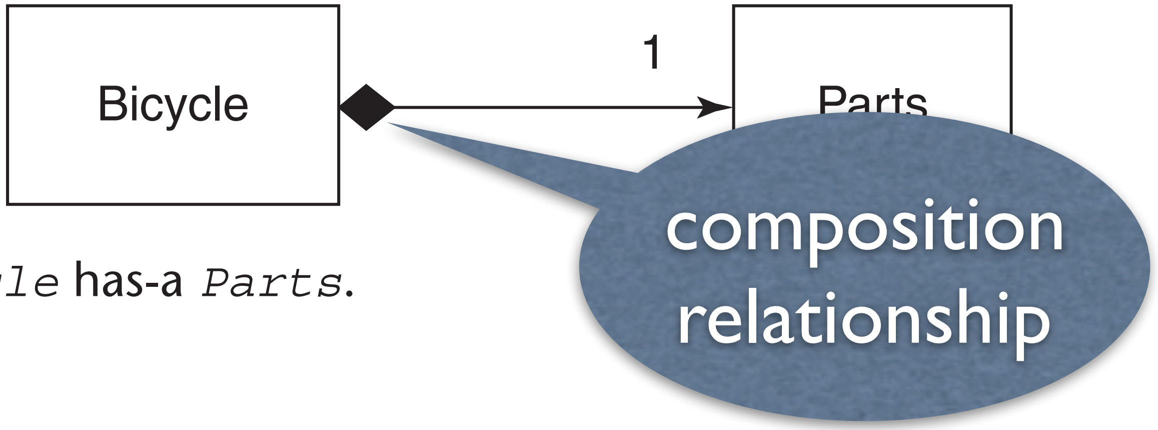


Figure 8.2 A *`Bicycle`* has-a *`Parts`*.

- `bicycles` will delegate responsibility for spares to their `parts`

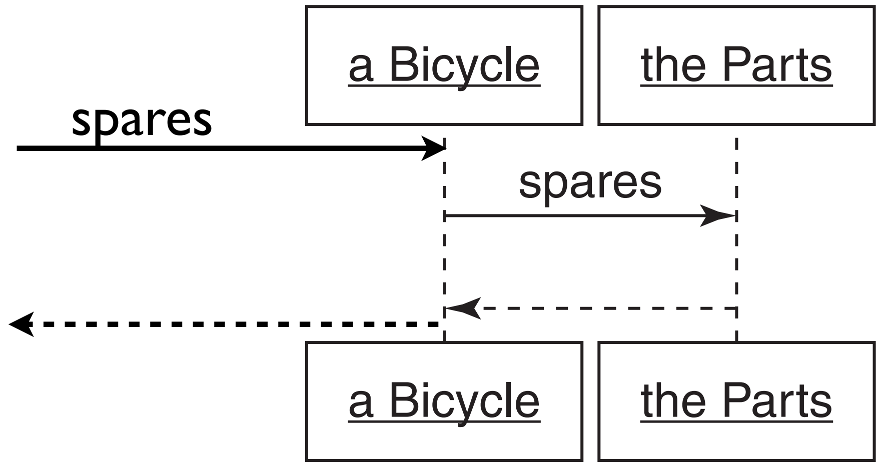


Figure 8.1 A *`Bicycle`* asks *`Parts`* for *`spares`*.

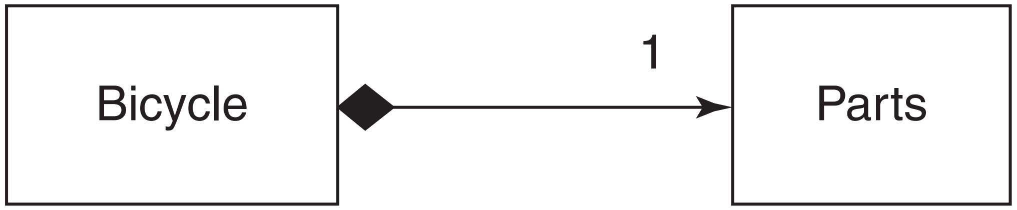


Figure 8.2 A *`Bicycle`* has-a *`Parts`*.

- `bicycles` will delegate responsibility for spares to their `parts`

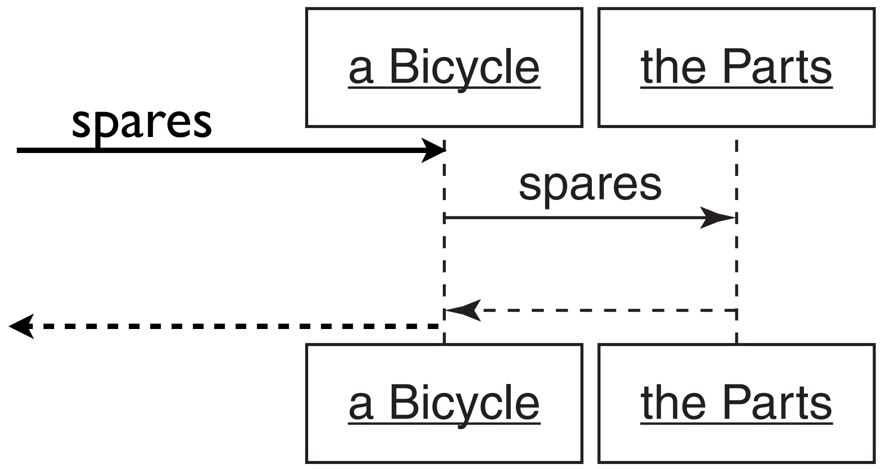


Figure 8.1 A *`Bicycle`* asks *`Parts`* for *`spares`*.

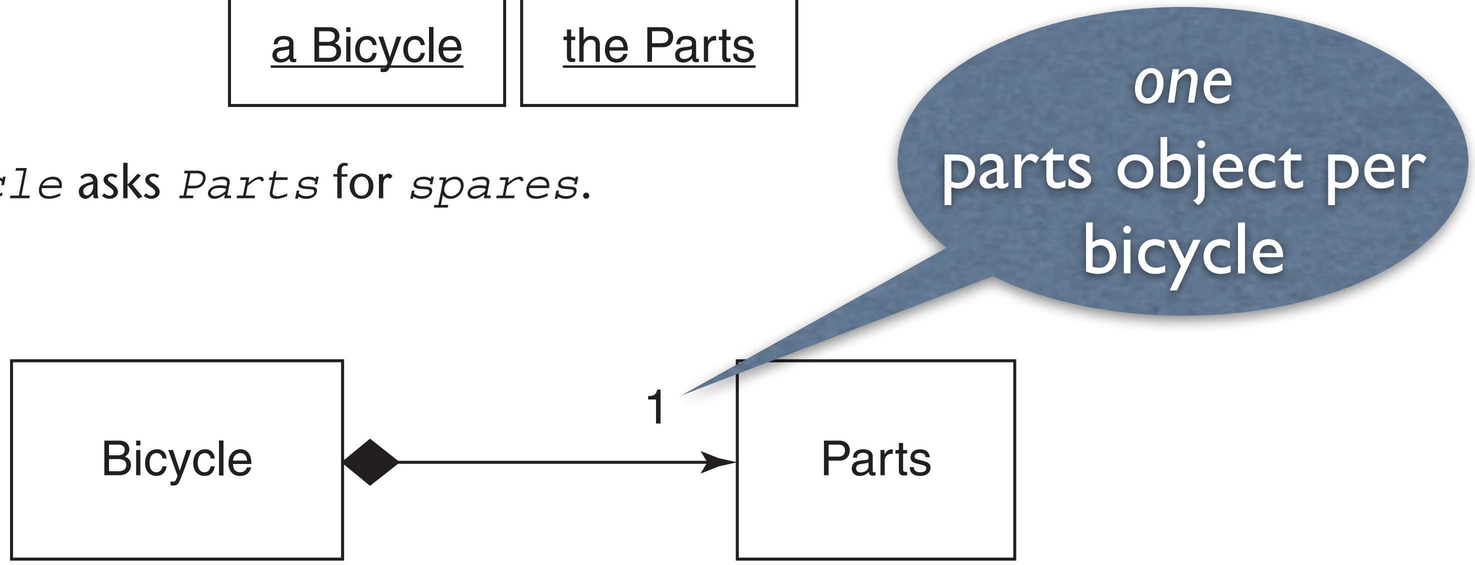


Figure 8.2 A *`Bicycle`* has-a *`Parts`*.

- `bicycles` will delegate responsibility for spares to their `parts`

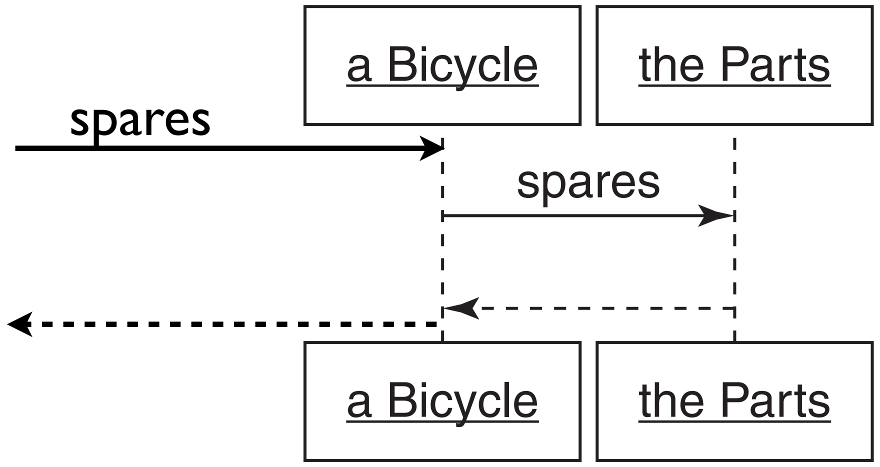


Figure 8.1 A *`Bicycle`* asks *`Parts`* for *`spares`*.

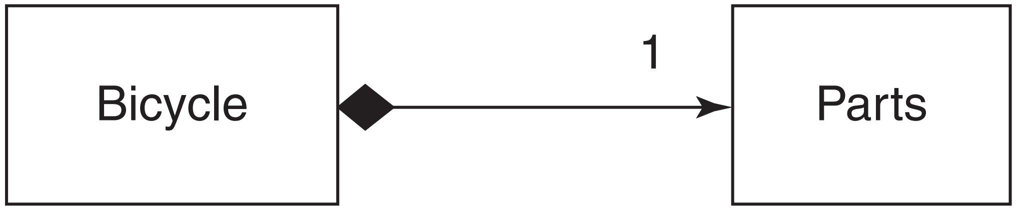
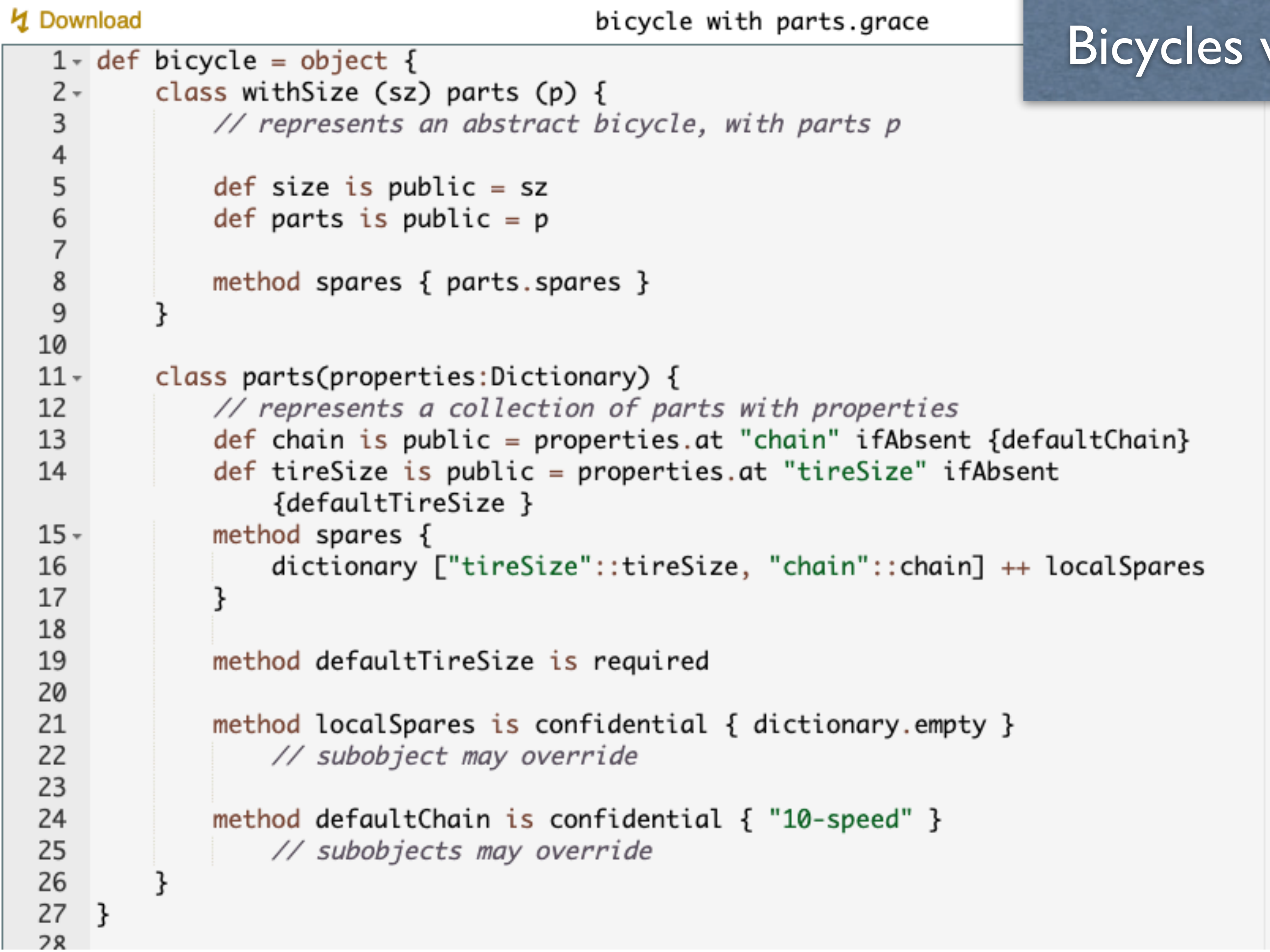


Figure 8.2 A *`Bicycle`* has-a *`Parts`*.

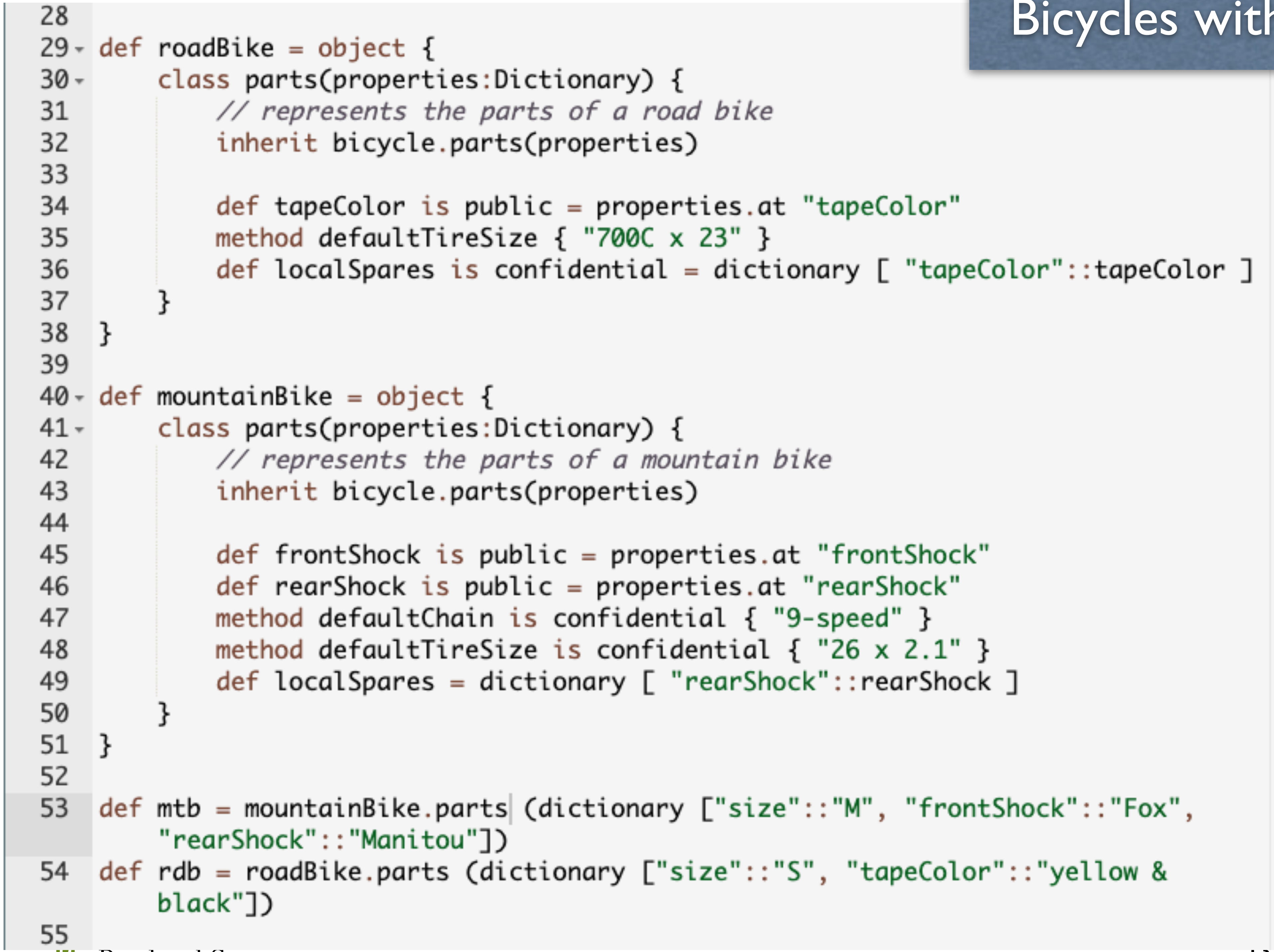
Bicycles with Parts



Download bicycle with parts.grace

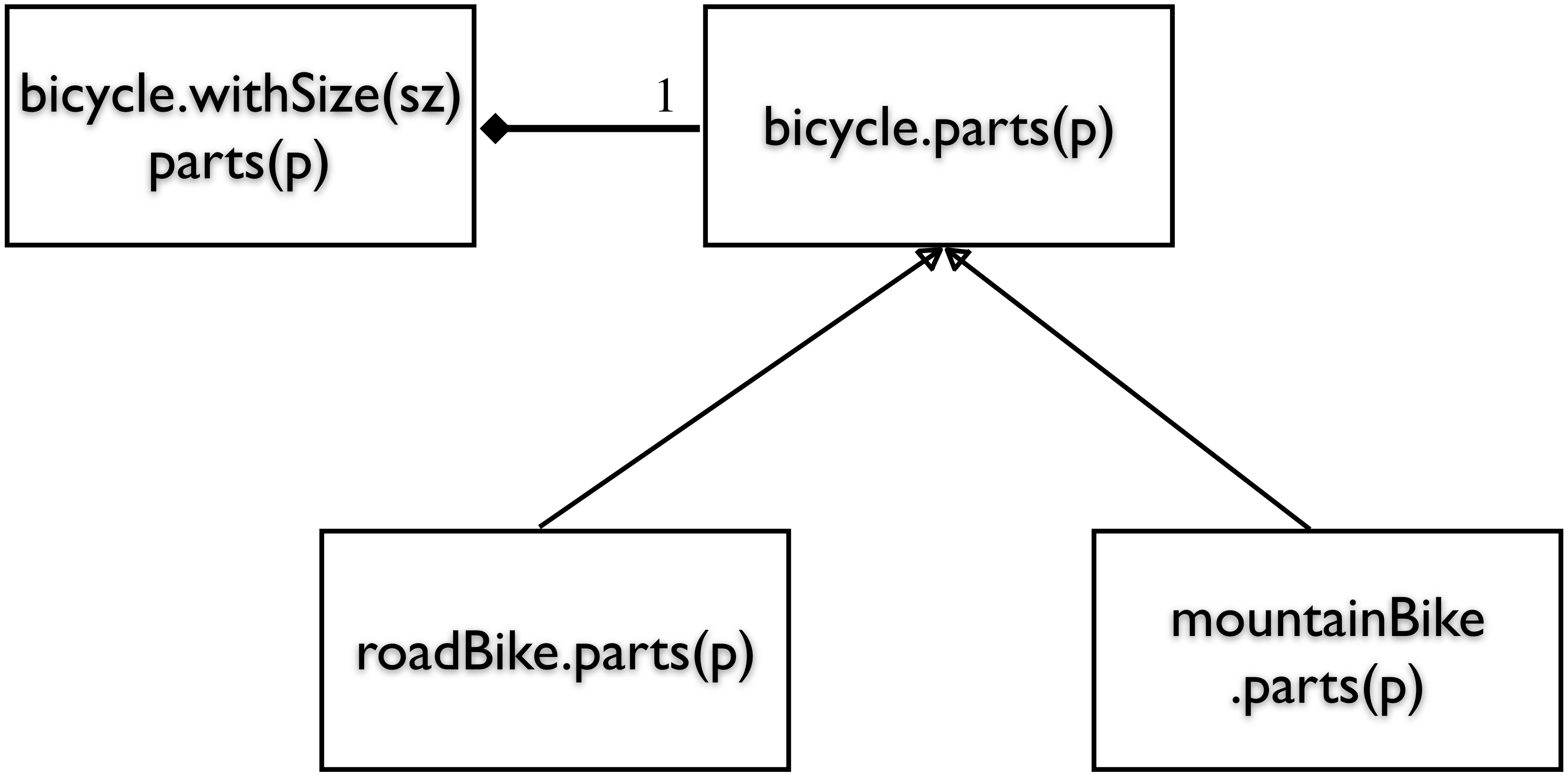
```
def bicycle = object {
    class withSize (sz) parts (p) {
        // represents an abstract bicycle, with parts p

        def size is public = sz
        def parts is public = p

        method spares { parts.spares }
    }

    class parts(properties:Dictionary) {
        // represents a collection of parts with properties
        def chain is public = properties.at "chain" ifAbsent {defaultChain}
        def tireSize is public = properties.at "tireSize" ifAbsent
            {defaultTireSize }
        method spares {
            dictionary ["tireSize"::tireSize, "chain"::chain] ++ localSpares
        }

        method defaultTireSize is required

        method localSpares is confidential { dictionary.empty }
            // subobject may override

        method defaultChain is confidential { "10-speed" }
            // subobjects may override
    }
}
```

Bicycles with Parts



```
def roadBike = object {
    class parts(properties:Dictionary) {
        // represents the parts of a road bike
        inherit bicycle.parts(properties)

        def tapeColor is public = properties.at "tapeColor"
        method defaultTireSize { "700C x 23" }
        def localSpares is confidential = dictionary [ "tapeColor"::tapeColor ]
    }
}

def mountainBike = object {
    class parts(properties:Dictionary) {
        // represents the parts of a mountain bike
        inherit bicycle.parts(properties)

        def frontShock is public = properties.at "frontShock"
        def rearShock is public = properties.at "rearShock"
        method defaultChain is confidential { "9-speed" }
        method defaultTireSize is confidential { "26 x 2.1" }
        def localSpares = dictionary [ "rearShock"::rearShock ]
    }
}

def mtb = mountainBike.parts (dictionary ["size"::"M", "frontShock"::"Fox", "rearShock"::"Manitou"])
def rdb = roadBike.parts (dictionary ["size"::"S", "tapeColor"::"yellow & black"])

```

Hierarchy (after Fig 8.3)



The result

- Most code from `bicycle` moves into `parts`
  - Metz: *wasn't a big change, and isn't much of an improvement*
  - *made it blindingly obvious just how little Bicycle specific code there was to begin with*
  - *Most of the code …deals with individual parts; the* `Parts` *hierarchy now cries out for another refactoring.*

Composing the Parts Object

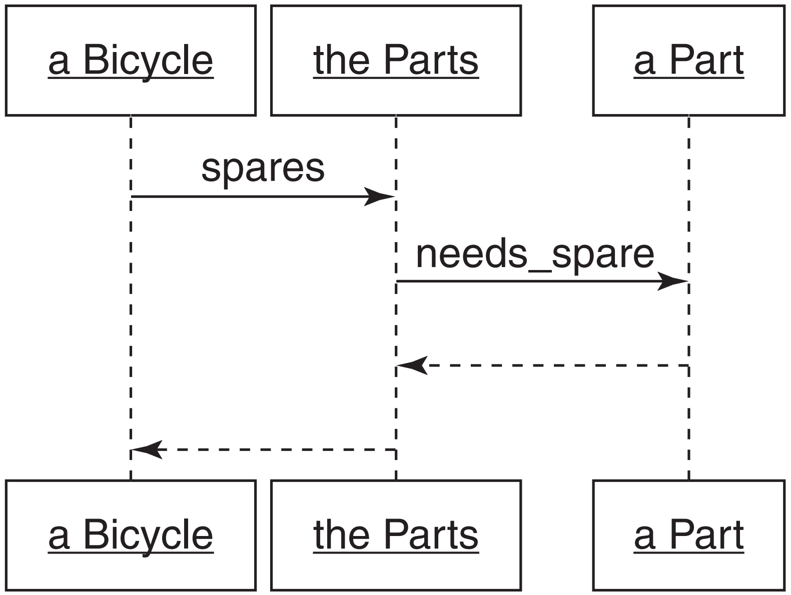


Figure 8.4 `Bicycle` sends `spares` to `Parts`, `Parts` sends `needs_spare` to each `Part`.

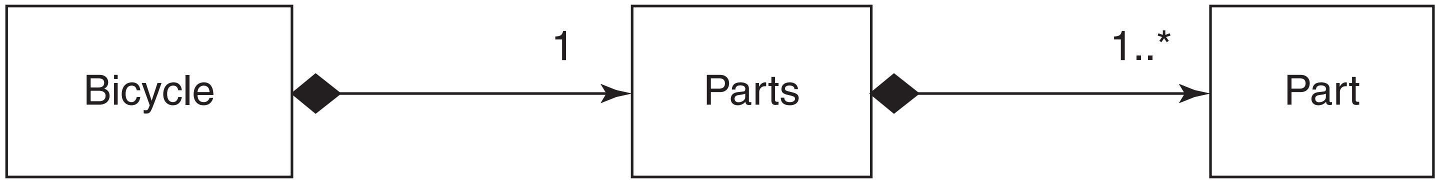


Figure 8.5 `Bicycle` holds one `Parts` object, which in turn holds many `Part` objects.

Composing the Parts Object

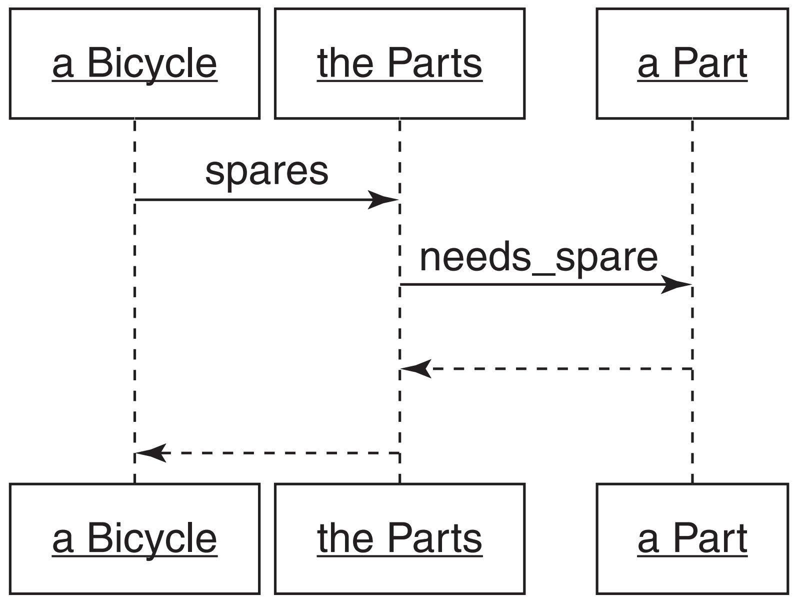


Figure 8.4 `Bicycle` sends `spares` to `Parts`, `Parts` sends `needs_spar`

one or more part objects per parts object

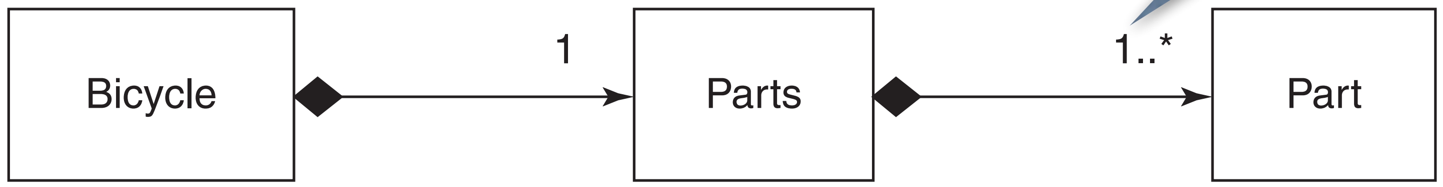


Figure 8.5 `Bicycle` holds one `Parts` object, which in turn holds many `Part` objects.

Composing the Parts Object

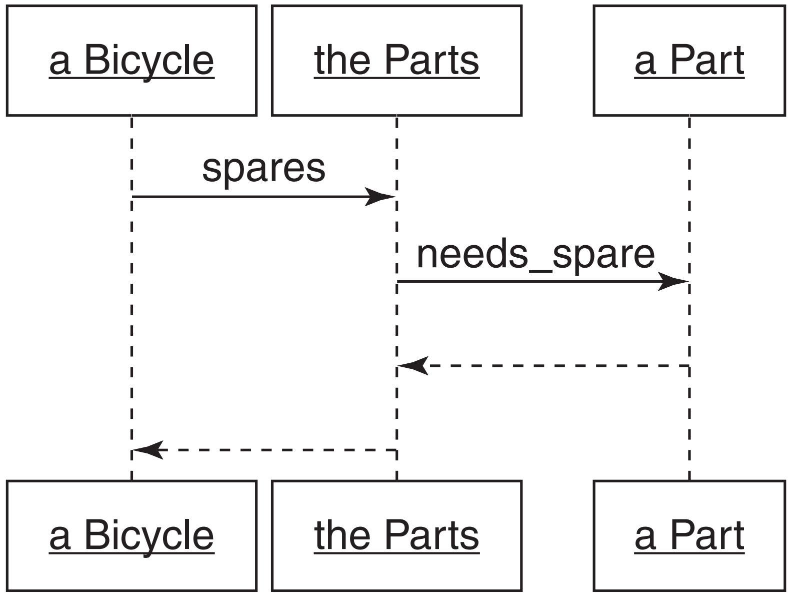


Figure 8.4 `Bicycle` sends `spares` to `Parts`, `Parts` sends `needs_spare` to each `Part`.

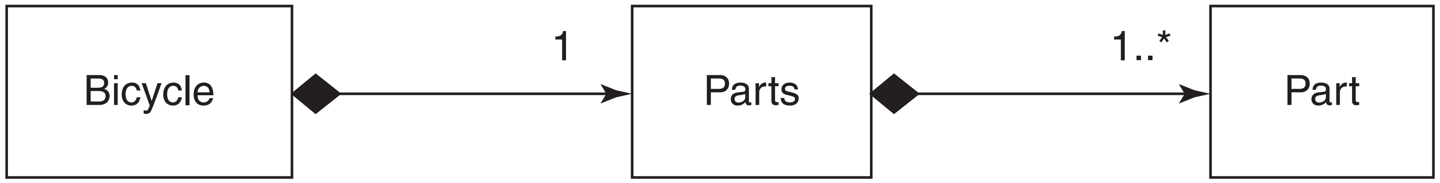


Figure 8.5 `Bicycle` holds one `Parts` object, which in turn holds many `Part` objects.

Portland State UNIVERSITY

Should the Parts object be like a List?

Portland State UNIVERSITY

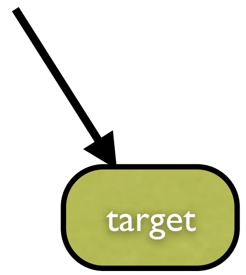
Delegation

- Delegation allows you to share implementation without inheritance
- Pass part of your work on to another object. Put that object in one of your instance variables
    - e.g., a *Path* contains a field *form*, the bit mask responsible for actually drawing on the display.
    - e.g., a *Text* contains a *String*

What about **self**?

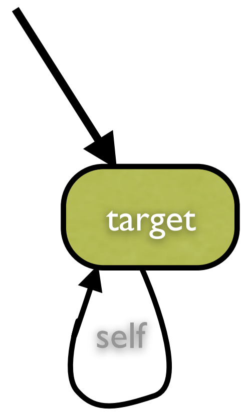
- When you delegate, the receiver of the delegating message (the *delegate*) is no longer the target
  - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, *forward* messages unchanged — Beck calls this *Simple Delegation*

What about **self**?



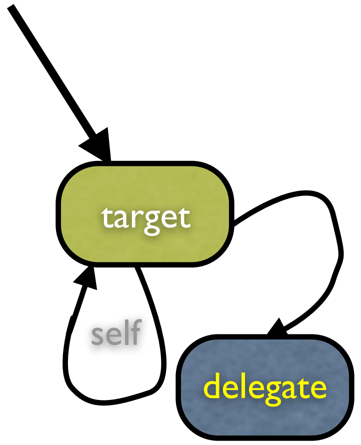
- When you delegate, the receiver of the delegating message (the *delegate*) is no longer the target
    - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, *forward* messages unchanged — Beck calls this *Simple Delegation*

What about **self**?



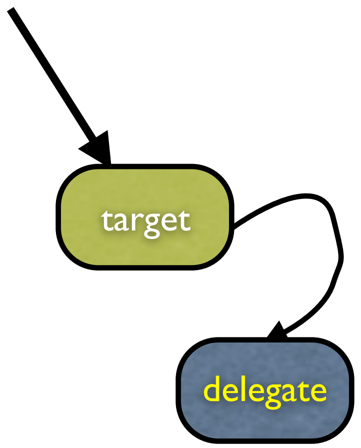
- When you delegate, the receiver of the delegating message (the *delegate*) is no longer the target
    - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, *forward* messages unchanged — Beck calls this *Simple Delegation*

What about **self**?



- When you delegate, the receiver of the delegating message (the *delegate*) is no longer the target
  - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, *forward* messages unchanged — Beck calls this *Simple Delegation*

What about **self**?

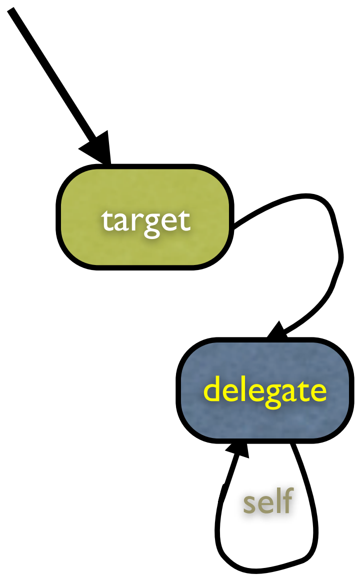


- When you delegate, the receiver of the delegating message (the *delegate*) is no longer the target
  - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, *forward* messages unchanged — Beck calls this *Simple Delegation*

Portland State UNIVERSITY

What about **self**?

- When you delegate, the receiver of the delegating message (the *delegate*) is no longer the target
  - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, *forward* messages unchanged — Beck calls this *Simple Delegation*



Portland State UNIVERSITY

Simple Delegation Example

```
method do(aBlock) {
    collectionOfPoints.do(aBlock) }

method map(aBlock) {
        def newPath = path.withForm(self.form)
        newPath.points :=
                (collectionOfPoints.map(aBlock)
        newPath }
```

Simple Delegation works when:

- you don’t need the state of the original target object
- you don’t need the behaviour of the original target object
- you don’t need the identity of the original target object

If you need these things, use *self delegation*

Self Delegation

- When the delegate *needs* a reference to the delegating object…
- Pass along the delegating object as an additional parameter.

Self Delegation Example

```
Dictionary: method at(key) put(value) {
      self.hashTable.at(key) put(value) for(self)
}
HashTable: method at(key) put(value) for(aCollection) {
      def hash = aCollection.hashOf(key)
}
Dictionary: method hashOf(anObject) {
      anObject.hash
}
PlugableDictionary: method hashOf(anObject) {
      injectedHash(anObject)
}
```