Reuse

CS 420/520 Object-oriented Programming

Andrew P. Black

Portland State
UNIVERSITY

Reuse vs. Reusability

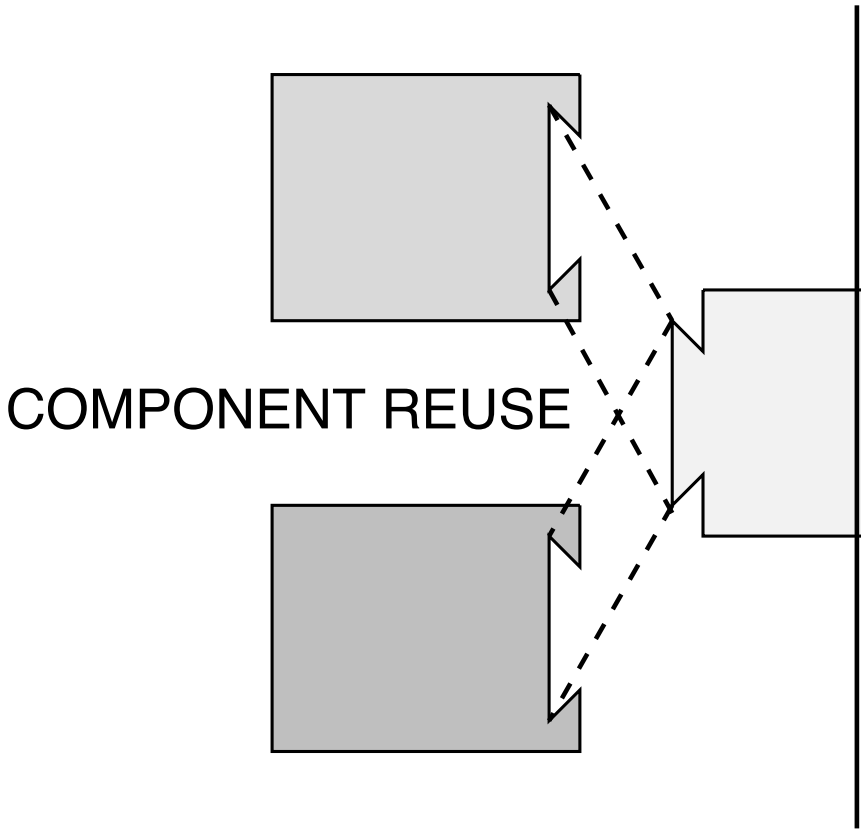
- Reuse
  - ✦ use of (source) code without copying
- Reusability
  - ✦ making it easy or possible to do reuse

⇒ Reuse is evidence of reusability!

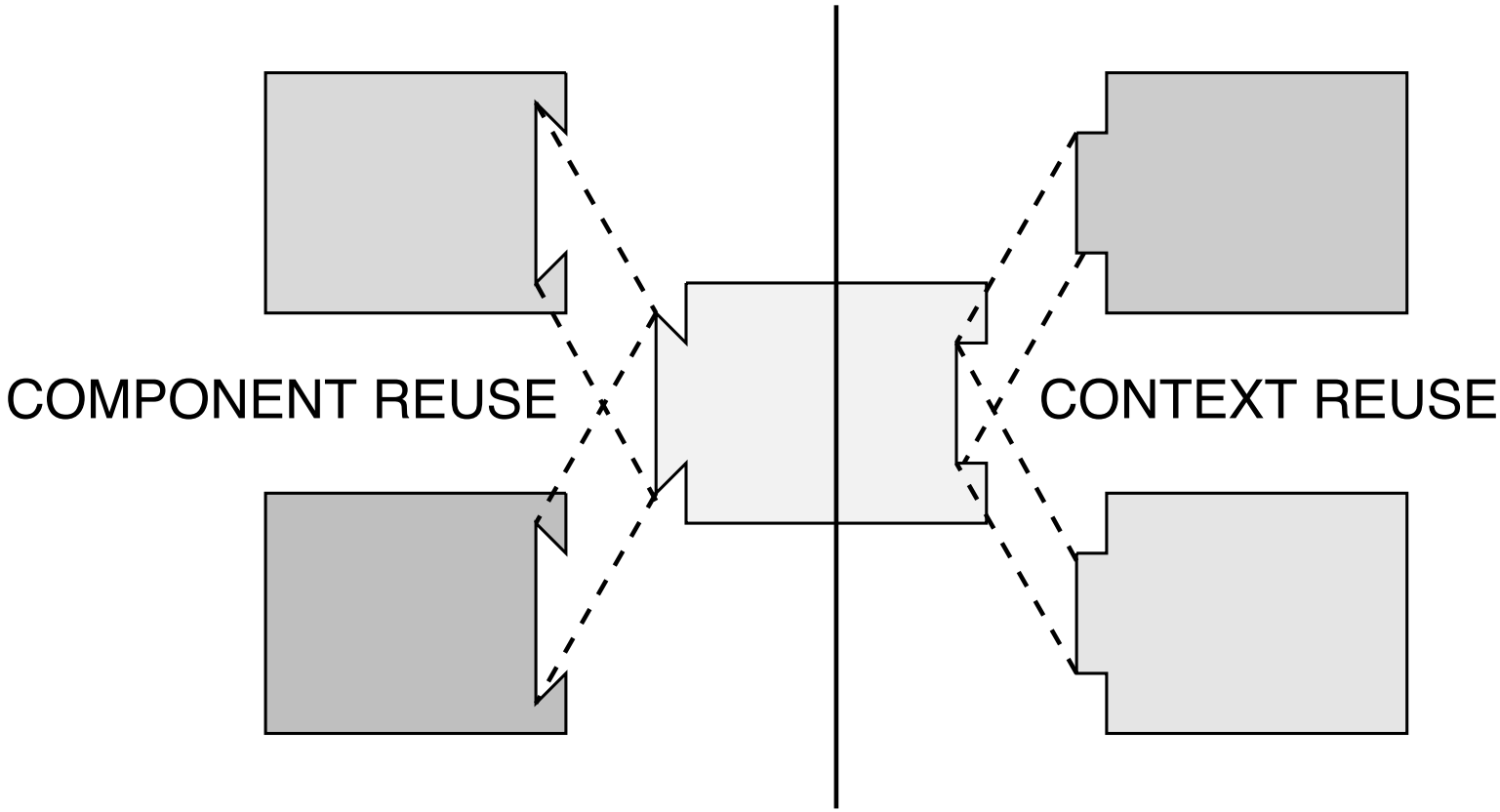
Why is Reuse Hard?

- In functional and procedural languages:
  - *reuse* means *using* a function procedure
  - *reusability* means writing a function so that it can be reused.
- In object-oriented languages:
  - *reuse* means *using* a method
  - *reusability* means writing a method so that it can be reused
- Is that all?

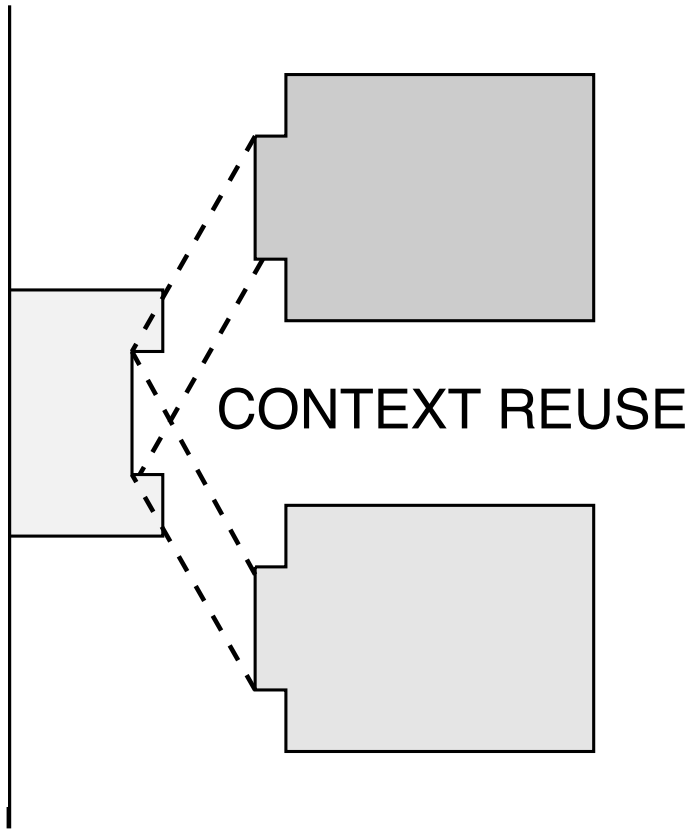
Components and Contexts



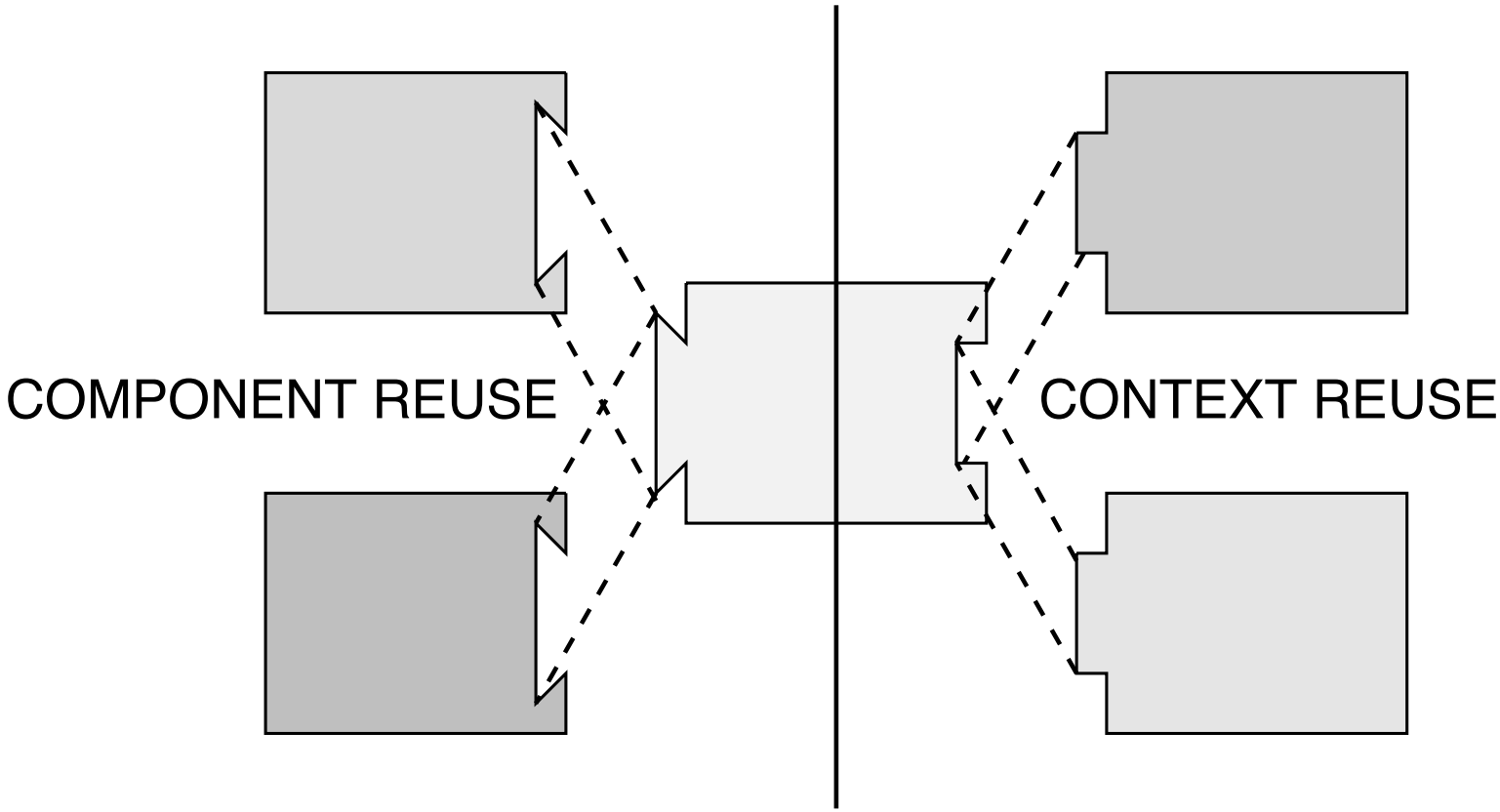
Components and Contexts



Components and Contexts



Components and Contexts



Two kinds of Reusability

- *Component Reuse*
  - increasing reusability of a *component* means increasing the number of contexts that can sensibly use the component
- *Context Reuse*
  - Increasing reusability of a context means increasing the number of components that can sensibly use the context

Dependencies

- A *dependency* between two pieces of code is a condition that one must meet in order to use the other, e.g.:
  - sender of message depends on receiver having a method with the right name
  - accessing an instance variable depends on target having the right instance variable
  - client of a Set depends on "no duplicates"

Increasing Reusability

Increasing Reusability

- Increasing Reusability means reducing dependencies?

Increasing Reusability

- Increasing Reusability means reducing dependencies?
  - ✦ Not so simple!

Increasing Reusability

- Increasing Reusability means reducing dependencies?
  - ✦ Not so simple!
- No dependency ⇒ no interaction!

Increasing Reusability

- Make dependencies:
  - explicit, if possible
    - e.g., parameter, import statement, …
  - checkable
    - e.g., sending a **self** or **super** message
  - intentional
    - deliberately introduced because the client will probably need them

Encapsulation

- Encapsulation constructs:
  - ✦ control unintentional dependencies
  - ✦ make intentional dependencies explicit
  - ✦ limit ways in which context can depend on component
    - ◦ allows new component to fit in old context (= reuse of context)

⇒ Encapsulation promotes reuse

Polymorphism

- When a single piece of code can operate on objects of several classes
  - ✦ Polymorphism requires only that the object has the required interface, *i.e.*, that the right messages are understood correctly.
  - ✦ No need to require a specific *implementation*

⇒ Polymorphism promotes reuse

Programming Patterns for Reuse

Portland State
UNIVERSITY

Complete Creation Method

Complete Creation Method

- Suppose:
  - Someone likes your class!
  - How to make it easy for her to *use* it!

Complete Creation Method

- Suppose:
  - Someone likes your class!
  - How to make it easy for her to *use* it!
- Provide methods that create well-formed instances.
  - Put them in the "instance creation" protocol on the class side
  - Name them with intention-revealing selectors

Complete Creation Method

- Examples:
  - Point x: 4 y: 3
  - Point r: 20 degrees: 36.8
  - SortedCollection new
  - SortedCollection
    sortBlock: [:a :b | a name <= b name]

Once and Only Once

Once and Only Once

- This means: if you have one thing to say, say it in once place

Once and Only Once

- This means: if you have one thing to say, say it in once place
- It also means: if you have more than one thing to say, don't say it all in one place!
  - Example: if the initialization of an instance variable is different from the setting of that instance variable, write *two* methods!

Example

Example

Window class >> **withTitle: aTextOrString**
    ↑ Window new title: aTextOrString;
      yourself

Example

```
Window class >> withTitle: aTextOrString
        ↑ Window new title: aTextOrString;
          yourself
```

```
Window >> title: aTextOrString
        initializing ← title isNil.

        title ← aTextOrString.

        initializing ifFalse: [self changed: #title]
```

Example (continued)

Example (continued)

Window class >> **withTitle: aTextOrString**

↑ Window new setTitle: aTextOrString;
yourself

Example (continued)

Window class >> **withTitle: aTextOrString**

↑ Window new setTitle: aTextOrString;
yourself

Window >> **setTitle: aTextOrString**

title ← aTextOrString.

Example (continued)

```
Window class >> withTitle: aTextOrString
    ↑ Window new setTitle: aTextOrString;
      yourself
```

```
Window >> setTitle: aTextOrString
    title ← aTextOrString.

Window >> title: aTextOrString
    title ← aTextOrString.
    self changed: #title
```

Dispatched Interpretation

- How can two objects cooperate when one wishes to conceal it's representation
  - Why *would* one wish to conceal its representation?
- Conceal the representation behind a *protocol*
  - e.g., Booleans with *ifTrue: ifFalse:*

But what if the representation is more complicated?

- pass an *interpreter to* the encoded object
- Beck's example:
  - a geometric shape
    - encoded as a sequence of line, curve, stroke and fill commands

- ShapePrinter >> **display: aShape**

```
| interp |
interp := anInterpreter writingOn: self canvass.
aShape sendCommandsTo: interp.
```

- Shape >> **sendCommandsTo: anObject**

```
self components do:
    [ :each | each sendCommandTo: anObject]
```

- How does the component know how to send a command to the interpreter?

- If the components are subclasses:
  - each one knows what command to send for itself. e.g.,
  - LineComponent >> **sendCommandTo: anObject**

```
self fromPoint printOn: anObject.
' ' printOn: anObject.
self toPoint printOn: anObject.
' line' printOn: anObject
```

- If the components are represented as symbols:
  - each Shape object will need a case statement …

- Why is this called "Dispatched Interpretation"?
  - the encoded object (Shape) dispatches a message to the client
  - the client interprets the message
  - You will have to design a *mediating protocol* between the objects. (Beck page 57)

- Note: all of the internal iterators are very simple examples of dispatched interpretation

  aComplexObject withSomeComponentsDo: aBlock

  - aBlock is an interpreter of avery simple protocol

    value: anArgument

Tell, Don't Ask

(Sharp Ch. 9)

- Tell objects what to do.
- Don't:
  - ask a question about an object's state,
  - make a decision based on the answer, and
  - tell the object what to do
- Why?

Tell, don't Ask — How to do it

Tell, don't Ask — How to do it

- Rectangle >> displayOn: aPort

```
    aPort isMemberOf: DisplayPort
        ifTrue: ["code for displaying on DisplayPort"].
    aPort isMemberOf: PrinterPort
        ifTrue: ["code for displaying on PrinterPort"].
    aPort isMemberOf: RemotePort
        ifTrue: ["code for displaying on RemotePort"].
```

Tell, don't Ask — How to do it

- Rectangle >> displayOn: aPort

```
aPort isMemberOf: DisplayPort
    ifTrue: ["code for displaying on DisplayPort"].
aPort isMemberOf: PrinterPort
    ifTrue: ["code for displaying on PrinterPort"].
aPort isMemberOf: RemotePort
    ifTrue: ["code for displaying on RemotePort"].
```

- What's wrong with this?

Tell, don't Ask — How to do it

- Rectangle >> displayOn: aPort

```
aPort isMemberOf: DisplayPort
    ifTrue: ["code for displaying on DisplayPort"].
aPort isMemberOf: PrinterPort
    ifTrue: ["code for displaying on PrinterPort"].
aPort isMemberOf: RemotePort
    ifTrue: ["code for displaying on RemotePort"].
```

- What's wrong with this?
  - ✦ How can we add new kinds of graphical object, like Ellipse?

Tell, don't Ask — How to do it

- Rectangle >> displayOn: aPort

```
aPort isMemberOf: DisplayPort
    ifTrue: ["code for displaying on DisplayPort"].
aPort isMemberOf: PrinterPort
    ifTrue: ["code for displaying on PrinterPort"].
aPort isMemberOf: RemotePort
    ifTrue: ["code for displaying on RemotePort"].
```

- What's wrong with this?
  - ✦ How can we add new kinds of graphical object, like Ellipse?
  - ✦ How can we add new kinds of Port?

Tell, don't Ask — How to do it

```
Rectangle>> displayOn: aPort
    aPort displayRectangle: self

Oval>> displayOn: aPort
    aPort displayOval: self

Bitmap>> displayOn: aPort
    aPort displayBitmap: self
```

... and similarly for the other graphical objects.

Tell, don't Ask — How to do it

- DisplayPort >> displayRectangle: aRect
  "code to display a rectangle on a displayPort"
  DisplayPort >> displayOval: aRect
  "code to display an oval on a displayPort"
  DisplayPort >> displayBitmap: aRect
  "code to display a bitmap on a displayPort"
  ... and similarly for the other graphical objects,

- PrinterPort >> displayRectangle: aRect
  "code to display a rectangle on a printerPort"
  PrinterPort >> displayOval: aRect
  "code to display an oval on a printerPort"
  PrinterPort >> displayBitmap: aRect
  "code to display a bimmp on a printerPort"
  ... and similarly for the other graphical objects

- similarly for the other display port classes.

How to do it: Double Dispatch

- Dispatch once on the graphical object:

```
Rectangle>> displayOn: aPort
    aPort displayRectangle: self
```

  - remember the result by using an intention reveling selector

- Dispatch again on what was the argument

```
PrinterPort >> displayRectangle: aRect
    "code to display a rectangle on a printerPort"
```

- Revealed in famous paper: Ingalls OOPSLA '86 pp. 347-349

Inheritance

- Kent Beck wrote (and then thought better of) in SBPP:
  - How do you design inheritance hierarchies?
  - Make all of your classes subclasses of Object at first. Create a superclass to hold common code of two or more existing classes.
- Why not start by designing the inheritance hierarchy?

What's Inheritance for?

1. AI folks: classification (is-a hierarchy)
   - a *Car* is-a *Vehicle*, *Mammal* is-an *Animal*
2. In programming languages: inheritance shares implementation
   - A *CodeEditor* is-implemented-like a *TextEditor*
3. C++, Java and Eiffel say: inheritance specifies subtyping (and 2 above)
   - a LinkedList can-be-substituted-for a Collection

What's Inheritance for?

- What's a programmer to do?
  - If you start by designing the is-a hierarchy, you will find that it conflicts with code sharing.
  - You can't start with code sharing, because you don't yet have any code
  - Ignore code sharing?
- Kent's advice: write code, then refactor

Inheritance ≠ Subtyping

- Specializing a class though inheritance does not in general produce a subtype (substitutable type)
  - ✦ Adding methods is OK
  - ✦ Specializing results is OK
  - ✦ Specializing arguments is *not* OK
- What's a programmer to do?

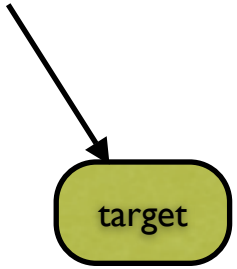
Delegation

- 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 an inst var *form*, the bit mask responsible for actually drawing on the display.
  - ✦ e.g., a *Text* contains a *String*

What about **self**?

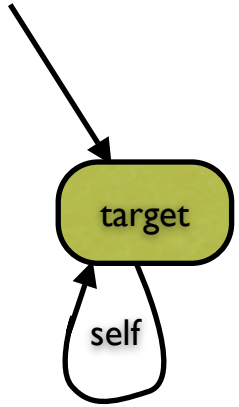
- When you delegate, the receiver of the delegating message is no longer the target
  - ✦ Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, delegate messages unchanged

What about **self**?



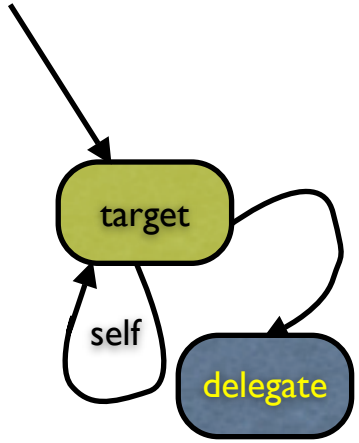
- When you delegate, the receiver of the delegating message is no longer the target
  - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, delegate messages unchanged

What about **self**?



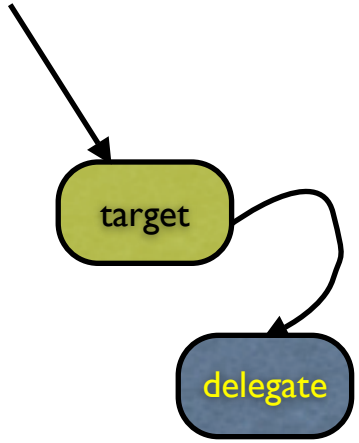
- When you delegate, the receiver of the delegating message is no longer the target
  - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, delegate messages unchanged

What about **self**?



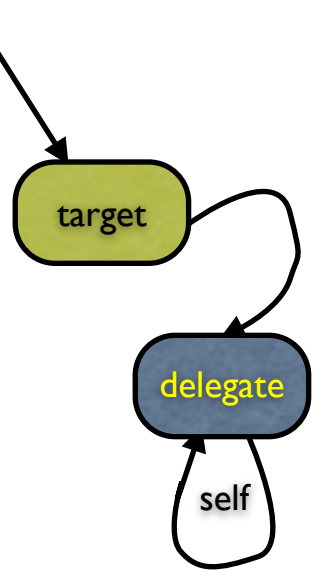
- When you delegate, the receiver of the delegating message is no longer the target
  - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, delegate messages unchanged

What about **self**?



- When you delegate, the receiver of the delegating message 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, delegate messages unchanged

What about **self**?



- When you delegate, the receiver of the delegating message 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, delegate messages unchanged

Simple Delegation Example

- Path >> **do: aBlock**

```
collectionOfPoints do: aBlock
```

- Path >> **collect: aBlock**

```
| newPath |
newPath ← self species new: self size.

newPath form: self form.
newPath points:
        (collectionOfPoints collect: aBlock).
↑newPath
```

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>>at: keyObject put: valueObject
  self hashTable
    at: keyObject
    put: valueObject
    for: self

- HashTable>>at: keyObject put: valueObject for: aCollection
  | hash |
  hash ← aCollection hashOf: keyObject.

- Dictionary>>hashOf: anObject
  ↑anObject hash

- IdentityDictionary>>hashOf: anObject
  ↑anObject basicHash

Pluggable Behavior

- Usually, instances of a class
  - ✦ share the same behavior…
  - ✦ but have different state
- Pluggable Behavior lets them have different behavior:

```
PluggableButtonMorph >> performAction
        self model perform: self actionMessage
```

Pluggable Behaviour

ActionButton
... instanceVariableNames: ' … action … '

This class represents a button that gives a user the opportunity to define an action associated with the mouseDown event.

```
ActionButton >> action: aBlock
    action ← aBlock

ActionButton >> mouseDown: anEvent
    action value
```

Pluggable Block Example

- Cannon >> initialize

```
fireButton ← ActionButton

        withAction: [self loadAndFire]
        andLabel: 'Fire'
```