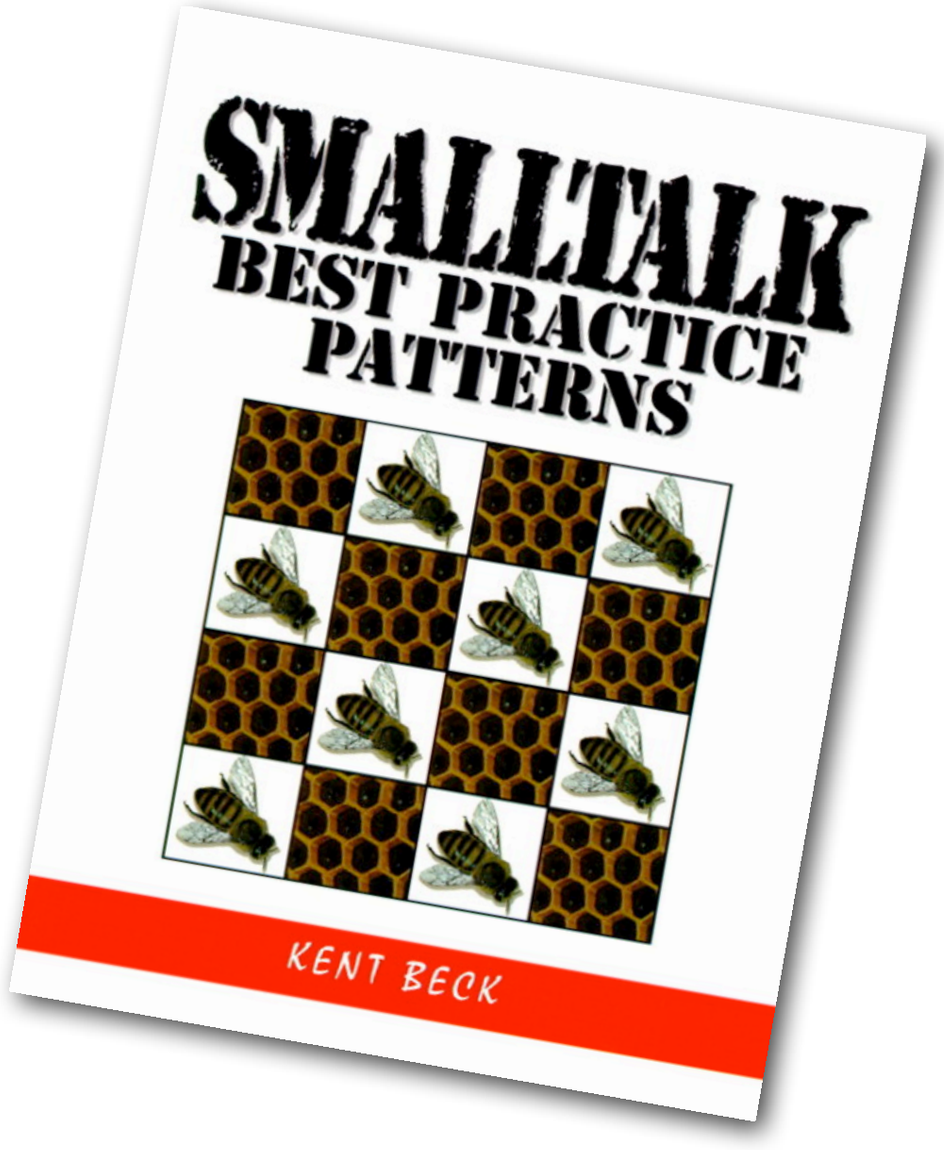
Smalltalk
Best Practice Patterns

Part I

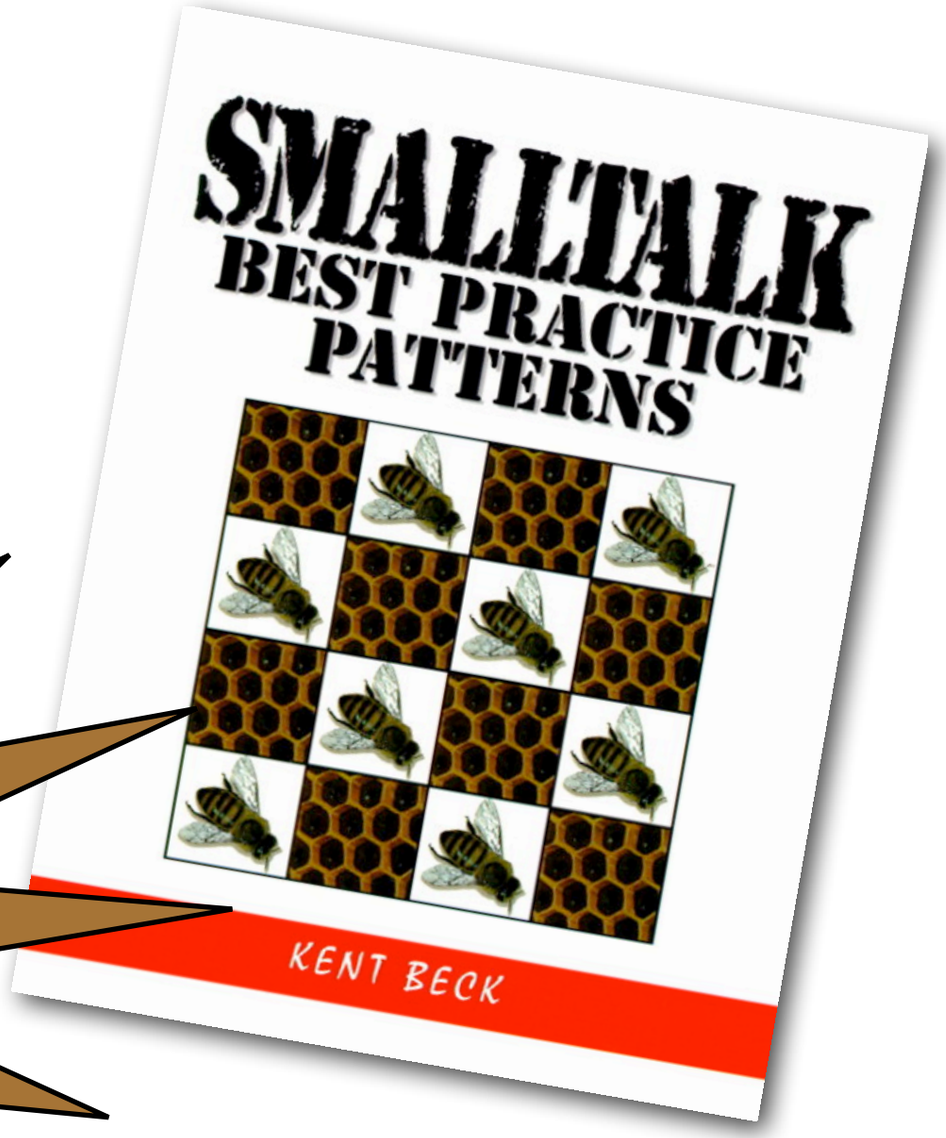
Based on the Book by …

Kent Beck



Based on the Book by …

Kent Beck



Very little here is Smalltalk-specific

Why Patterns?

Why Patterns?

- There are only so many ways of using objects
  - many of the problems that you must solve are independent of the application domain
  - patterns record these problems and successful solutions

Why Patterns?

- There are only so many ways of using objects
  - many of the problems that you must solve are independent of the application domain
  - patterns record these problems and successful solutions
- Remember: the purpose of eduction is to save you from having to think

What's hard about programming?

- Communicating with the computer?
  - not any more!
  - we have made real progress with languages, environments and style

- Communicating with other software developers!
  - 70% of the development budget is spent on "maintenance"
    - discovering the intent of the original programmers

How to improve communication

- Increase bandwidth
  - within the development team
  - between the team and the re-users
- Increase information density
  - say more with fewer bits
  - make our words mean more

A Pattern is:

- A literary form for capturing “best practice”
- A solution to a problem in a context
- A way of packing more meaning into the bytes of our programs

Patterns exist …

- At many levels:
  - Management Patterns
  - Architectural Patterns
  - Design Patterns
  - Programing Patterns
  - Documentation Patterns

Patterns exist …

- At many levels:
  - Management Patterns
  - Architectural Patterns
  - Design Patterns
  - **Programing Patterns**
  - Documentation Patterns

Behavioral Patterns

- *Objects Behave!*
  - Objects contain both state and behavior
  - *Behavior* is what you should focus on getting right!

Patterns for Methods

- Composed Method
- Complete Creation Method
- Constructor Parameter Method
- Shortcut Constructor Method
- Conversion
- Converter Method
- Converter Constructor Method
- Query Method
- Comparing Method
- Execute Around Method
- Debug Printing Method
- Method Comment

Composed Method

How do you divide a program into methods?

- ➡ *Each method should perform one identifiable task*
- ➡ *All operations in the method should be at the same level of abstraction*
- ➡ *You will end up with many small methods*

Complete Creation Method

How do you represent instance creation?

- ➡ *Don't: expect your clients to use new and then operate on the new object to initialize it.*
- ➡ *Instead: provide methods that create full-formed instances. Pass all required parameters to them*
  - Put creation methods in a protocol called *instance creation*

Non-example:

➡ *Point new x:10; y:20; yourself*

Example:

➡ *Point x:10 y:20*

Constructor Parameter Method

You have a constructor method with parameters. How do you set the instance variables of the new object?

➡ *Define a single method that sets all the variables. Start its name with "set", and follow with the names of the variables*

- Put constructor parameter methods into the *private* protocol
- Answer **self** explicitly (INTERESTING RETURN VALUE)

Why not use the ordinary setter methods?

- ➡ *Once and Only Once*
- ➡ *Two circumstances:*
  - initialization
  - state-change during computation
- ➡ *Two methods*

Shortcut Constructor Methods

What is the external interface for creating a new object when a Constructor Method is too wordy?

➡ *Represent object creation as a method on one of the arguments.*

- Add no more than three such shortcut constructor methods per system!
- Examples: 20@30, key->value,
  20@30 extent: 10@10
- Put shortcut constructor methods into the *converting* protocol

Conversion

How do you convert information from one object's format to another?

- *Don't: add all possible protocol to every object that may need it*
- *Instead: convert from one object to another*
  - If you convert to an object with similar responsibilities, use a CONVERTER METHOD.
  - If you convert to an object with different protocol, use a CONVERTER CONSTRUCTOR METHOD

Converter Method

How do you represent simple conversion of another object with the same protocol but a different format?

Kent Beck tells a story …

➡ *If the source and the destination share the same protocol, and there is only one reasonable way to do the conversion, then provide a method in the source object that converts to the destination.*

➡ *Name the conversion method "asDestinationClass"*

- examples: Collection ›› asSet,
  Number ›› asFloat, but *not* String ›› asDate

Converter Constructor Method

How do you represent the conversion of an object to another with a different protocol?

➡ *Make a constructor method that takes the object-to-be-converted as an argument*

- Put Converter Constructor Methods in the *instance creation* protocol
- Example: Date class ›› fromString:

Query Method

How do you represent the task of testing a property on an object?

What should the method answer?

What should it be named?

➡ *Provide a method that returns a Boolean. Name it by prefacing the property name with a form of "be"—is, was, will, etc.*

Examples:

➡ *Switch ›› on*
status := #on
Switch ›› off
status := #off

Switch ›› status
^ status

Examples:

➡
```
Switch ›› on
        status := #on
Switch ›› off
        status := #off
Switch ›› status
        ^ status
```

➡
```
Switch ›› turnOn
        status := #on
Switch ›› turnOff
        status := #off
Switch ›› isOn
        ^ status = #on
Switch ›› isOff
        ^ status = #off
```

Comparing Method

How do you order objects with respect to each other?

➡ *Implement < to answer true if the receiver should be ordered before the argument, and = to answer true if the objects are equal.*

- Put comparing methods into a protocol called *comparing*

➡ *Implement* < and = *only if there is a single overwhelming way to order the objects*

Execute Around Method

How do you represent pairs of actions that should be taken together?

- *Open a file — close a file*
- *Acquire a lock — release a lock*

Obvious solution: make both methods part of the protcol

- *File » open Stream » close*
- *Lock » acquire Lock » release*

What's wrong with that?

Clients are responsible for "getting it right"

How should they know?

Solution

Code a method that takes a block as an argument.

Name the method by appending “During: aBlock” to the name of the first method

```
File » openDuring: aBlock
    | s |
    s := self open.
    aBlock value: s.
    s close
```

Solution

Code a method that takes a block as an argument.

Name the method by appending “During: aBlock” to the name of the first method

```
File » openDuring: aBlock
    | s |
    s := self open.
    aBlock value: s.
    s close
```

```
File » openDuring: aBlock
  | s |
  s := self open.
  aBlock value: s
          ensure: [s close]
```

Which protocol?

Put Execute Around methods in the protocol that contains the methods that they encapsulate

➡ *so openDuring: goes in the “opening” protocol, along with open*

Reversing Method

A composed method may be hard to follow because messages are going to too many receivers

- **Point››printOn:** aStream
  x printOn: aStream.
  aStream nextPutAll: '@'.
  y printOn: aStream.

➡ *How do you code a smooth flow of messages?*

- **Point››printOn:** aStream
  x printOn: aStream.
  aStream nextPutAll: '@'.
  y printOn: aStream.

Why isn’t this smooth?

➡ *We want to think of the method as doing three things to aStream. But, that's not what it says!*

- **Point››printOn: aStream**

```
x printOn: aStream.
aStream nextPutAll: '@'.
y printOn: aStream.
```

Why isn’t this smooth?

➡ *We want to think of the method as doing three things to aStream. But, that's not what it says!*

Instead:

- **Point››printOn:** aStream

```
aStream print: x.
aStream nextPutAll: '@'.
aStream print: y.
```

- **Point››printOn: aStream**

```
x printOn: aStream.
aStream nextPutAll: '@'.
y printOn: aStream.
```

Why isn't this smooth?

➡ *We want to think of the method as doing three things to aStream. But, that's not what it says!*

Instead:

- **Point››printOn:** aStream

```
aStream
    print: x;
    nextPutAll: '@';
    print: y
```

Method Object

Method Object

What do you do when COMPOSED METHOD doesn't work?

Method Object

What do you do when COMPOSED METHOD doesn't work?

Why doesn't it work?

Method Object

What do you do when COMPOSED METHOD doesn't work?

Why doesn't it work?

➡ *many expressions share method parameters and temporary variables*

Beck:

- *“This was the last pattern I added to this book. I wasn't going to include it because I use it so seldom. Then it convinced an important client to give me a really big contract. I realized that when you need it, you really need it”*

The code looked like this:

- **Obligation ›› sendTask:** aTask **job:** aJob
  | notProcessed processed copied executed |
  … 150 lines of heavily commented code …

What happens when you apply COMPOSED METHOD?

- **Obligation ›› sendTask:** aTask **job:** aJob
  | notProcessed processed copied executed |
  … 150 lines of heavily commented code …

Turn the method into a class:

```
Object subclass: #TaskSender
        instanceVariableNames: 'obligation task job
        notProcessed processed copies executed'
```

- *Name of class is taken from original method*
- *original receiver, parameters and temp become instance variables*

new class gets a CONSTRUCTOR METHOD

```
TaskSender class ›› obligation: anObligation task:
aTask job: aJob
    ^ self new
        setObligation:  anObligation
        task: aTask
        job: aJob
```

and the CONSTRUCTOR PARAMETER METHOD

Put the original code in a compute method:

TaskSender››compute

… 150 lines of heavily commented code …

- Change aTask (parameter) to task (instance variable) *etc*.
- Delete the temporaries

Change the original method to use a TaskSender:

◦ **Obligation ›› sendTask:** aTask **job:** aJob

```
^ (TaskSender obligation: self task: aTask job: aJob)
    compute
```

Now run the tests

Now apply COMPOSED METHOD to the 150 lines of heavily commented code.

- *Composite methods are in the TaskSender class.*
- *No need to pass parameters, since all the methods share instance variables*

Beck:

➡ *“by the time I was done, the compute method read like documentation; I had eliminated three of the instance variables, the code as a whole was half of its original length, and I’d found and fixed a bug in the original code.”*

Debug Printing Method

How do you code the default printing method?

- ➡ *Smalltalk provides a way of presenting any object as a String*
- ➡ *printOn: is there for you, the programmer*
  - other clients get their own message

Converting Objects to Strings

There are now four **getters** defined in trait Object for converting an Object to a String:

Show ASCII

```
getter asString(): String      (* for normal use *)
getter asDebugString(): String      (* for debugging; may contain more information *)
getter asExprString(): String      (* when considered as Fortress expression, will equal self *)
getter toString(): String      (* deprecated *)
```

In the trait, all of the other methods are defined in terms of `asString`, so `asString` is the principal method that you should override when you create a new trait. Frequently, programmers write a method that emits more information about the internal structure of an object to help in debugging. If you do that, make it a **getter** and call it `asDebugString`.

`asExprString` is intended to produce a fortress expression that is equal to the object being converted.

Examples

The automatic conversion to String that takes place when an object is concatenated to a String uses `asString`.

The `assert(a, b, m ...)` function uses `asDebugString` to print `a` and `b` when $a \neq b$

Here are the results of using the three getters on the same string:

```
asString:      The word "test" is overused
asExprString:  "The word \"test\" is overused"
asDebugString: BC27/1:
                       J15/0:The word "test"
                       J12/0: is overused
```

Here they are applied to the range 1:20:2

```
asString:      [1,3,5,7,... 19]
asExprString:  1:19:2
asDebugString: StridedFullParScalarRange(1,19,2)
```

Method Comment

How do you comment a method?

- *Communicate important information* that is not obvious from the code *in a comment at the beginning of the method*

How do you communicate what the method does?

- INTENTION-REVEALING SELECTOR

…what the arguments should be?

- TYPE-SUGGESTING PARAMETER NAME

…what the answer is?

- other method patterns, such as QUERY METHOD

…what the important cases are?

- Each case becomes a separate method

What's left for the method comment?

Method Comment

How do you comment a method?

- ➡ *Communicate important information* that is not obvious from the code *in a comment at the beginning of the method*

Between 0% and 1% of Kent's code needs a method comment.

- ➡ *use them for method dependencies, to-do's,* reason *for a change*

But:

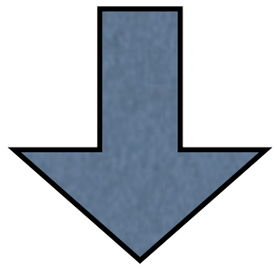
- ➡ *method dependencies can be represented by an EXECUTE-AROUND METHOD*
- ➡ *to-do's can be represented using the self flag: message*

Useless Comment

show

(self flags bitAnd: 2r1000) = 1 "am I visible"

ifTrue: […]



isVisible

^ (self flags bitAnd: 2r1000)

show

self isVisible ifTrue: […]

Message Patterns

Message

- Conditional code:
  - do this or do that, depending
- Encapsulation:
  - do that code over there
- Message-send
  - do this code over there, or that code over yonder, I don’t really care

Message-send replaces conditional

- You are building a complex tool. You find that it behaves the right way for “green” objects, but not for “blue” objects.
- What to do?

```
target isGreen
    ifTrue: [ target doExistingThing ]
    ifFalse: [ target doNewThing ]
```

Message-send replaces conditional

- You are building a complex tool. You find that it behaves the right way for “green” objects, but not for “blue” objects.
- What to do?

```
target isGreen
    ifTrue: [ target doExistingThing ]
    ifFalse: [ target doNewThing ]
```

Message-send replaces conditional

- You are building a complex tool. You find that it behaves the right way for “green” objects, but not for “blue” objects.
- What to do?

~~target isGreen~~
~~ifTrue: [target doExistingThing]~~
~~ifFalse: [target doNewThing]~~

Message-send replaces conditional

- You are building a complex tool. You find that it behaves the right way for “green” objects, but not for “blue” objects.
- What to do?

```
target doAppropriateThing

Green » doApproriateThing
    self doExistingThing

Blue » doApproriateThing
    self doNewThing
```

This is the most important lesson of the quarter

Portland State
UNIVERSITY

Take this lesson to heart

- Whenever you discover that a method is making a choice, ask yourself
  - is it doing a single abstract action?
- If so, invent a name for that action
  - a message
- tell an object to do it
  - send that message to the object
- respond appropriately
  - code methods on the receiving objects

Example

```
BrowserNameMorph » onClick
    self representedClass showDefinition
```

Example

Example

```
BrowserNameMorph » onClick
    self representedClassOrTrait showDefinition
```

Example

```
BrowserNameMorph » onClick
    self representedClassOrTrait showDefinition
```

- Problem: showDefinition is the right behavior if I represent a class, but not if I represent a trait.

Wrong Solution

```
BrowserNameMorph » onClick
    | ct |
    ct := self representedClassOrTrait.
    ct isClass
        ifTrue: [ ct showDefinition ]
        ifFalse: [ ct showSubtraits ]
```

Right Solution: Choosing Message

- Think of a good name for what is to be done
  - send that message
  - implement two methods in the receiving classes

```
BrowserNameMorph » onClick
    self representedClassOrTrait showStructure

ClassMorph » showStructure
    self showDefinition

TraitMorph » showStructure
    self showSubtraits
```

- Sometimes even when beginners have several kinds of objects they still resort to conditional logic:

```
responsible := (anEntry isKindOf: Film)
                    ifTrue: [anEntry producer]
                    ifFalse: [anEntry author]
```

- Code like this can always be transformed into communicative, flexible code by using a Choosing Message:

```
Film»responsible ^self producer

Entry»responsible ^self author
```

- Now you can write:

```
responsible := anEntry responsible
```

- but you probably don't need the EXPLAINING TEMPORARY VARIABLE any more.

Decomposing Message

- Send messages to self to break a computation into little pieces
- Most Smalltalk methods are 3 or 4 lines long — certainly less than 10
- Why?
  - Smalltalk's development tools allow programmers to be productive with small code fragments
  - Smalltalk gives the programmer higher-level abstractions

- don’t write:

```
sum := 0.
1 to: collection size
    do: [ : i | sum := sum + (collection at: i)]
```

- write:

```
collection sum
```

Intention Revealing Message

- You are sending a message to invoke a *really simple* computation. How do you communicate your intent?
- Send a message that communicates *what* you want to do (*not* how it is accomplished)

collection isEmpty

number reciprocal

color darker

Intention Revealing Message

- Write a simple method to implement your message

```
Collection » isEmpty
        ^ self size = 0

Number » reciprocal
        ^ 1 / self

Color » darker
        ^ self adjustBrightness: -0.08
```

Intention Revealing Selector

- How do you name a method?
  - Name it after *how* it accomplishes its task
  - Name it after *what* it is supposed to accomplish
    - leave the “how” for the body of the method
  - Examples:

    Array»linearSearchFor:
    Set»hashedSearchFor:
    BTree»treeSearchFor:

Intention Revealing Selector

- How do you name a method?
  - Name it after *how* it accomplishes its task
  - Name it after *what* it is supposed to accomplish
    - leave the “how” for the body of the method
  - Examples:

~~Array»linearSearchFor:~~
~~Set»hashedSearchFor:~~
~~BTree»treeSearchFor:~~

Intention Revealing Selector

- How do you name a method?
  - Name it after *how* it accomplishes its task
  - Name it after *what* it is supposed to accomplish
    - leave the “how” for the body of the method
  - Examples:

~~Array»linearSearchFor:~~
~~Set»hashedSearchFor:~~
~~BTree»treeSearchFor:~~

Collection»includes:

- Not so easy to apply when you have just one implementation
  - Imagine a second, very different implementation
    - Would you give it the same name?
      - if so, the name is probably “sufficiently abstract” — for now

Programming Patterns for Reuse

Review: Complete Creation Method

Review: Complete Creation Method

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

Review: 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

Review: 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 one place

Once and Only Once

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

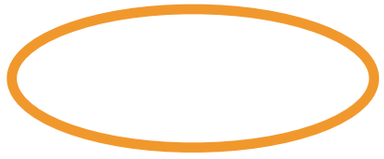
```
Window class » withTitle: aTextOrString
    ↑ Window new title: aTextOrString;
      yourself
```

```
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 its 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 objects, subclasses of the general case:
  - 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 a very 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.

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-revealing selector
- Dispatch again on what was the argument

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

  - Revealed in a famous paper: “A Simple Technique for Handling Multiple Polymorphism” by Ingalls, OOPSLA ’86

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?

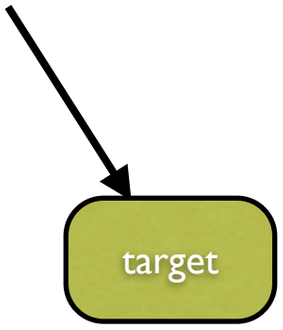
Delegation

- 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**?

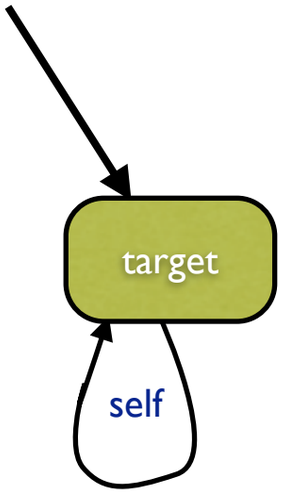
- When you delegate, the receiver of the delegating message is no longer the target
  - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn’t matter, delegate messages unchanged

What about **self**?



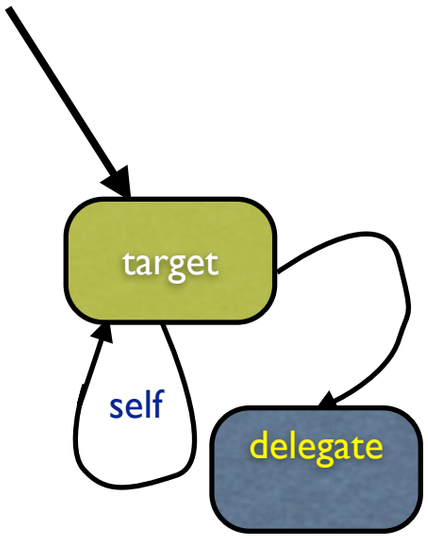
- When you delegate, the receiver of the delegating message is no longer the target
  - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn't matter, delegate messages unchanged

What about **self**?



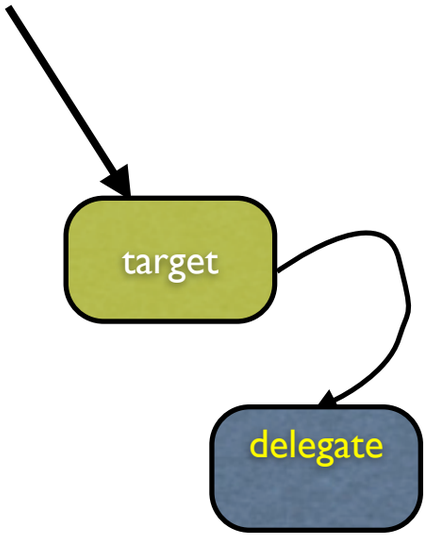
- When you delegate, the receiver of the delegating message is no longer the target
  - Does it matter? Does the delegate need access to the target? Does the delegate send a message back to the client?
- If it doesn’t matter, delegate messages unchanged

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

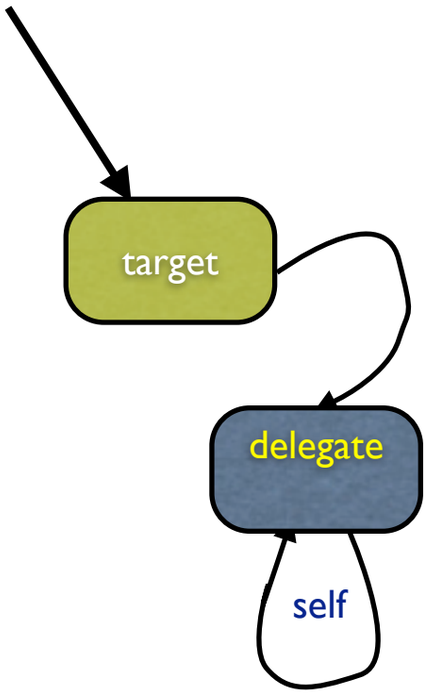
What about **self**?



- 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'
```