Smalltalk
Best Practice Patterns

Part II

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

What do you do when COMPOSED METHOD doesn't 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.”*

Execute Around Method

How do you represent a pair of actions that have to be taken together?

- *open a file … close a file*
- *push a context … pop a context*
- *acquire a lock … release a lock*

What's wrong with making both operations part of your protocol?

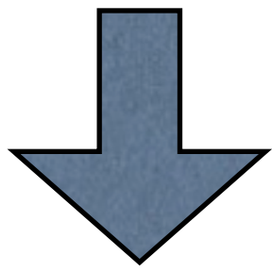
➡ *Write a method that takes a block as a parameter.*

- Name the method by appending *During: aBlock* to the name of the first message
- In the body, send the first message, evaluate the block, and send the second message

```
- Lock  ›› acquireDuring: anActionBlock
        self acquire.
        anActionBlock value.
        self release.
```

Better: use an ensure: message

```
Lock  ›› acquireDuring: anActionBlock
        self acquire.
        anActionBlock value.
        self release.
```



```
Lock  ›› acquireDuring: anActionBlock
        [self acquire.
        anActionBlock value] ensure: [self release]
```

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  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.

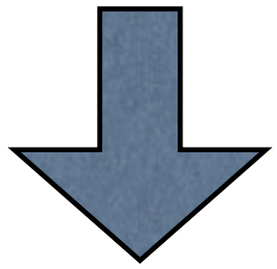
- ➡ *use them for method dependencies, to-do's,* reason *for a change*

But:

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

Useless Comment

```
(self flags bitAnd: 2r1000) = 1 "am I visible"
        ifTrue: [ … ]
```



```
isVisible
    ^ (self flags bitAnd: 2r1000)

self isVisible ifTrue: [ … ]
```