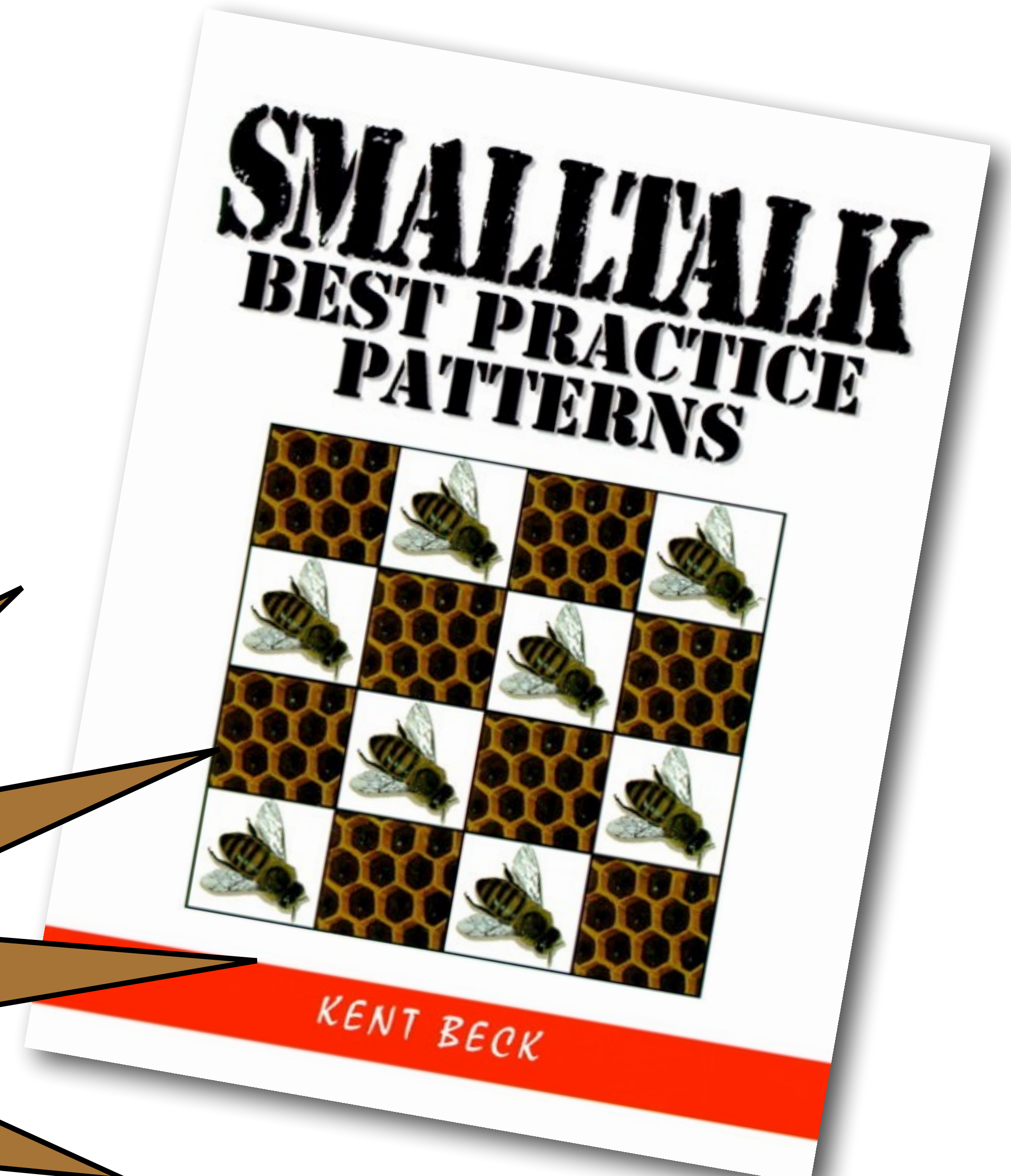
Grace

Best Practice Patterns

Based on the Book by …

Kent Beck



Very little here is Smalltalk-specific

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

Portland State UNIVERSITY

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!
- Sandi Metz:
  - You don't send messages because you have objects, you have objects because you send messages

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 factory methods that create fully-formed instances. Pass all required parameters to them*

Grace note:

- it doesn't matter to the *computer* whether you use a class, or a method returning an object: they do the same thing: create and answer a new object.
- Are all methods that return objects **class**es?
  - No. Many methods that returns objects get a new object from a class, manipulate it in some way, and then return it
  - I'm going to refer to methods that returns new(ish) objects as *factory methods*.

Non-example:

- ***def** nyssa = dog.new*
  nyssa.name := "Nyssa"
  nyssa.breed := mutt
  nyssa.sound := whine

Example:

- ***def** nyssa = dog.named "Nyssa"*
  breed (mutt) sound (whine)

Why not use the ordinary setter methods?

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

Portland State UNIVERSITY

Shortcut Constructor Methods

What is the external interface for creating a new object when a Factory 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, 1..10

Builder
(from GoF p97, Alpert p47)

How do you construct an object when there are a multitude of possible initialization options?

➡ *too many combinations to have one factory method for each*

Solution: use a builder object to collect the options

Example of Builder

```
def carBuilder := fordBuilder
carBuilder.add2doorSedanBody
carBuilder.add6CylinderEngine
carBuilder.addLeatherBucketSeats
…
def car = carBuilder.result
```

- Note that the builder doesn't return anything interesting; it collects the options internally, and answers the new object when asked for its *result*
- Conventionally, builders return *self*

Portland State UNIVERSITY

Returning **self** enables "chaining"

def carBuilder = fordBuilder.add2doorSedanBody.
add6CylinderEngine.addLeatherBucketSeats
…
def car = carBuilder.result

or even

def car = fordBuilder.add2doorSedanBody.
add6CylinderEngine.addLeatherBucketSeats
….result

Builders can do more …

- *add parts provide by part factories*
- *choose which pert factories to use based on prior options*
- *transform arguments into new parts*
- *interpret and abstract specification (UIBuilder)*

An object can be its *own* builder

```
import "graphix" as g
def graphics = g.create(300, 300)
graphics.addCircle.at(100@200).
    colored "red" .filled(true).draw
```

- here addCircle returns a circle-builder ... or does it return a circle?
- We don’t have to know!

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 FACTORY 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 …

For a long time, it bothered me that there was a String » asDate method. I couldn't quite put my finger on what it was that bothered me about it, though. Then, I walked into a project where they had taken the idea of conversion to extremes. Every domain object had twenty or thirty different conversion methods. Every time a new object was added, it had to have all twenty or thirty methods before it would start working with the rest of the system.

One problem with representing conversion as methods in the object to be converted is that there is no limit to the number of methods that can be added. … Another is that it [*couples*] the receiver, however tenuously, with a class of which it would otherwise be oblivious.

I avoid the protocol explosion problem by representing conversions with a message to the object to be converted *only* when:

- The source and destination of conversion share the same protocol
- There is only one reasonable way to do the conversion.

Converter Method Pattern

- ***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 "asDestinationType"*
  - examples: aList.asSet, aSequence.asDictionary, but *not* aString.asDate

Converter Factory Method

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

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

- Put Converter Factory Methods in the same object as the other instance creation method for that class of object.
- Example: aDate.fromStringMonthFirst(s: String)
  aDate.fromStringYearFirst(S: String)

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:

- **var** *status* ***is*** *readable == "on"*
- ***method*** *on { status := "on" }*
- ***method*** *off { status := "off" }*

Examples:

- ~~**var** *status* ***is*** *readable == "on"*~~
- ~~***method*** *on { status := "on" }*~~
- ~~***method*** *off { status := "off" }*~~

- **var** status := "on"
  method turnOn { status := "on" }
  method turnOff { status := "off" }
  method isOn { status == "on" }
  method isOff { status == "off" }

1. It's now clear from the names which methods *test* the state, and which *change* it.

Example:

```
➡ var status := "on"
  method turnOn { status := "on" }
  method turnOff { status := "off" }
  method isOn   { status == "on" }
  method isOff  { status == "off" }
```

2. This encapsulates the representation of the state — reducing coupling. I can change the representation without clients needing to know:

```
➡ var stateIsOn := true
  method turnOn { stateIsOn := true }
  method turnOff { stateIsOn := false }
  method isOn   { stateIsOn }
  method isOff  { stateIsOn.not }
```

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.
- ➡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 protocol

- ➡*file.open -> Stream   aStream.close*
- ➡*lock.acquire   lock.release*

Portland State UNIVERSITY

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

```
method openDuring(aBlock)
 def s = self.open
 aBlock.apply(s)
 s.close
```

Solution

```
method openDuring(aBlock) {
   def s = self.open
   aBlock.apply(s)
   s.close
}
```

Even better:

```
method openDuring(aBlock) {
    try {
        def s = self.open
        aBlock.apply(s)
     } finally { s.close }
}
```

Reversing Method

A composed method may be hard to read because requests are made of too many receivers

```
method printOn(aStream) {
    x.printOn(aStream)
    aStream.append "@"
    y.printOn(aStream)
}
```

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

```
method printOn(aStream) {
    x.printOn(aStream)
    aStream.append "@"
    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!*

```
method printOn(aStream) {
    x.printOn(aStream)
    aStream.append "@"
    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:

```
method printOn(aStream) {
    aStream.append(x)
    aStream.append "@"
    aStream.append(y)
}
```

Instead:

```
method printOn(aStream) {
    aStream.append(x)
    aStream.append "@"
    aStream.append(y)
}
```

or even:

```
method printOn(aStream) {
 aStream
     .append(x)
     .append "@"
     .append(y)
}
```

assuming that append answers **self**

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 (on *obligation* objects) looked like this:

```
method sendTask(aTask) job(aJob) {
    def notProcessed = list.empty
    def processed = list.empty
    var copied
    var executed
        … 150 lines of heavily commented code …
}
```

What happens when you apply COMPOSED METHOD?

```
method sendTask(aTask) job(aJob) {
    def notProcessed = list.empty
    def processed = list.empty
    var copied
    var executed
    … 150 lines of heavily commented code …
}
```

Turn the method into a object!

➡ *define a local class:*

```
class taskSender (obligation, aTask, aJob) {
    …
}
```

- *Name the class on the original method*
- *original receiver and parameters become* ***parameters*** *of the class*
- *all of the method temporaries become* ***fields*** *of the object*

Portland State UNIVERSITY

the new C*LASS* *initializes the* **fields:**

```
class taskSender(obligation, task, job)
    def notProcessed = list.empty
    def processed = list.empty
    var copied := …
    var executed := …
    ...
}
```

Put the original code in a compute method:

```
method compute {
  … 150 lines of heavily commented code …
}
```

- code that previously referred to method parameters now refers to class parameters!
- code that previously referred to method locals now refers to the object’s fields

Change the original method in the obligation object to use a TaskSender:

```
method sendTask(aTask) job(aJob) {
    taskSender(self, aTask, aJob).compute
}
```

Now run the tests

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

➡ *Composite methods are in the taskSender object.*

➡ *No need to pass parameters*

- all the methods share instance variables, and
- can access class parameters

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.”*

String Conversion Methods
was: DEBUG PRINTING METHOD

- Converting objects to strings is powerful:
  - strings fit nicely into generic interfaces, like menus, tables, and text editors
  - strings are useful to the programmer: they should tell you most of what you need to know about an object to diagnose a problem
- Grace provides two ways of presenting any object as a String
  - *asDebugString* is there for you, the programmer
  - *asString* is there for client objects

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*

Portland State University

How do you communicate what the method does?

- INTENTION-REVEALING METHOD NAME

…what the arguments should be?

- type annotations in the method header

…what the answer is?

- other method patterns, such as QUERY METHOD
- type annotation in the header

…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,* *TODOs*, reason *for a change*

But:

- ➡ *method dependencies can be represented by an EXECUTE-AROUND METHOD*
- ➡ *TODOs can be represented using a request like* flag "message"

Useless Comment

show

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

isVisible

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

show

```
self isVisible ifTrue: [ … ]
```