Spoon
a dynamic object system

Craig Latta
the NetJam project

draft six
estimated length: 500 pages

preface

I have been an avid Smalltalk[1] developer for many years. When I first encountered it, the Smalltalk system was rather mysterious and hard to get. I was lucky enough to come across a copy of the “blue book”[2] in college; in its pages I imagined what it might be like to build a system with *objects*. For me, the message-oriented syntax was more than clean and compact, it seemed to give the objects distinct personalities. This made it very easy for me to visualize collaborating sets of objects, and to keep their roles organized in my mind. I was intrigued.

When I started building real systems, I came to appreciate the *dynamic* nature of the environment. The ability to fix problems in the system without restarting it enabled new approaches to problem-solving that weren’t possible otherwise. I could also halt the system on one computer and resume it on another, without worrying about operating systems and processors. Smalltalking was a lot of fun. I was becoming addicted.

---

[1] If you’ve never encountered Smalltalk before, feel free to skip to the end of this preface.

[2] *Smalltalk-80: The Language and Its Implementation* by Adele Goldberg and David Robson

Like many new Smalltalk enthusiasts, I began to wonder: *Why isn't Smalltalk more popular? What's holding it back?* At the time, the most widespread programming language was C. As a university student, it seemed clear to me why it had succeeded. It was easy to get! Its owners had a relatively liberal distribution policy, and this enabled a thriving community of academic hackers who used it. When these people graduated, they would enter industry with both C experience and a desire to keep using it. Furthermore, the 1970s, when C was introduced, were a special time in computing history. There were many programming languages in use, none particularly dominant. It was easier to assert one's technical preferences.

The designers of Smalltalk clearly knew the value of "getting them young"; Alan Kay, the creator of the project, had the improvement of primary education in mind. But Smalltalk's corporate owners never committed to this vision. They were only interested in Smalltalk to the extent that it enhanced their traditional business, and seemed unable to envision new ones.[3] The blue book was about all the world was going to get from them. The young minds for whom Smalltalk was intended would have to wait.

In the meantime, I started to notice technical problems with Smalltalk that hindered its usefulness, quite apart from its political limitations. Generally, these problems were simply signs of the system's immaturity. As advanced as it was, Smalltalk left the research lab before its implementors had a chance to provide several critical facilities. There were also constraints placed on the system by the state of computing hardware at the time.

One of the wonderful things about Smalltalk is that is was designed to be used *and understood* by a single person[4]. Unfortunately, this emphasis seems to have led to a certain myopia where collaborative development is concerned. There are some measures of support for collaboration that are fundamental to the system design, and are

---

[3] see *Fumbling the Future* by Douglas K. Smith and Robert C. Alexander

[4] see *Design Principles Behind Smalltalk* by Dan Ingalls, BYTE magazine, August 1981

not well done as afterthoughts. I like to think that, given a few more turns of the observe-formulate-test cycle that Ingalls mentions in his August 1981 BYTE article, the Smalltalk team would have addressed some of these issues. As it is, some unfortunate design decisions seem to have become entrenched in subsequent incarnations of Smalltalk.

The most problematic of these is a reliance on *files* to transmit behavior. Despite the Smalltalk maxim that everything is done by sending messages, conveying classes and methods between systems is traditionally done by writing source code to a file (a “fileout”), then reading the file and recompiling the source code. Another wonderful thing about Smalltalk is its reflective implementation; it is built using the very technology it describes. It would be simpler and more direct to take advantage of that quality, by transferring classes and methods directly, using messages sent over a network between machines. There is typically no need to recompile source code, nor to store information about system components outside the system.

In 1983 when Smalltalk was released, processors were relatively slow and network connections were rare (although, ironically, Ethernet was developed at nearly the same time and place). It’s conceivable that sending Smalltalk messages over a network would have been seen as utterly impractical. On the other hand, Smalltalk itself was fairly impractical at the outset; part of the spirit of the project was to figure out how we’d like to use computers and *make* it practical. At any rate, by the mid-eighties distributed operation was feasible by any measure, but by then Smalltalk development had moved from research to the commercial realm.

I missed out on those early research days. By the time I actually got to use a Smalltalk system, I had no time for changing the fundamentals of the system; I had “real work” to do. But the more real work I did, the more acute the system’s shortcomings became. Integration of a team’s work became a dreaded task because of the conflict resolution that it entailed, despite the fact that much of it could have been automated through direct negotiation between the target and source

systems. But even if I had had the time, changing the system wasn't a viable option. Although Smalltalk's object memory has traditionally been very open (obfuscating method source code isn't easy even for the determined), the virtual machine was typically written in a different, lower-level language like C or assembler, and the source code was not included.

Squeak[5] changed this in 1996. With its virtual machine implemented almost entirely in Smalltalk, navigable and executable with familiar Smalltalk tools, conducting experiments with the fundamental elements of the system became easier than it has ever been. Although I switched to Squeak when it was released, I was still too busy to pursue my own research topics. But in 2002, I found myself at a loss for real work, so the fun could begin! I decided it would be most entertaining to turn Squeak into my desired system in a continuous fashion, never losing the unbroken thread of persistent object memories dating back to the 1970s. For the gory details on that adventure, see the appendix.

for those new to objects...

If you've never heard of Smalltalk before, or have never tried programming for that matter, this history may not be very interesting yet. That's fine; this book is not meant as a history lesson, it's an introduction to object-oriented programming and system design, and, more importantly, how much *fun* they can be. I hope you find this book as intriguing as I found the original blue book.

[5] see *Back to the Future The Story of Squeak, A Practical Smalltalk Written in Itself* by Dan Ingalls et al

acknowledgments

I'm grateful for the encouragement and support of many people while I worked on Spoon. Thanks to Alan Kay for creating the Smalltalk idea, to Dan Ingalls, the original virtual mechanic, for his pioneering implementations of Smalltalk, to Adele Goldberg and Dave Robson for the wonderfully lucid blue book (whose structure I loosely emulate here), to Ken Causey and Ken Brown for their early testing help and for being my first real developers, to Jeff Eastman for real code to modularize, to John McIntosh, John Randolph and Jeremy Thorpe for Macintosh help, to Gale Pedowitz and the rest of the Squackers for demo feedback and good cheer, and to Brenda Larcom, Tim Rowledge, Brian Rice, and the entire Squeak community for technical camaraderie and design feedback. Special thanks to my parents, Joan and Milton, for their constant love and confidence during my "lean years".

Craig Latta
Palo Alto, California
June 2007

contents

part one: getting started 13

chapter one: why?.....15

chapter two: installing and starting the system.....21

part two: the memory 25

chapter three: objects and messages.....27

chapter four: syntax.....33

chapter five: the Spoon object model.....41

chapter six: what all objects can do.....43

chapter seven: fundamental constants.....45

chapter eight: magnitudes..... 47

chapter nine: numbers.....49

chapter ten: collections.....51

chapter eleven: streams..... 53

chapter twelve: processes..... 55

part three: tools 57

chapter thirteen: inspectors.....59

chapter fourteen: the class browser.....61

chapter fifteen: the debugger.....63

part four: the processor 65

chapter sixteen: architectural overview..... 67

chapter seventeen: the object memory.....69

chapter eighteen: the instruction set.....71

chapter nineteen: the primitives.....73

chapter twenty: remote messages..... 75

chapter twenty-one: building your own processor... 77

part five: distributed operation 79

chapter twenty-two: imprinting and modules.....81

chapter twenty-three: collaboration..... 88

chapter twenty-four: deployment.....92

appendix A: the Spoon story 94

appendix B: notes on the text 96

index 99

part one: getting started

chapter one: why?

Why program?

I sometimes ask myself this question late at night, in the middle of an intense debugging[1] session. There's something magical about getting a computer to do your bidding. It's certainly nice to watch a movie or listen to music with a computer, but more *fun* to use it to express your own ideas. And it's a special kind of fun, where one is engaged by solving problems and creating new ones. It's *hard fun*[2].

When we use computers for hard fun, we get at Smalltalk's original purpose, the amplification of thought. We fulfill the original vision of the personal computer as an extension of the mind. We also transcend our default societal roles as consumers, becoming producers as well as *collaborators*. This is a powerful form of freedom, in which we are more engaged with the world and its possibilities.

---

[1] problem-solving; search the web for the amusing etymology of this word

[2] see *Hard Fun* by Seymour Papert, an article on the web

Another thing that makes this sort of fun special is that it gets results. When you solve a problem, you can reuse the solution in future situations and use it to solve more complicated problems. The computer then becomes a tool for *inspiration*. Using what you've made will lead you to think of new capabilities you hadn't considered before, and refinements that make your previous solutions better. This is a feedback loop that grows ever more satisfying over time.

Why program with Spoon?

For the computer to be most effective as a tool for thinking, it should complement one's thought processes without distracting from them. Ideally, it will enable a state of flow[3], in which one is fully immersed in an activity, successful with a feeling of effortlessness, frequently losing track of time in the process. I enter this state when I improvise music with other people. My instrument becomes part of me, responsive and suggestive of new possibilities, never getting in the way of my experimentation. The time between "What if...?" and a result is always very short.

So it is with dynamic object-oriented programming. The entire system is *live*, open to modification at all times because it's always running. In the best environments, the system is implemented using the same facilities made available to the programmer[4]. When one can change anything about the system at any time and without having to use different tools or adopt a different mindset, one has the freedom to take an improvisatory approach to programming.

This freedom is vital not only when building a new system, but also when modifying an existing one. Since you never have to stop the running system, you never lose the state you have acquired from running it. This is invaluable in diagnosing why things have gone wrong. With Smalltalk, since the entire system is built around a single concept (objects sending messages to each other), once you have diagnosed

---

[3] search for Mihaly Csikszentmihalyi for historical background on flow in psychology

[4] this is sometimes referred to as being implemented "in itself"

a problem, you can continue a process from the point of failure, or any earlier point. With Spoon, the system you're editing may be on your own computer or your friend's (or, more likely, spread across both).

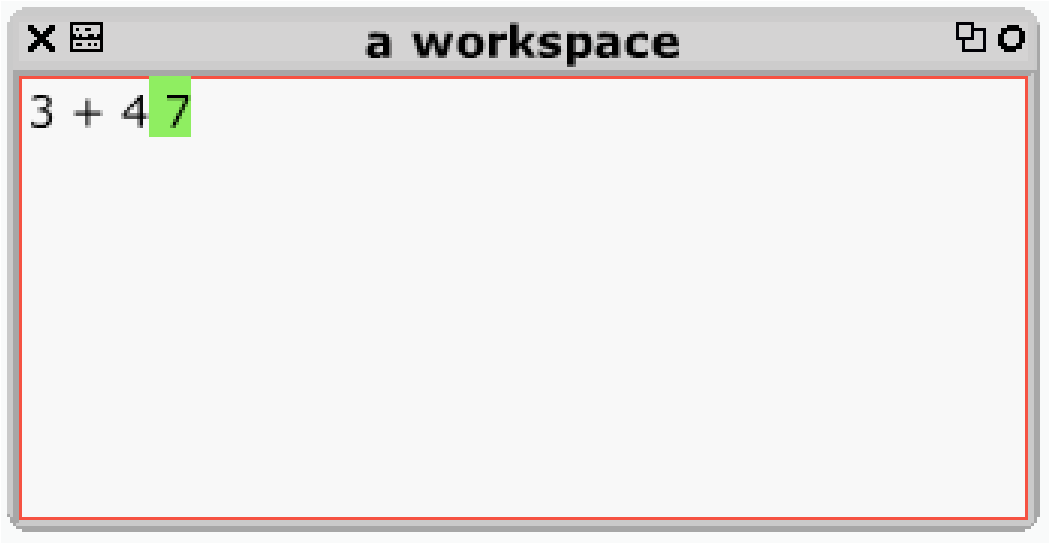
It's enough to make you forget to go to bed and stay up late into the night!

chapter two: installing and starting the system

This is the part that was missing from the blue book... take a moment to bask in how fortunate you are now. Well, do one thing first: visit http://netjam.org/spoon/releases/current/. You can bask during the few seconds it takes to download the current release.

Follow the instructions that came with the release. You'll end up with a Spoon system running on your machine, communicating with you through a web browser:

Click on the link for viewing a list of available modules. Load the **development tools** module. A new window will open. This is a Spoon display, with its own window system. A *workspace* will open within it. Now it's time for another reverent moment: type "3 + 4" into the workspace, select the text, and type meta-p[1].



You've just evaluated your first Spoon *expression*. It happens to be the canonical first test for Smalltalk systems (a surprisingly large portion of the system must work correctly for this to run properly). You've also used your first tool in the Spoon environment, a workspace. Workspaces are like handy pieces of scratch paper, ready for evaluating expressions which don't yet have a formal home. We'll just be using this workspace for a while.

[1] On Macintosh, the meta key is the one with the Apple logo. On Windows and Linux it's the *alt* key.

part two: the memory

chapter three: objects and messages

Now that you've evaluated a Spoon expression, you've used objects. In particular, you've *sent messages to objects*. Everything in the Spoon system is done by sending messages to objects. In the expression you evaluated, you sent the message "+ 4" to the object 3. The object 3 responded with the answer, the object 7. To get the evaluation to happen, you pressed keys on the keyboard, which resulted in messages sent to a keyboard event handler object. To display the answer, the system sent messages to graphics objects.

An object is a combination of *state* and *behavior*. An object's state is a private collection of references to other objects, while its behavior is a set of operations that it can perform. A *message* is a request by an object for another object to perform one of its operations. The object to which a message is sent is the *receiver*; the object sending it is the *sender*. The receiver of a message determines how it will react. There is a fundamental separation here, between the goal expressed by the sender through a message, and the strategy implemented by the receiver in response to that message.

Since sending messages is the only way for objects to interact, the nature of an object is typified by its responses to messages. This is why an object's operations are called its "behavior". Formally, each operation that an object makes available to other objects is a contract, an agreement to perform work according to a shared meaning of the message which invokes it. The union of all the messages to which an object responds is its *message interface*.

With discretion over the implementation of its operations, an object is free to use the most suitable state to represent itself, as long as it fulfills its message interface. For example, the state employed by a integer like 3 will probably differ from that of a fraction. Nevertheless, both objects support the same message for performing addition. With this independence of behavior from representation, we may compose modular systems. One need not know anything about the internal structure of an object to use it; only its behavior is important.

Message-sending is the core idea of object-oriented programming[1]. From now on, we'll discuss tactics for organizing a system around this idea, and for using such a system.

The first organizational question we face is this: where do objects come from? Like everything else, creating new objects is something accomplished by sending messages. There are a set of objects in the Spoon system that support a message interface for creating new objects. These objects are called *classes*. When a class receives an object-creation message, it responds with a new object that is an *instance* of itself. Every object in the system, including the classes themselves, is an instance of some class.

All the instances of a class share the same *format*; that is, the state of each instance is arranged in the same way, although the particular objects to which an instance refers may be different. All the instances of a class also share the same behavior. Both the state format and behavior are defined by the class.

---

[1] Indeed, it might more appropriately be called *message-oriented programming*.

The permanent parts of an object's state are called *instance variables*[2]. Each instance variable refers to some other object in the system. Each instance of a particular class has the same number of instance variables, and the same meanings associated with them. The behavior that a class provides for its instances is composed of *methods*.

When you sent the message "+ 4" to the object 3, you invoked a method, provided by 3's class, for adding a number to the receiver and answering the result. The object 4 is a parameter for that method. It's a *temporary variable* from the receiver's point of view, a temporary part of its state. Generally, a method's activity consists of sending messages to the objects of the receiver's state. A method concludes by *answering* an object (in this case, 7).

Methods determine what happens when you send messages to objects; they are the venue in which objects send messages to each other. You'll spend most of your time reading and writing them. "Programming" with Spoon means:

- defining a desired effect
- identifying the objects for creating that effect
- defining and refining the behavior for those objects

It's important to note that the class/instance aspect of Spoon objects is not fundamental to object-oriented programming; it is simply one way of organizing behavior[3].

For further details on the composition of individual objects, see chapter five. First, though, we'll look at how to refer to objects and send messages to them through expressions; we'll learn how to write methods.

---

[2] also known as *slots*

[3] Another is based on the concept of *prototypes*, in which each object is responsible for its own behavior. See, for example, the Self system.

chapter four: syntax

Let's take one more look at that first expression, "3 + 4". The receiver is the object 3. The message we're sending to 3 is "+ 4". A message has two parts: a *selector*, which is the name of the message, and one or more *parameters*. For this message, the selector is "+" and the parameter is the object 4.

We're going to create our own method, which is just a collection of one or more expressions. The first thing we need is a *goal*; what will this method do? Let's start with something simple, again in the realm of arithmetic. Our goal will be to calculate the square of the receiver.

Given this goal, we need a name for our method. Note the conversational nature of the first selector we encountered. It made the expression, "3 + 4", look like a phrase you might encounter in normal conversation. You can think of an object's selectors as its vocabulary, a list of commands that it understands. Through conversation, we anthropomorphize with which we interact. It is traditional to refer to each object as if it were autonomous, with a will of its own. Objects are very much like characters in a story[1].

[1] Indeed, one object system refers to them as *actors*.

With this rationale, the most appropriate name for a method which answers the square of the receiver is “squared”. This will yield an expression like “3 squared”. The very next thing we need to do is record our method’s goal, so that it will be clear to anyone reading the method later. We do this with a *comment*. Comments are enclosed in double quotation marks. So far, our method looks like this:

```
squared
    “Answer the square of myself.”
```

In keeping with the anthropomorphic nature of objects, the comment is written in the first person, from the receiver’s point of view. Note also that we used the word “answer” instead of “return”, in keeping with our conversational metaphor.

Now we must consider our tactic for answering the square of the receiver, our *algorithm*. As you probably remember from grade school arithmetic, the square of a number is simply the product of the number with itself. We’ve already seen how to add two numbers together; multiplying them is similar. The selector for the method for multiplication is “*”. What we need to know now is how to refer to the receiver itself.

We do this with the *special variable* “self”. So, instead of writing “3 + 4”, we write “self * self”. Now all we need to know is how to answer a result. This is done with the “^” operator[2] (pronounced “answer”). Here’s our finished method:

```
squared
    “Answer the square of myself.”

    ^self * self
```

[2] In the original Smalltalk font, this character appeared as an upward-pointing arrow.

To run this method, we need to *compile* the source code we have written and install it in some class, so that an instance of that class will understand a message that invokes it. There are sophisticated tools for entering source code and compiling it, using a graphical user interface. However, for now, we'll do everything by evaluating expressions directly in our workspace.

In which class shall we install our method? Well, for now, we're interested in using the object 3 as our receiver, so we'll install it in the class of which 3 is an instance. You can do that by evaluating the following expression:

```
3 class compile: '
    squared
        "Answer the square of myself."

        ^self * self'
```

Now we can evaluate an expression which uses our method. Evaluate "3 squared", yielding 9. You've just extended the Spoon system; now every instance of that class knows how to answer the square of itself (try evaluating other expressions like "4 squared").

We have now seen three kinds of method selector. The first ("+") was a *binary* selector. Messages using binary selectors use one parameter (as in "3 + 4" or "self * self"). The second ("squared") was a *unary* selector. Messages using unary selectors use no parameters. We saw the third when we compiled our new method with "compile:". This is a *keyword* selector. Keyword selectors are composed of one or more keywords, each ending with a colon.

Let's write another method, this time using a keyword selector. The goal this time will be to answer the receiver raised to a given power, a generalization of our previous method (where we raised the receiver to the second power). We'll call this method "raisedTo:".

Our expression will look like this: “3 raisedTo: 4”. In our method, we’ll need to have a way to refer to the parameter in the message, by giving it a name. It’s usually best to use names which are evocative of the *domain* of the problem. The problem domain here is arithmetic, and a good name for the parameter would be “exponent”.

```
raisedTo: exponent
    “Answer myself raised to the power given by
exponent.”
```

In this method, “exponent” is a temporary variable; we can use it whenever we need to send messages to the parameter, or if we need to use it as a parameter in other messages we send.

We can also define temporary variables that are not parameters of the method. These are usually used to store intermediate results that will get used more than once in the method. We define temporary variables by enclosing all their names in a single pair of vertical bars at the beginning of the method, after the comment.

Let’s define a temporary variable to keep track of a running product. We’ll use it to keep track of a number of multiplications of the receiver with itself. We want to initialize it to one, since any number raised to the zeroth power is one. Variables are set with the “:=” operator[3] (pronounced “gets”).

In our algorithm here, we’ll want to multiply the product by the receiver repeatedly, as many times as the `exponent` parameter. There is a method in the system for evaluating a set of expressions repeatedly, called timesRepeat:. It’s part of the behavior of integers. Its parameter is the set of expressions to be evaluated, enclosed in a *block closure*. We create a block closure in source code by enclosing expressions in square brackets.

[3] With the original Smalltalk font, a left-pointing arrow was used for assignment.

```
raisedTo: exponent
    "Answer myself raised to the power given by
exponent."

    | product |

    "Initialize the product to one."
    product := 1.

    exponent timesRepeat: [
        "Multiply the product by myself."
        product := self * product].

    "Answer the product."
    ^product
```

We now have multiple expressions in our method. They are separated by periods, like English sentences.

Block closures are used in many situations to defer the evaluation of expressions. Their most common use is with the conditional messages ifTrue:, ifFalse:, and ifTrue:ifFalse:, understood by boolean objects. We can use one of them to deal with negative exponents:

```
(exponent < 0) ifTrue: [
    "Answer the reciprocal of myself raised to the
power of exponent negated."
    ^1 / (self raisedTo: exponent negated)]
```

Note in that snippet that we invoked the method we're writing. This is quite common, often as recursion.

The numbers we've been using (like 0, 1, 3, and 4) are examples of *literal* objects. We can refer to them directly with the language syn-

tax, without having to use variable names. There are other kinds of literals as well; let's take a look at them. The first kind we'll look at is an *array* literal.

To make an array literal, enclose one or more other literals in parentheses, with a leading number sign ("#"). For example, "#(3 4)". This denotes an array with two elements, the literal objects 3 and 4. You can nest array literals, and only the outermost number sign is required. For example, you can write #(#(3 4) #(5 6)) or #((3 4) (5 6)).

A *character* literal denotes a single alphanumeric character, with a leading dollar sign. For example, $A is the capital letter A, and $3 is the character 3 (as opposed to the *number* 3).

strings

chapter five: the Spoon object model

chapter six: what all objects can do

chapter seven: fundamental constants

chapter eight: magnitudes

chapter nine: numbers

chapter ten: collections

chapter eleven: streams

chapter twelve: processes

part three: tools

chapter thirteen: inspectors

chapter fourteen: the class browser

chapter fifteen: the debugger

part four: the processor

chapter sixteen: architectural overview

chapter seventeen: the object memory

chapter eighteen: the instruction set

chapter nineteen: the primitives

chapter twenty: remote messages

chapter twenty-one: building your own processor

part five: distributed operation

chapter twenty-two: imprinting and modules

As we saw in chapter twenty, we can send any message to any object on any machine, with any other objects as parameters. Since all the infrastructure for organizing classes and methods exists as objects, we can use our remote message-sending capability to install behavior on other machines. The direct installation of behavior on remote machines is called *imprinting*.

An analogy I like to make is with a scene in the film *The Matrix*. At one point, the character Trinity has an urgent need to fly a helicopter, but has no idea how. Fortunately for her, the helicopter is just an element in a computer simulation to which her brain is connected, and a computer-wielding colleague outside the simulation can simply impress helicopter-flying knowledge upon her mind. After a short phone call to request this knowledge from the operator, she knows how to fly the helicopter.

A Spoon system can make similar requests of other systems on the net. To enable this, the systems must be connected. The connections between Spoon systems are called *wormholes*. When a system resumes, it activates a singleton instance of `WormholeServer`, which listens for connections from other systems. This `WormholeServer` can also establish connections to other systems; its clients are instances of `Wormhole`, a subclass of `MessagingSession`.

After a connection is established, each system's `WormholeServer` has a `Wormhole` client associated with the connection. Each system can access the other's `Wormhole` by sending `peer` to its local `Wormhole`. Once a remote system can send messages to it, a `Wormhole` is the remote system's initial point of presence in the local system. The `Wormhole` provides various system services to the remote system, such as making object memory snapshots.

Since remote messages can be sent to instances of any class, with any objects as parameters, the methods of `Wormhole` may be written without regard to the fact that they will be invoked from afar. Similarly, the protocol for sending remote messages need not concern itself with the details of any particular messages that might be sent. In particular, a system can use a remote `Wormhole` to gain access to a peer system's classes, so that it can install methods and create new classes.

This access is not provided directly by the `Wormhole`, however. Most of the protocol for transferring behavior between systems is provided by *modules*. Once you have a reference to a remote `Wormhole`, you can get a new remote `Module` by sending the message `module` to it. `Modules` record the presence and absence of methods in the system. They also know how to define new classes and install compiled methods, under the direction of the remote originals. A `Module` may be made a prerequisite of other `Modules`.

Many of the objects that `Modules` manipulate, such as classes, are known to humans by textual names. Since names can change and conflict, Spoon avoids using those names as much as possible. In-

stead, named objects are associated with *universally unique identifiers*, or UUIDs[1]. Each `Metaclass`, for example, has an instance variable `id` which refers to an instance of `UUID`, a subclass of `ByteArray`. Every behavior, therefore, may identified by a combination of a `Metaclass` id and a `Boolean` indicating whether or not it's a `Metaclass`. For example, class `Array` is identified by the ID for (`Array class`) and `false`, while (`Array class`) is identified by the same ID and `true`.

In a network conversation between `MessagingSessions`, these `UUID`/`Boolean` combinations are conveyed by instances of `BehaviorID`, another subclass of `ByteArray`. Each `BehaviorID` contains the bytes of the `UUID`, and a bit for the `Boolean`. Whenever a `Module` needs to refer to a `Behavior`, it uses a `BehaviorID` instead of the `Behavior`'s name.

Another prominent system class which specifies an ID is `Author`. Each `Author` corresponds to a code-generating entity (usually a person). Each version of each class, method, and module has at least one `Author`. Finally, each `Module` has an ID as well.

New `Module`s may be created at will. One may then specify the method presences and absences asserted by the `Module`:

```
| module |

module := (
    Module
        named: 'a sample module'
        withDescription: 'just a short example').

module
    addMethodNamed: #yourself inBehavior: Object;
    addAbsenceOfMethodNamed: #zork inBehavior: Object
```

[1] search the web for "Leach Salz UUID" for the specification

When this `Module` is installed in a remote system, it will ensure that the target system has the method `Object>>yourself`, and that it does *not* have the method `Object>>zork`.

A `Module` records method information with instances of `MethodDescription`. Each `MethodDescription` refers directly to the local class, selector, version, and author of a method. In remote messages, `MethodDescription`s are represented by instances of `MethodID`, a subclass of `BehaviorID` which includes storage for a selector and method version.

For other people to install our `Module`, they need to *discover* it first. Spoon provides a relay network of server systems which both advertise available modules and provide for their installation (for the details of this system, see chapter twenty). For now, let's assume that someone has discovered our module and, armed with the module's ID, has begun installing it.

The receiving system's `WormholeServer` connects to the providing system's `WormholeServer`, creating a `Wormhole` that represents the connection. Sending `peer` to this local `Wormhole`, the receiving system obtains the corresponding `Wormhole` in the providing system. The receiving system then asks the providing system's `Wormhole` for the `Module` corresponding to the desired module ID. Finally, the receiving system creates a new empty module and asks the providing module to synchronize with it.

At this point, the providing module takes over, guiding the receiving module in modifying its system so that the providing module's assertions are true in both systems. After setting the receiving module's name and description to match its own, and ensuring that the receiving system is synchronized with all of its prerequisite modules, the providing module asks each of its asserted compiled methods to define itself in the receiving system.

Whenever an object in the providing system has a question about the receiving system, it simply asks the receiving module. For

example, when an asserted method is asked to ensure its existence in the receiving system, the first thing it does is ask the receiving module if an equivalent method already exists there. If so, then the method doesn't need to do anything. If not, the method ensures that its `Behavior` exists in the receiving system, defining it if necessary, and installs a copy of itself.

This marks an important difference between using live objects and static files for transferring behavior. With files, the state of the receiving system is not taken into account. If a method is provided for a class which is missing or has an old definition, an unrecoverable error occurs. If a method is provided which is already present, the receiving system spends time compiling and installing it anyway, creating a superfluous version. With live objects negotiating directly, only what is needed is transferred, and the receiving system doesn't have to compile anything.

When the modules are finished synchronizing, the receiving module is a perfect copy of the providing module, able to convey the same assertions to other systems.

chapter twenty-three: collaboration

Most of your time using Spoon will be spent collaborating with others and with *yourself*. You will find your past self to be your most important collaborator, and very much like another person as your frame of mind changes. Over time, you will forget the context in which you made various design and implementation decisions. The tools will help you answer the frequently-asked question “What was I thinking?”

While the environment is a great aid to understanding past development, it is most effective when used with a certain discipline. The central virtue of this discipline is *creating documentation*. You can attach a comment to nearly every system component: methods, method categories, classes, class categories, authors, and modules. Your primary activity will be reading old code, either learning how to use it with your own code, or fixing problems with it. The more comments you write, the easier your development life will be.

chapter twenty-four: deployment

appendix A: the Spoon story

appendix B: notes on the text

Teaching object-oriented programming to newcomers is tricky.

index

colophon

This book was typeset by the author on an Apple Macintosh PowerBook G4 computer, using the Pages editor, and converted to PDF format for distribution and printing. Figures were prepared with Squeak.