A mentoring course on Smalltalk

Andrés Valloud

September 12, 2010

Contents

Introduction

The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.
—Donald Knuth, The Art of Computer Programming, Vol. 1, p. v.

I started programming when I was 10 years old. My first hacks were turtle-drawn geometric kaleidoscopes running on a TI 99/4A computer. I still remember the tricks to make it go faster, such as making the turtle invisible so that Logo would not have to draw it[1].

The instructor, Daniel Gentelesca, gave me some privileges in exchange for responsibilities. Sometimes, he would let me take the brass key to the computer room so I could go there to program during the 15 minute breaks between classes. In return for those minutes of bliss, I would prepare all the computers for his next class.

One day, he loaded a floppy disk in the only more advanced Spectrum ZX-80 computer we had at the school, and what I now know to be copper bars appeared on the screen. I was astonished! One could have the computer display all those colors *at the same time*! My rudimentary turtle drawings became obsolete in a fraction of a second.

I asked Daniel what had been used to program those color bars. He answered they had been programmed in assembler. I demanded to be taught assembler immediately. Despite my repeated requests, he refused on the grounds that I was too young and therefore I would not be able to understand it.

But I knew that if I put myself to it, I would master any programming language. At that moment, Logo was enough proof of this for me. Therefore, I began setting ever more ambitious goals for my computer programming skills.

[1] And since that implementation of Logo ran out of "ink" and refused to continue drawing after some point, hiding the turtle had the benefit of saving the precious available "ink" for your own stuff.

Regarding the roads to success

We all experience a rush when we set a new personal record of proximity to perfection according to predetermined rules. As rules become easy to meet, however, excitement gives way to boredom and lack of challenge. At this point people usually choose stricter rules to improve some aspect of their activities, and the cycle starts again.

If we assume the point is to refine the way we perform an activity, then we will tend to choose new constraints with the goal of increasing the ratio between work that contributes towards reaching a goal, and work that does not.

As with everything, there is a spectrum of possibilities. For example, you could penalize every act that can be shown to be useless or detrimental. At first, this approach may be very beneficial because it can be used to prevent major mistakes. However, the issue with it is that it does not scale well. In the long run, most sets of rules based on punishment for possible offences will quickly grow and become impossible to understand by a single person. And it is way too tempting to make up yet another scenario where things can get messed up. In this situation it is easy to find ambiguous cases, while generalizations become impossible in the presence of numerous special exceptions.

This is why lawyers specialize in certain branches of law. I am sure you are familiar with this situation.

In my opinion, computer programmers often try to improve by choosing stricter rules along these lines, in the sense that they impose more and more punitive restrictions on themselves. In this sort of situation, people typically exhibit a tendency to follow the path of least resistance. In other words, we try to spend the least energy possible to live within the rules. To avoid punishment becomes the accomplishment.

This is a hefty price to pay! When the proportion of positive and negative statements is biased towards the negative ones, we end up missing opportunities for more transcendental improvement. Our choices silently and pervasively limit the point to which we can be successful.

There is another way to choose rules. Nobody deserves to try being creative while complying with the Developer's Code of Acceptable Conduct. So instead of choosing a set of restrictions, we could just as easily take a set of positive rules that, by means of their interactions, put us in a situation where every positive act reinforces the others.

And certainly, it is much better to naturally tend to success than to be frequently saved from failure.

A positive approach

This is where I think there is a void in the body of object oriented literature. There are so many introductory books, cookbooks, pattern books, and even so called advanced books which are no more than user manuals for nice tools in disguise... but once you read them all, where do you go? Where is the book that teaches you how to come up with a strategy to tackle previously unseen problems with elegance?

Mentors are usually necessary to fill the gap. But there are only so many lucky encounters with these rare creatures. It is my hope that I will be able to help address this problem.

This book describes a set of tendencies that work together to create a positive approach to successful programming. It accomplishes this by describing the application of these principles to solve problems where often the solutions are hard to design in simple terms.

The results are fantastic. Programs are beautiful, a true joy to read and maintain. The amount of code written is invariably small, even tiny. It is easy to reach extremely high efficiencies at run-time. And with practice, it becomes natural.

In addition, this different state of mind in which programs are produced is so much fun that it becomes deliciously addictive. It becomes alive and takes over. It is a liberating metamorphosis.

Reality check

This book will assume that you are reasonably comfortable with Smalltalk, that you have a desire to take your skills a significant step further, and that you understand that the process will require serious and sustained effort as at times the road may not be easy to negotiate.

There ain't no such thing as a free lunch.

All the material in this book comes from real life projects. In addition, the frameworks, the approaches and the techniques are in place in production systems, making a big difference every day.

As a matter of personal preference at the time of writing, the source code examples were prepared using VisualWorks Smalltalk. Fortunately, VisualWorks runs on all major PC platforms. You can obtain a free, non commercial version of VisualWorks for your computer by visiting http://www.cincomsmalltalk.com. In addition, at this website you will see that there is a mailing list for the non commercial users of VisualWorks, and that there is also an IRC channel for when more direct contact with other Smalltalk developers is desired.

Exercises

The connection between computer programming and mathematics, particularly algebra, is often downplayed. Yet, one should never underestimate it because missing how they complement each other takes the fun away from both.

As with mathematics, it is impossible to master a computer programming skill without applying it. Therefore, this book contains exercises for the reader to get more familiar with the content being presented.

"In general, to study *means to memorize things in order to apply them later on an as-is basis. For example, with a good memory, one could be a good doctor or a good lawyer, but not necessarily a good mathematician. Memory does not matter. What works is being able to act, to do, to think. Mathematics require a creative, curious, observing, yearning attitude. This attitude is connected with the ability to do for oneself. It is useless to know something if one does not know how to use it."*
—Enzo Gentile

Exercises are rated in difficulty by means of an integer between 0 and 50. The scale is meant to be logarithmic, and follows the guideline below (which in turn is taken from The Art of Computer Programming).

00 Trivial. Requires no time for readers familiar with the material.

10 Easy. Requires about five minutes to find a solution.

20 Moderate. Requires about half an hour of work.

30 Difficult. Requires several hours and substantial effort to solve.

40 Project. Requires many weeks to be addressed adequately.

50 Open problem. Research is necessary, no solution is known.

It is hard to judge the time necessary to solve an exercise, especially when you create the exercises yourself. The ratings represent the author's best estimate.

Answers to the exercises are provided in an appendix at the end of the book. Readers will get the most benefit from them after an honest attempt at solving each problem.

Feedback and further information

This book will become better only with the help of readers. Should it become necessary to contact the author, maybe to report a typographical error, a code bug or other problem, or perhaps to make a suggestion, please send an email to smalltalkMentoringCourse@gmail.com. Any feedback is welcome.

Acknowledgements

There is more than the author behind this book. The people that helped me write it are among the most successful and gifted object oriented software artists. We share a true feeling of joy when we do our job, and we take our fun very seriously. Thank you so much for your input (in alphabetical order): Andrés Fortier, Blaine Buxton, David Caster, Eliot Miranda, John Sarkela, Leandro Caniglia, Luciano Notarfrancesco, Sheldon Nicoll, and Valeria Murgia.

In addition, I would like to express my appreciation to the careful readers who have provided feedback to make this book better. In alphabetical order, they are: Georg Heeg, Lawrence Trutter, and Reinout Heeck.

Thanks for reading!

Exercises

Exercise 0.1 [08] List some of the restrictions you adhered to at your first programming job.

Exercise 0.2 [25] Install VisualWorks in your computer.

Exercise 0.3 [50] Find an efficient way to break RSA keys.

Chapter 1

It's all in a name

1.1 Preliminaries

This book begins with a discussion on a quite abstract subject: some of the peculiar characteristics of how we perceive. With these details in mind, it will be much easier to arrive at powerful generalizations later on. In addition, many things that are usually considered self-evident will be clearly seen as consequences of much simpler and more profound principles.

The act of seeing things is such a common state of affairs that it is usually assumed to be obvious, even axiomatic. Because we take this feature of our lives for granted, we seldom question the mechanism that makes it happen. But much waits to be understood behind this apparently intractable phenomenon.

1.2 The very act of naming

1.2.1 Distinctions

Take a moment and look around yourself. You can easily recognize familiar objects around you. Have you ever considered the difference between a colorful blob as recorded by your retinas and *seeing things*?

I often think about it. Sometimes, by accident, a snapshot of the colorful blob is recorded on my mind, without any of the boundaries that let me distinguish between different things. It is as if I were seeing a piece of cloth on which several cups of paints of different colors had been almost carelessly emptied. It always seems to be out of focus. Then, just as I am trying to get a better glimpse of it, in a split second the illusion disappears and suddenly I am seeing things again.

Sensation: you notice the colorful blob. Perception: you recognize a car. Cognition: it is your car and somebody is driving off with it.

What are the characteristics of the process by which we see things? It seems that things, whatever they are, have a strict boundary that separates them from their exteriors.

For example, speaking in very general terms, we clearly understand where something like a wall starts in terms of physical location. Maybe there is some fuzzyness in as much as the wall perhaps does not perfectly fit against the floor, but even a low resolution idea of the wall's location is typically enough to avoid walking into it. In the same way, we also understand that what is outside the boundary of the wall is not the wall. Thus, it can be said that, out of all the blob we can perceive, we *distinguish* the wall by separating it from everything else.

Without exception, living beings have a boundary which distinguishes them from their environment.

Thus, *distinctions* are characterized by having a limiting boundary. They represent the act of reifying the existence of, or maybe thinking about, an object. More precisely, a distinction is the consequence of separating something from its environment.

The distinction's boundary has the quality of being unambiguous. It sorts all locations in the universe into two sets: the inside, and the outside.

Generally speaking, you can think of a distinction as a circle drawn on a piece of paper. As you can see, the circumference unambiguously sorts all locations of the plane into two regions: the inside of the circumference, and the plane with the circle carved off from it. Thus, the circle itself and the rest are now distinct places separated by the circumference.

In three dimensions, we typically think of the book, the chair, the coffee cup, and many other things. When we listen to music, properly tempered tones are associated to a well defined range of frequencies. When we think of time using a wrist watch, we easily draw distinctions in terms of before and after. In general, we construct our experience of the world in terms of distinctions. As far as we are concerned, all the objects we distinguish have precisely defined boundaries.

"The environment is everything that is not me."
—Albert Einstein

1.2.2 Intentions

Distinctions are drawn by an observer. There cannot be distinctions without intentions. Different intentions will result in different distinctions. These are fundamental notions that must be thoroughly understood and boldly taken to their final consequences.

It is easy to see that different intentions lead to different distinctions. If you want to make a payment out of your checking account, you may distinguish a debit card in your wallet. But let's say your front door locks you out of your home,

putting you and the keys on different sides of the boundary that distinguishes your place from the outside. In this situation, you could see the same piece of plastic with very different eyes, with quite different intentions. It would not be a debit card anymore. It might even become a tool to get in.

This change in point of view brings extremely powerful consequences. By means of putting the piece of plastic on the same level as a tool, all the things you can think of doing with tools become a possibility for a piece of plastic that you scanned at a point-of-sale terminal not long ago.

1.2.3 Labels

If you have circles drawn on a piece of paper, it is quite natural to give them names to be able to distinguish between them.

An important fact to notice is that, once you remove the names, the circles are quite equivalent to each other. Thus, it is clear that the most disparately named distinctions can be on an equal footing when it comes to the meaning of carving a portion out of the universe of possible things. It is perfectly normal for distinctions to be quite mundane, lacking a special place on the piece of paper.

For example, if you were to look around, you would see things. However, what you perceive as things are regions of your perception space defined by a boundary. The boundary is set by you, the observer, according to your intentions.

If you pay enough attention to let some regions be determined by means of boundaries, but do not give names to these regions, you will see that they are quite similar to each other as far as bounded regions go. At this point, they are distinctions without a name, and as such they are quite equivalent to each other.

Names are labels which are applied on distinctions, just as if they were post-it notes. This leads to the following principle, which is extremely important to keep in mind at all times: *labels should never be confused with the labelled objects.*

A language can be constructed by means of a set of labels which, considered as distinctions named as themselves, have a particular relationship between each other. The relationships between the labels can be thought of as the connections between the nodes of a graph in which the labels are the nodes.

In a language built this way, the *meaning* of a label is derived from the local topology of the graph in the immediate neighborhood of the label's node. This applies regardless of which distinctions the labels are applied to.

Languages give meaning to things by means of the relationship between the labels. When labels are applied, the meaning given to distinctions *does not depend on the labels themselves.*

The particular appearance of the labels does not matter either. The only practical requirement for the representation of the labels, which can be thought of as the "words" of the language, is that it should be possible to distinguish between them.

All of this puts no restrictions on *naming* distinctions using the labels of the language. We can simply go and label distinctions according to our intentions. In particular, when you have different intentions, you apply different labels to possibly different distinctions and thus derive different meanings[1].

"Point of view is worth 80 IQ points." —Alan Kay

But what is the process by which all of this works? In particular, what are the properties of labelled objects that are needed for labels to be useful in a consistent manner?

1.2.4 Laws of form and object oriented programming

The concepts of distinctions and labels come from the book Laws of Form, by G. Spencer-Brown. In his book, Spencer-Brown introduces the two following principles on which the previous discussion is based.

> The law of calling: *the value of a call made again is the value of the call.*
>
> The law of crossing: *the value of a cross made again is not the value of the cross.*

The law of calling means that if you name a distinction by means of applying a label, and if you repeatedly use the label to refer to the distinction, this use is compatible with the common-sense notion of names that do not change their meaning over time[2].

For example, if I called Joe once, and then called him again in exactly the same context, the repeated use of his name does not change the distinction behind the label "Joe": Joe himself.

This, despite the fact that all of Joe's atoms are replaced every seven years.

The law of crossing, on the other hand, means that if you cross a distinction, then cross it again, the result is the same as doing nothing.

[1] Regardless of how elaborate a language can be in terms of relationships between its words, it is still up to us to be able to represent different patterns of ideas with it. Sophisticated languages do not guarantee sophisticated ideas.

[2] Note how in science, one attempts to find distinctions that describe the world in such a way that time disturbs them the least. A perfect example of this is the principle that anything we label as *energy* is not allowed to be created nor destroyed.

For instance, on a piece of paper, if you cross a circumference to get on its inside, then cross the same circumference again getting on its outside, you end up in the same place you were before crossing the circumference for the first time (assuming nothing else changed in the mean time, of course). Hence, you have done nothing.

More elaborately, drawing a distinction is cutting the whole universe into two pieces: the inside of a boundary, and the rest. Distinctions sort all possible things into either the *something* bucket or its complementary *not something* bucket. Clearly, taking the complement of a bucket twice is the same as doing nothing.

In the strictest sense, this means that drawing *the same* distinction around a distinction is equivalent to *nothing*.

1.2.5 Distinctions, labels, and Smalltalk

In Smalltalk, objects *are* distinctions. Since everything in Smalltalk is an object, then everything in Smalltalk is a distinction.

If we look at this in terms of the piece of paper analogy, instance variables inside an object are circles inside an enclosing circle. To distinguish between instance variables in the context of an object, we typically *name* them just like we name circles.

In Smalltalk, assignment lets us name any distinction by applying a label. Since everything in Smalltalk is a distinction, then everything in Smalltalk can be named. This is an extraordinary ability to have.

Note that assignment should not be interpreted as loading a value into some named memory cells. Naming an object should not be seen along the lines of modifying the object to hold its name either. On the contrary, the act of giving a name should be thought of in terms of remembering an object by means of a label. Also, since a distinction is not the private property of the name either, you are free to give a distinction as many names as you want.

In some Smalltalks, assignment is written as ← instead of `:=`. This gives the wrong impression as well. It would be much better to use → or `:>`.

Commonplace keywords like `instanceVariableNames:` and expressions such as "temporary variable names" take a whole new meaning from this point of view. In fact, they should not even contain the term *variable* at all. Therefore, we will not refer to names by means of, or assisted by, the word *variable* any further.

Ideas like instance name encapsulation also take a new powerful meaning when viewed in terms of distinctions. Instance names live inside the boundary of instances, thus in order to get at them the boundary of the instance has to be

Note how without the word variable*, this statement becomes trivial.*

crossed.

Since everything that happens in Smalltalk occurs in terms of messages, it turns out that messages are the only entities that can cross boundaries, possibly carrying other objects (distinctions) with them.

Did we mention Alan Kay has a biology background?

Compare this mass of distinctions and messages crossing boundaries with other complex systems in which objects interact. One example could be a live multicellular being such as yourself, in which multiple cells typically influence the behavior of each other by means of protein messages which cross cell membranes.

Or let's say you would like to have lunch with a friend. In an assembler or obfuscated C world, you would reach inside your friend's mind and manipulate values and pointers until you made lunch appear attractive. You might even tickle the stomach to get the appetite going.

But Smalltalk imposes much more civilized manners. In the same way that your friend is not a `struct{...}`, objects are not allowed to arbitrarily cross distinctions and mess with internal details. Instead, objects must communicate by means of messages. As long as the agreed upon behavior is respected, the implementation artifacts are happily taken for granted.

If we keep in mind that everything in Smalltalk is a distinction, and that all distinctions can be named, then it follows that it must be possible to name every distinguished bit of behavior. In Smalltalk, the names of bits of behavior are called *selectors*.

Smalltalk sentences are no more than invocations of bits of behavior, where the specific behaviors in question are distinguished and called upon by means of their selectors.

Since messages provide the only mechanism to cross into a distinction, it obviously follows that even the behavior named `instVarAt:`[3] must be a message.

1.2.6 On the use of labels with the special object array

One of the first things stored in Smalltalk images is a small array of special objects, which usually contains references to objects like `Smalltalk`, `nil`, `true` and `false`. Why is it necessary to give special treatment to these things, and what does this special treatment imply?

[3]For those not familiar with the message `instVarAt:`, what sending it does is to answer the object referenced by the nth instance name of the receiver. This works because all instance variables are stored in the order given by the class and superclass definitions. Feel free to check out the implementation in your Smalltalk of choice.

Consider the implementation of the message `==`. This message is implemented as a primitive, and as such the actual work is performed by the virtual machine running the image. Essentially, what it will do is to see if the argument and the receiver are not just equal, but exactly and identically the same object. Whatever the objects involved are, after the comparison is done the virtual machine will have to answer either `true` or `false`. Why these objects and not anything else? Because the rest of the code in the image is written in a way that requires one of these to be answered by the primitive. And how does the virtual machine know the objects `true` and `false`? By looking them up in the special object array.

Of course, one could have the virtual machine behave more like C and answer 1 or 0 instead. And in fact one could have it do so, because the virtual machine itself does not care. Therefore, the fact that the virtual machine answers `true` or `false` is the result of some arbitrary choice. As such, we can separate the act of answering two different things (the actual answer) from the actual things being answered (how the answer is labelled for our convenience).

But then, if the actual answer could be whatever pair of distinct objects we want, we can also see that the meaning of these objects is not given by their name (`true`, `false`, 0, 1), not even by their identity. Rather, they become meaningful because of their location in the network of message paths. If we replace `true` and `false` by 0 and 1, the image would still work if what is expected from booleans is satisfied by their replacements.

This is an excellent opportunity to remind ourselves to avoid confusing the labels with the labelled objects. It is perfectly fine to refer to distinctions by means of names, as long as we always keep in mind that a name is something we assign according to our intentions. But labels are not the labelled objects.

1.3 Large systems

1.3.1 Emergent properties

We are surrounded by systems made of an extremely large number of parts. We are, ourselves, one example of such a system.

Everywhere inside our boundary, the uncertainty principle rules the chemical reactions that keep us alive. Our cells communicate with each other by means of mechanisms that are not guaranteed to succeed. When our brains are built, chaos reigns at the lowest level where neurons connect to each other for the first time. And there you are, reading this book. What are the principles that make this possible?

Assume there is a master plan with detailed step-by-step instructions for all your cells to interact with each other such that you can read this sentence. Due to the enormous amount of possible scenarios, it would be impossible for it to account for every single thing that could go wrong. The whole universe would not be large enough neither rich enough to store all the contingency plans.

"Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius, and a lot of courage, to move in the opposite direction."
—Albert Einstein

Therefore, master plans written in terms of detailed instructions are not only impractical — they are infeasible. In general, life does not work by means of commandments given to each piece of the system. Rather, things happen due to the idiosyncrasies of the chaotic interactions between the parts.

As an example, let's consider how solar systems form. At first there is an amorphous cloud of gas and dust. This may seem a quite unattractive state of affairs. But give the right nudge to the system, and it will cross a point of no return. From that moment on, *the cloud will not be able to help itself*. Regardless of what actually happens between all the particles of the cloud, at least one star will form out of it.

What makes the star out of the cloud are the characteristics of the relationship that exists between each of the particles in the system, a suitable initial condition, and enough time to let the cloud converge towards the implicit consequences.

If you let the system get to the initial condition, there will be no possible outcome other than at least one star.

Systems of interacting parts tend to converge towards a steady state. Emergent properties are the characteristics of the steady states.

Stars are *emergent properties* of clouds of gas and dust, and so are we.

In exactly the same way, storms and wind bands are emergent properties of the atmospheres of rotating planets in orbit around a star. We can readily see that in our own solar system. Storms and wind bands occur regardless of the planet's composition, the gases involved, the planet size, the orbit's inclination, the distance to the star, the rotation speed, whether the planet has seas or not, etc. They are present even in planets' satellites as well.

These atmospheric phenomena are self-organizing emergent properties of the system as a whole. They really cannot help themselves. There is no alternative for these atmospheres other than to have storms and wind bands.

1.3.2 Behavior in Smalltalk

Large systems and the chaotic interaction between their parts can be readily modeled by numerous distinctions which message each other in such a way that, given an initial condition and enough time, the outcome of the system's evolution will be completely determined by the characteristics of the interaction between the distinctions.

In Smalltalk, everything is a distinction and everything that happens is a message send. Because of this, Smalltalk is an extremely suitable environment in which to model complex systems and their emergent properties.

This key observation will be tacitly used to let behavior occur as an emergent property of objects and their interactions, as opposed to giving commands and issuing orders. This change of perspective, even though delicately subtle, is of the utmost importance.

The art of object oriented programming is the conscious and careful design of the pieces such that the desired outcome is an emergent property of the whole.

This book is an attempt to illustrate this distinctive and artful way of building programs, how it can become second nature for you, and how you can obtain valuable results by using it.

1.4 Some best practices summarized

When you view things from the point of view of distinctions and the laws of form, things have a tendency to become simpler. For example, we could consider some of the well-established productive patterns of Smalltalk programming behavior. Rather than hard won individual bits of good sounding advice, they become simple consequences of deeper principles. Lengthy justifications become trivial deductions that amount to not much more than an assertion.

Let's go over some familiar examples to see the axioms of Laws of Form in action. While we will discuss them abstractly at this time, they will be put to full use throughout the rest of the book[4].

[4]This inevitably brings up the topic of refactoring. When I first heard about refactoring in 1996, it was described as some sort of homework that you had to do to improve the code after it was done. But if I learned how to refactor, then that would imply that I would have enough knowledge to distinguish between refactored code and stuff in need of refactoring. Thus, I wondered why should I knowingly write messy code that would need refactoring later. Therefore, I decided to train myself to refactor code as I was writing it to avoid causing myself homework. It was painful at first, especially because I also learned Smalltalk in 1996. After some time and a lot of practice, however, it became more or less natural.

Familiarity is always more powerful than comfort.
—Virginia Satir

I find this skill, honed and sharpened for nearly as long as I have used Smalltalk, to be absolutely indispensable. If you do not currently do this, then I strongly suggest that you start immediately. The following may serve as further illustration.

1.4.1 Intention-revealing names

Labeling distinctions allows us to clearly distinguish between them. But it also transfers the meaning behind the label to the labelled object. Thus, labels must be chosen carefully so that the names hint at the right meaning.

From another point of view, the existence of objects is the consequence of a particular set of intentions. If you do not know the intentions because whoever wrote the code in the first place did not spell them out explicitly, then you are left with figuring them out. The key observation here is that it is easier to figure intentions out when the objects are labelled with intention-revealing names.

Developers should always keep a thesaurus handy.

This kind of self-documentation can lead to the discovery of unsuspected properties of the labelled objects. The potential impact of good quality names should never be underestimated.

1.4.2 Small methods

Long methods are usually a concatenation of several shorter pieces of code that accomplish something in a blow-by-blow fashion. Sometimes, there are comments

---

A visitor to an Irish castle asked the groundskeeper the secret of the beautiful lawn. The answer was: *just mow the lawn every third day for one hundred years.*

Roger Whitney comments in college course material that spending a little time frequently is much less work than big concentrated efforts, and that it produces better results in the long run. Therefore, he concludes that it is wise to frequently spend time cleaning up your code. I could not agree more.

Refactoring in this continous and merciless manner is connected with software quality as well. Here are some definitions.

Quality Problem: Every software problem is a quality problem.

The Zeroth Law of Software: If the software does not have to work, then you can meet any other requirement.

The Zeroth Law of Quality: If you do not care about quality, then you can meet any other requirement.

The Zeroth Law of Software Engineering: If you do not care about quality, then you can meet any other objective.

The Boomerang Effect: Attempts to shortcut quality always make the problem worse.

Quality is producing things of value to some people. Real quality improvement starts with knowing what your customers want. What is the quality of the programs you are working on, right now?

or blank lines in between the pieces as well. These separations are strong hints of where distinctions should have been more explicitly drawn.

The point of separating the pieces and putting them in methods is simply that of distinguishing the parts and applying labels. This allows smaller parts to interact, which leads to behavior networks which are easier to change and improve.

While large sections of at first sight cohesive code may be acceptable when written in a workspace or an inspector where one may want to accomplish a one time goal, they really do not belong in methods because typically one creates classes to manage functionality that will be around for longer than a workspace.

To maximize the benefit of using small methods to implement behavior, the written expression of a method should consist of the following two parts:

1. An external interface definition, consisting of the name of the selector and the internal names of any supplied arguments. Comments may be added if *absolutely* necessary.

2. A small amount of source code that implements the behavior described by the external interface definition above.

Note the strong contrast between the two[5]. The external interface is made known to any potential sender, while the implementation is kept private to the receiver[6]. The distinction between the two should always be explicitly illustrated by means of an obvious visual cue such as a blank line. In addition, no well-factored method should be interlaced with blank lines or comments since these hint at distinctions which are not being made explicit for everyone to see.

Selectors should be used to clearly reveal the intention behind relatively small methods. When this is done, comments display a strong tendency to become redundant.

Refactoring is a process by which one can address these and other deficiencies code may have.

1.4.3 Elimination of conditional logic

More often than not, a high frequency of the message `ifTrue:ifFalse:` suggests that distinctions which might have been possible at design time are being drawn

[5]You could think of the selector as the cell membrane receptor, and the method being the implemented response to the receptor recognizing a message.

[6]This means that a definition consisting of selector pattern and subsequent code sits on both sides of the fence at the same time: the selector sits on the outside of the object, while the method lives inside the object. Note that all of this is left unsaid, and that the accomplished brevity allows for an exquisitely concise way to take advantage of the structure of it all.

at run time. But if distinctions were drawn at design time in the form of classes, then it would not be necessary for the running program to repeatedly rediscover what was known by the developers.

This leads to conclude that drawing design time distinctions at run time is *intention obscuring* by nature. It is a sign of missing explicit representation of valuable distinctions, of information kept private in the developer's mind. It is also the manifestation of unnecessary homework left to any reader of the code. This is because classes are explicit consequences of the intentions behind their existence.

From a more pragmatic point of view, what is even worse about this homework is that it applies to the virtual machine as well. Drawing design time distinctions at run time takes computer resources which could be more wisely spent.

Polymorphic message sends allow the meaning of messages to be defined by the context in which the message is received. Therefore, in programs in which design time distinctions are drawn explicitly as classes, `ifTrue:ifFalse:` tends to become unnecessary.

If you need to improve performance, classes and polymorphism are some of your most powerful allies because they can make it so that the run time does not need to spend any time whatsoever recreating known information.

The assertion that more classes can result in improved performance may seem strange at first. However, it is the case that virtual machines make large systems of objects evolve faster and more efficiently when behavior depends heavily on polymorphic message sends rather than on `ifTrue:ifFalse:`. This is because virtual machines are well equipped with machinery that executes polymorphic message sends extremely fast. Therefore, let go of the temptation to do the work of the virtual machine by invoking special messages such as `==`, `isKindOf:` and `ifTrue:ifFalse:`.

You can only name what is already distinguished.

In short, *distinguish* and conquer.

An excellent example of using polymorphism to avoid drawing unnecessary distinctions at run time, and of letting the meaning messages be defined by the context in which they are received, is how `true` and `false` themselves are implemented.

Note how sending `isNil` literally causes the appearance of `ifTrue:ifFalse:`.

There are other familiar situations in which this problem comes into play, even though at first sight the issue is not as evident. For instance, a typical symptom of drawing unnecessary distinctions at run time are repeated `isNil` checks. Usually, it seems almost obviously *necessary* to check for several cases and make objects forcefully choose what to do every time by means of issuing commands.

This situation, however, should be interpreted as a clue telling us that either `UndefinedObject` or some equivalent object that represents the lack of a value should become polymorphic with the objects that represent the presence of a value.

A way to solve this issue is to create classes. For instance, the class of the object `nil` is `UndefinedObject`. This class could in turn have a subclass named `UndefinedString`. For convenience, the only instance of `UndefinedString` could be added to the global name context with a name such as `nilString`.

1.4.4 Allow, do not force

Consider for a moment a program in which all message sends make perfect sense in the context of their receivers[7]. As we just discussed, such a program would not need to draw design time distinctions at run time, and would certainly seem to be lacking `ifTrue:ifFalse:` when compared to average (for lack of a better qualifier) programs.

What does `ifTrue:ifFalse:` represent when it is used to draw run time distinctions which could have been made at design time? Clearly, it is the job of the developer to draw design time distinctions at design time. If this is left as homework for the run time, then the developer has to provide code that will draw those distinctions so the running program can do the homework. In Smalltalk, the perfect example of such code is `ifTrue:ifFalse:`.

While that is bad already, it may be the case that a high frequency of `ifTrue:ifFalse:` actually represents more than just a matter of unfinished work. In fact, we could even go as far as to contend that any occurrence of the message `ifTrue:ifFalse:` written with the particular purpose of drawing a design time distinction at run time, be it explicit or implicit such as with case statements, represents the developer issuing commands in the first person.

In large systems, behavior is most easily and effectively achieved in terms of emergent properties. Therefore, since no complex system depends heavily on commands, no program should depend heavily on developers writing code in the first person[8].

If this conclusion is taken to its final consequences, it can cause a tremendous change in perspective. For example, consider a reasonably large program based

[7]The reader may point out at this point that the assertion is tantamount to some utopia.

[8]Unfortunately, it is not always possible to avoid drawing design time distinctions at run time. For example, consider the implications of implementing `Integer>>isPrime` without using `ifTrue:ifFalse:`.

mostly on code written in the first person. Orders, as opposed to messages, do not necessarily respect distinction boundaries. Hence, the whole system can be considered to be an enormous `struct{...}` accompanied by a huge set of commandment tables.

For this kind of program to work, the commandments must be written in the context of the `struct{...}`. But the `struct{...}` exists in a global context, hence the developer must write the commandments from a point of view in which everything is known and anything can be done. In other words, the developer must become omniscient, omnipotent, and perfect.

To begin with, nobody can be expected to succeed in these circumstances. In fact, it is an excellent setup to achieve failure! Nevertheless, assume that after a lot of pain and suffering, the 10-zillion-commandment tables are finally produced. Are we done? Is success achieved?

Unfortunately, the answer is no. This is because things change all the time. Since the commandments were created in a global context which quickly becomes out of date as change occurs, then *every* commandment is potentially out of date as well[9]. Like Sisyphus, the developer must then restart the work[10]. Why would anybody put up with such masochist torture?

Rather than obtaining behavior from commandments, it is considerably more productive to carefully design the interaction of the pieces such that the emergent properties of the composite *are* the desired behaviors.

To get the most benefit out of this approach, the pieces must be designed such that small changes in their interaction produce different varieties of interesting emergent properties. If, such as with Smalltalk, the cost of these small changes is small as well, one has a setup which allows sustainable success.

Smalltalk is a powerful tool because it does not have frameworks which impose a large overhead on small changes. For example, the lack of an explicit, early bound type system is a vital characteristic that keeps the cost of small changes at a low level. In the long run, this is far more important than preventing a few simple mistakes.

From a technical point of view, if a Smalltalk program is written under these guidelines using very little code, it is hard to see why it should not succeed.

---

[9]This excludes the possibility of the understanding of the problem itself changing, which from experience we know to be a very frequent event.

[10]Wikipedia provides the following background: In the underworld Sisyphus was compelled to roll a big stone up a steep hill; but before it reached the top of the hill the stone always rolled down, and Sisyphus had to begin all over again (Odyssey, xi. 593). As a result, pointless or interminable activities are often described as Sisyphean.

1.4.5 On courage

More often than not, we write programs with some lack of understanding. If this becomes too apparent to ourselves, it can cause fear. It is well documented that fear is not a good advisor because it tends to block our most powerful intellectual capabilities.

Under stress, developers can impose excessive control on their work as a direct response to fear. When frustrated because things are not going well, the definition of achievement shifts from solving the client's problem towards dealing with a self-assigned, self-created fire-fighting problem as a way to boost morale and self-esteem, with the ultimate goal of squelching fear.

The most common example of fear-induced responses is a tendency to draw design time distinctions at run time by means of `ifTrue:ifFalse:`. In general, fear induces a bias towards issuing orders to obtain behavior.

But being successful requires accepting the fact that one will have to deal with incomplete understanding most of the time. This is normal. Do not let emotions ruin the potential for finding a good solution[11].

1.4.6 Code metrics

Maximum context size

Typically, programs of even modest size require the work of a team of developers. In this context, for the project to succeed, it is imperative to keep the cost of changing the emergent properties of the system at a minimum.

The cost of changing a program clearly depends on the difficulty associated with understanding and changing code written by a member of the team. This can be measured directly in terms of the size of the context needed to completely grasp the intentions behind each piece of code.

Since developers are not omniscient, the maximum size of the contexts we can manage is an important amount to keep in mind. Amazingly, as far as computer programming is concerned, we are limited to contexts no more than 7 ± 2 distinctions in size[12].

Interesting: the amount is bounded between "one hand" and "almost two hands".

[11] In sports, letting the fear of winning get in the way is referred to as *choking*. We can recognize this when we watch games, and therefore we have the capability to see things like that in ourselves as well. We may not like to recognize it within us, but we cannot fix what we do not acknowledge as an issue.

[12] See The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information, by George A. Miller (The Psychological Review, 1956, vol. 63, pp. 81-97).

Think of the implications carefully: it is possible to think of many more distinctions implicitly by means of considering a *collection* of related things, or *chunking*. This amounts to storing many distinctions inside a new distinction, and then only considering the outermost boundary. However, even when we take advantage of chunking, what happens is that we are limited to considering contexts with no more than 7 ± 2 top level distinctions.

You can see this for yourself by performing some simple experiments. For example, look at a piece of rough sheet rock. How do you conclude it is rough? Of course, the bumps are quite evident to the eye. But how many can you concentrate on *at the same time*?[13]

So in reality, the fact is you conclude the sheet rock is rough because you sample a few representative locations and find that all of them have bumps. However, at no point in time you can individually distinguish more than just a few of them.

In the same way, look at a dotted tile. How do you know it is dotted? Because you sample a few places and find dots in all of them. Then, you extrapolate and conclude the whole tile is dotted. But you will see that you cannot focus on more than just a few dots at the same time.

If you have travelled by plane, then perhaps you have flown over a city at night. Ah, such a pretty view, all the lights on the ground — and although you may realize there are thousands of lights, you can concentrate only on a most limited amount of them in the same instant.

Finally, look at this page. Clearly, it has many letters printed on it. How many can you focus on at the same time? Regardless of the distance at which you hold the book from your eyes, you will find you cannot distinguish more than a few of them simultaneously.

When the context we are working with requires dealing with more distinctions than we can handle, we follow the fate of the juggler who fails to keep all the balls in the air. Thus, we must chunk distinctions together if we are to avoid failure.

Regardless of whether we like it or not, this is how modest and limited our thought processes really are. At least we have enough to be able to reflect on these constraints. Thus, we can conclude that, when writing computer programs

[13]The members of the Piraha tribe in Brazil count in terms of "one", "two", and "many". The word "one" can also mean "a few", and "two" can also mean "not many". They use a single word for "he" and "they". Standard quantifiers such as "more", "several" and "all" do not exist in their language. Numerical Cognition Without Words: Evidence from Amazonia, by Peter Gordon (Science 15 October 2004; 306: 496-499).

meant to be read and understood by people, it is extremely important that in every context from where messages are sent *there are no more than* 7 ± 2 *top level distinctions*[14].

This implies that any context with more than 7 ± 2 top level distinctions is hard to understand by definition. Therefore, we should take it easy and avoid asking too much of ourselves — let's just settle on 7 as the maximum size of the contexts we can think of.

Compliance

The consequences are all-encompassing, devastating. To begin with, no object should have more than 7 instance names, because otherwise the implementation of its behavior would be hard to understand by definition.

In addition, no method or block should reference more than 7 names. This includes references to the special names `self` and `super`, as well as references to argument names, temporary names, instance names, shared names[15], pool names and global names. Within each code closure, all of these contribute towards the limit. This reasoning can be extended further by limiting the amount of sentences in a code closure to no more than 7.

It is painfully clear that there is not a lot of wiggle room. You may even feel that the maximum context size is way too small. In fact, sometimes developers *cheat* and get around this limit by inlining several Smalltalk sentences into a long one when it is not absolutely necessary. Besides giving the impression of being clever, it is frequently argued that this style should be easier to understand. The justification used is that all the inlining causes things like temporary names to disappear, and in this way just a few names are used.

However, this comes at the expense of referencing implicit objects inside parentheses, blocks, and arguments of keyword messages. Because of our previous discussion, for any sentence to be understandable it must reference no more than 7 objects, because otherwise the context defined by the sentence itself would be difficult to understand by definition. Therefore, the very nature of sentences consisting of many other inlined sentences frequently makes them difficult to understand by definition.

This line of reasoning can be extended further. Since labeling is such a fundamental action, it should be clearly distinguished from sending messages.

[14]This, of course, applies to *any* work we produce for the benefit of others.

[15]Formerly known as class variables. Keep in mind that these names qualify as a global name in the context of all instances of a class.

To avoid mixing messages with assignments, the labeling should be performed at a sentence boundary. The value of a sentence can be labelled only after the value is determined. Hence, the only kind of allowable assignment should be the following one:

```
aName := aReceiver message: withArguments.
```

The time it takes to debug an embedded assignment quickly outweighs any performance benefits obtained. If performance is critical, write a primitive instead.

In particular, assignment inside parentheses ought to be strictly forbidden.

Final remarks

As we have seen, our ability to handle contexts of names is so modest and limited that frameworks like early-bound type systems are, on their own right, on the brink of being hard to understand by definition. Trying to implement something on top of things like that is only going to make the headache worse.

In the same way, our own shortcomings dramatically show why Smalltalk can be such a powerful tool: it does not have so called features that make programs hard to understand *by definition*. This is not a deficiency. Rather, it is an essential quality[16].

Finally, it should be noted that code metrics such as the count of classes are inherently misleading. The count of lines of code per method is better, but evidently it is not strict enough since it fails to penalize methods consisting of heavily inlined sentences.

1.5 Exercises

Exercise 1.1 [05] If you expressed no intentions, how many distinctions would you be able to think of?

Exercise 1.2 [10] What would happen to the laws of form if it were possible to perform an assignment on an argument name?

Exercise 1.3 [10] What would messages need to do in order to break the law of crossing?

[16]For a more detailed discussion comparing Smalltalk to Java in particular, read Why Java Isn't Smalltalk: An Aesthetic Observation, by Sigrid E. Mortensen. However, note that the assumed asymptotic performance of developers probably does not hold.

Exercise 1.4 [27] Change VisualWorks' compiler to let it recognize assignments with `:>` as well as with `:=`.

Exercise 1.5 [15] Consider the following message send.

```
self performActionWith: self
```

Checking for senders of `performActionWith:` finds just the message send above. What, if anything, is wrong with this?

Exercise 1.6 [22] Remove all conditional logic from the following hypothetical method.

```
Number>>decimalPrintString

  (self isKindOf: Fraction) ifTrue: [^self asFloat printString].
  (self isKindOf: Integer) ifTrue: [^self printString, '.0'].
  ^self printString
```

Measure the change in execution speed due to your changes with receivers of several subclasses of `Number`. Does using `self class ==...` instead of `self isKindOf:...` in the code above make any difference?

Exercise 1.7 [24] *(Leandro Caniglia)* Consider what would happen if you tried to implement boolean logic with the integers 0 and 1 as discussed in the section regarding the special object array.

Exercise 1.8 [46] Write a tool that determines the sizes of the name contexts defined by arbitrary sets of code. Use the measurements to produce a score that indicates how understandable the code is.

Make sure the penalties for going over the context size limit grow faster than a linear function of the excess.

Exercise 1.9 [07] If the names inside an instance are called instance names, then cleary the expression *temporary names* is not very clear because it does not refer to the place where the names in question appear. Find a better name for temporary names.

Exercise 1.10 [14] Apply the 7 ± 2 metric guidelines and decide whether to use accessors or not.

Exercise 1.11 [15] Regarding the use of an implicit `self` in statements, Alan Kay commented that since under these assumptions one would have

```
(self self) == self
```

then one can use or omit `self` as appropriate. What could be the motivation for the expression above?

1.5.1 More difficult exercises

This section has exercises which I have collected over time from a variety of sources. These exercises require a higher than average understanding of Smalltalk to be solved, and are excellent material for thought. Study them carefully — you may be asked to solve similar problems in job interviews!

Pencil and paper exercises

While using a computer to solve these exercises is acceptable, you will get the most benefit from them if you tackle them in your head.

Exercise 1.12 [24] Describe the behavior of the sort block below.

```
[:x :y | ^x someValue < y someValue]
```

Exercise 1.13 [16] What is the answer of the message below?

```
SmallInteger class>>test

  ^super class == self superclass
```

Exercise 1.14 [18] What is the answer of the message below?

```
SmallInteger>>test

  ^self class == super class
```

Exercise 1.15 [28] Implement the following message.

```
Object>>interesting

  ^true
```

What happens when you evaluate the expression below?

```
Object new interesting
```

What about this other expression?

```
Object interesting
```

Exercise 1.16 [10] Is `aFraction negated` always a fraction?

Exercise 1.17 [10] Is `aFraction reciprocal` always a fraction?

Exercise 1.18 [10] Is `aFloat reciprocal` always a float?

Exercise 1.19 [10] Is `aSmallInteger negated` always a small integer?

Exercise 1.20 [10] Is `aLargeInteger negated` always a large integer?

Exercise 1.21 [10] Is the sum of small integers always a small integer?

Exercise 1.22 [10] Is the sum of large integers always a large integer?

Exercise 1.23 [26] Take a look at the following test method.

```
testExample

  self
    shouldnt: [SomeClass new performAction]
    raise: ActionException
```

The intention of `shouldnt:raise:` is to fail if evaluating the block raises the exception given. This means that if `performAction` is not implemented, this test will pass because the exception corresponding to a message not being understood is not `ActionException`. What, if anything, is wrong with this?

Live Smalltalk exercises

Solving these exercises will definitely require a computer. Give them your best effort before looking at the solutions.

Exercise 1.24 [31] In Classic releases of ObjectStudio Smalltalk (those prior to its inclusion in VisualWorks), some messages have no source code. One such message is `Object>>class`. The net result is that the number of the primitive that returns the class of an object is not readily available.

Let's say you create a subclass of `nil` called `AbstractProxy`. How could you make instances of `AbstractProxy` and all of its subclasses respond to and implement the behavior of `class`? Write only one method. Hardcoding class names or overloading `doesNotUnderstand:` are not allowed.

Exercise 1.25 [36] Since the expression below works,

```
2.0d raisedTo: -1070
```

one would expect that the one below would work as well.

```
1.0d / (2.0d raisedTo: 1070)
```

And yet, it does not. Why?

Exercise 1.26 [29] What, if anything, is wrong with this?

```
0.4 = (2/5) "true"
```

Exercise 1.27 [35] Create a new class called `Test`. In a workspace, evaluate the following code.

```
t := Test new.
Smalltalk at: #test put: t.
Test become: String new
```

After doing this, trying to send any message to `t` will cause the virtual machine to crash because the class of `t` will be the empty string. How do you clean up the broken object without sending a single message to `Smalltalk`? Note this means you cannot inspect `Smalltalk` either.

Exercise 1.28 [33] *(Alan Knight)* Consider the binary search implementation written in `SequenceableCollection` and shown below.

```
SequenceableCollection>>binaryIndexOf: anObject

  ^self
    binaryIndexOf: anObject
    from: 1
    to: self size
    ifAbsent: [0]


SequenceableCollection>>
  binaryIndexOf: anObject
  from: start
  to: stop
  ifAbsent: aBlock

    | pivotIndex pivotObject newStart newStop |
    start > stop ifTrue: [^aBlock value].
    pivotIndex := start + stop // 2.
    pivotObject := self at: pivotIndex.
    pivotObject = anObject ifTrue: [^pivotIndex].
    (self binarySearchIs: anObject lessThan: pivotObject)
      ifTrue: [newStart := start.  newStop := pivotIndex - 1]
      ifFalse: [newStart := pivotIndex + 1.  newStop := stop].
     ^self
        binaryIndexOf: anObject
        from: newStart
        to: newStop
        ifAbsent: aBlock


SequenceableCollection>>
  binarySearchIs: anObject
  lessThan: anotherObject

    ^anObject < anotherObject
```

```
SortedCollection>>
  binarySearchIs: anObject
  lessThan: anotherObject

    ^self sortBlock value: anObject value: anotherObject
```

It seems well written, but it does not handle this case well at all.

```
| collection |
collection := SortedCollection
  sortBlock: [:x :y | x key <= y key].
1 to: 1000 do: [:each | collection add: each -> each].
1 to: 50 do: [:each | collection add: 42 -> each].
collection binaryIndexOf: 42 -> 6
```

Modify the implementation of this binary search so that it works for the test case shown above.

Chapter 2

Complex conditions

2.1 Preliminaries

Names are such an important form of intention communication that their design requires the best effort possible.

For example, names should be chosen in a very direct manner, referring to distinctions in the terms of the problem at hand. Names require justifications and comments when they are not chosen carefully. There should be no effort spent into translating the language of the task into more computer-oriented jargon.

Names are read 100 *to* 1000 *times more often than they are written. Abbreviations waste more reading time than the writing time they save.*

Moreover, names should not contain artificial abbreviations. One thing is to adopt the language of the problem. Another is to abbreviate because of the apparent time saved by avoiding a few keystrokes.

If the argument for abbreviations were valid, should this book be written as if it were a cellphone text message? I could easily claim that my typos and contractions would save me time, and that they would get the book to you sooner.

But in the same way readers would disapprove of such a writing style, any savings are made by abbreviating names in Smalltalk would quickly vanish when examining or changing the same code at a later point in time.

The implementation of behavior is a much more permanent expression of intention than a quick scribble. At the same time, it must allow change at a low cost. Therefore, there is no excuse for laziness nor carelessness.

Names should not betray the private nature of implementation details either. For example, a message called `singleton` tells the senders that they will obtain the unique instance. This causes problems immediately. As long as senders of `new` receive an instance as advertised, why do they need to know it is a singleton? Since the behavior in both cases is to obtain an instance, why should there be

more than one way to invoke it? Why should senders of either `new` or `singleton` require any maintenance if the implementation behind instance creation were to change?

More deeply, naming a piece of behavior in terms of `singleton` takes the meaning of the message away from the context of the receiver, and puts it in the context of the sender. This allows the sender to cross the boundary of the receiver, something that only messages should do.

It may be that `new` is not always the right name either. The selector `observe` is a more appropriate name for the behavior. However, `new` is the de-facto standard selector for obtaining instances.

Ad-hoc rules such as when to send `new` and when to send `singleton` take our precious time and effort. As such, they should be avoided.

From a wider point of view, the message `singleton` is merely an example of a pervasive pattern. In very general terms, computer programmers are taught to choose names that repeatedly shortchange them. Up to some point, all of us were told to put technique first, and to disregard the work it takes to obtain a clear understanding of the underlying problem. This is how we passed our exams, hence applying recipes is what we became good at first.

Our own background creates the opportunity for peer pressure to keep us solving the same old exercises during our whole career. In this context, some fellow developers may even perceive choosing truly revealing names as some sort of treacherous sin. Clearly, that style does not look like the one in the approved cookbook.

In this kind of environment, challenging the conditioning to automatically censor oneself by discarding perfectly good names can come down to a display of sheer courage. This is why it also takes considerable effort to accept the names one should really adopt, the names one truly uses to think.

Truth will set you free, but first it will make you feel miserable.

However, these shackles are not mandatory because in the end, our lack of freedom is just self-imposed. If one gains enough confidence from a liberating experience, true and profound change becomes first possible, then irresistible.

Being a piano virtuoso means nothing if one cannot put one's own proficiency to the service of communication. In the same way, transcendental computer work can only be accomplished when programming skill is subordinated to the clear expression of ideas in a computer.

We use names to articulate thoughts. Mastering a proper naming technique to precisely convey purpose is essential to achieve success.

What constitutes an example of a good name in Smalltalk is a mixture of how well it reveals the intentions behind it, and how well it interacts with other names, such as when selectors are used to put sentences together.

This chapter shows how valuable the benefits of good names can be by tackling the expression of complex boolean conditional code in Smalltalk.

2.2 Complex boolean expressions

2.2.1 Motivation

There are a number of ways to handle complex boolean expressions in Smalltalk. However, they tend to be quite verbose or hard to maintain. Let's consider the following conditional statement.

```
aHasDot := a includes: $.
bHasDot := b includes: $.
^(aHasDot and: [bHasDot])
  ifTrue: [a < b]
  ifFalse:
    [
      (aHasDot not and: [bHasDot not])
        ifTrue: [a > b]
        ifFalse:
          [
            (aHasDot not and: [bHasDot])
              ifTrue: [true]
              ifFalse:
                [
                  (aHasDot and: [bHasDot not])
                    ifTrue: [false]
                ]
          ]
    ]
```

What a lovely piece of code! After a while of analysis, we can see it is choosing a particular action for each combination of `true` or `false` for both `aHasDot` and `bHasDot`, as shown in the table below.

Value of `aHasDot`	Value of `bHasDot`	Statement value
`true`	`true`	`a < b`
`true`	`false`	`false`
`false`	`true`	`true`
`false`	`false`	`a > b`

Once we reach this point, we can rewrite the return statement using fast-out logic. This makes the whole thing much easier to understand at first sight.

```
(aHasDot and: [bHasDot]) ifTrue: [^a < b].
(aHasDot and: [bHasDot not]) ifTrue: [^false].
(aHasDot not and: [bHasDot]) ifTrue: [^true].
(aHasDot not and: [bHasDot not]) ifTrue: [^a > b]
```

Rewriting code can be the expression of better understanding. As such, it should not be punished.

This is much shorter and clearer! We can also perform some reordering...

```
(aHasDot and: [bHasDot]) ifTrue: [^a < b].
(aHasDot not and: [bHasDot not]) ifTrue: [^a > b].
(aHasDot and: [bHasDot not]) ifTrue: [^false].
(aHasDot not and: [bHasDot]) ifTrue: [^true]
```

Now that the similar statements are together, it is easier to see that whether the answer is `true` or `false` in the last two lines is equivalent to asking whether `bHasDot` is `true`[1].

```
(aHasDot and: [bHasDot]) ifTrue: [^a < b].
(aHasDot not and: [bHasDot not]) ifTrue: [^a > b].
^bHasDot
```

This is certainly better than the original code. Nevertheless, it is not as good as it can be because there is a lot of redundancy: the intermediate results are referenced 5 times, 6 logical operators are being used, and the code still has a bunch of parentheses and square brackets.

In the middle of this, what is our original intention? And where is it clearly revealed in code above?

2.2.2 Intention

Let's examine how clearly we are communicating our intentions. How do we read the following piece of code?

```
(aHasDot and: [bHasDot]) ifTrue: [^a < b].
(aHasDot not and: [bHasDot not]) ifTrue: [^a > b].
^bHasDot
```

[1]The case of `aHasDot` and `bHasDot` having the same value cannot happen at this time. Since `aHasDot` and `bHasDot` do not have the same value, then one is `true` and the other is `false`. We could have chosen `aHasDot not`, but `bHasDot` is simpler.

Like this, maybe?

```
If aHasDot and bHasDot, ^a < b.
If aDoesNotHaveDot and bDoesNotHaveDot, ^a > b.
^bHasDot
```

It is possible to do better than that.

```
Given aHasDot and bHasDot:
  if they are all true, ^a < b;
  if any of them is true, ^bHasDot;
  otherwise, ^a > b
```

The phrasing above seems to make it easier to grasp what the intentions are. But how do we articulate these intentions more clearly in Smalltalk now? Let's go over the last expression. The first line roughly implies that `aHasDot` and `bHasDot` are together, as if they were in a collection. Then, a decision is made such that if all the conditions in the collection given are `true`, then do this. If at least one of them is `true`, do that. In any other case, do so and so.

But clearly, the receiver of all those conditional messages is the *collection of the conditions given.*

2.2.3 Distinctions

If the goal is to be more expressive, the first step is to create collections of simple conditions. What shape could these collections have? Looking at what Smalltalk already provides, boolean values can be concatenated using the `|` and `&` messages. However, they are not very useful because all the conditions must be evaluated before a decision can be made. This disallows lazy evaluations of conditions (also known as short-circuit logic), which is a very nice feature to have.

Note how making a decision can be thought of in terms of drawing a distinction.

The messages `and:` and `or:`, on the other hand, are a better match. They are sent to booleans with block arguments. The block is evaluated only if it is necessary. This is an excellent example of the meaning of the message being determined by the context of the receiver.

When there are many conditions, however, `and:` and `or:` foster deep block or parenthesis nesting. This makes debugging, maintaining and changing code unnecessarily difficult, as nesting causes mistakes to become very easy to make.

A more subtle problem with `and:` and `or:` is that they are keyword messages. This means they get evaluated last, when usually the messages that need to be

evaluated last are the `ifTrue:ifFalse:` action determinations. Using `and:` and `or:` automatically causes a pair of parentheses.

Mandatory pairs of parentheses can be a symptom of deficient selector design. If they are necessary all the time, they are no more than superfluous syntactic sugar. In some cases, they may even represent a distinction that has not been drawn in a sharp and focused way.

Sometimes, these implicit distinctions can be made more explicit by means of a more careful choice of selectors. Let's explore this issue thoroughly.

The constraints will lead us to a solution because they cannot help themselves. Choose the initial conditions carefully, let the system evolve, and watch it happen.

The conditions that need to be joined together seem to be blocks. This is pretty much a requirement in order to support short-circuit logic.

Because the last messages that should be evaluated have keyword selectors, and because automatic parentheses should be avoided, there is no other way out other than grouping blocks using a binary message.

There is something in Smalltalk that addresses similar requirements: the comma message in `Collection`. It behaves as shown below.

Expression	Value
`#(3 4), #(5)`	`#(3 4 5)`

Condition collections could follow the same pattern, satisfying at the same time the requirement that conditions are evaluated in the order they are added.

Expression	Value
`[true], [false]`	a collection of conditions

To allow collections with more than two conditions, both blocks and the collection will have to understand the comma message. In both cases, the argument will be a single block. Using polymorphism between the piece and the composite avoids parentheses by letting the meaning of the message be defined by the context in which it is received. Thus, we can write:

Expression	Value
`[true], [false]`	a collection of conditions, let's call it `x`
`x, [true]`	`x`, with the condition `[true]` added at the end

Let's go over the small details with care. It is time to design the small pieces so that the whole works harmoniously.

Since in Smalltalk even blocks are objects, we can simply implement the comma message in the class of blocks, `BlockClosure`.

The condition collection should be a subclass of `Object` so the behavior of the comma message in `Collection` is not broken (the implementation of the comma message in `Collection` expects a collection as an argument). Let's give the subclass of `Object` the name `ComplexCondition`. These beautiful methods become possible.

```
BlockClosure>>, aBlock

  ^self asComplexCondition, aBlock


BlockClosure>>asComplexCondition

  ^ComplexCondition new, self
```

By bootstrapping the complex condition with the comma message, we avoid instance creation messages with keyword selectors such as `with:`.

2.2.4 Behavior

What behavior could we expect from a collection of conditions? First of all, it should be able to tell if its conditions are all `true`, or all `false`, or if there is at least one of each. The following unary messages suggest themselves.

```
allTrue
allFalse
anyTrue
anyFalse
```

Although these four messages are essential for the implementation, they are not very useful on their own because chances are `ifTrue:ifFalse:` will be sent to their answer.

If we always need to send the complex condition a unary message before the really interesting stuff can be evaluated, the unary message becomes syntactic sugar. On top of it, since comma needs to be evaluated before the unary messages can be sent, such an arrangement forces a pair of syntactic sugar parentheses.

But the most important observation, however, is to notice the subtle way in which things can get in the way of expressing intention. The keyword message will draw a distinction on the collection of conditions. There is no excuse for letting anything get in the way. Never ever let any toll booth become mandatory for messages that draw distinctions!

In other words, since we use distinctions to interpret the world, language expressions that do not refer to a distinction are superfluous and misleading. Hence, such expressions should be eliminated.

Since messages that do not draw distinctions are toll booths by definition, *no message should be sent unless it draws a necessary distinction.*

So instead of imposing mandatory syntactic sugar and obstructing the work of intentions, we could *combine* unary messages with `ifTrue:ifFalse:` and solve all our problems at once. This is a bold move against unnecessary complexity. As a result, we obtain the following single keyword messages.

```
ifAllTrue:
ifAllFalse:
ifAnyTrue:
ifAnyFalse:
```

We can extend the pattern to two argument keyword messages.

```
ifAllTrue:otherwise:
ifAllFalse:otherwise:
ifAnyTrue:otherwise:
ifAnyFalse:otherwise:
```

Colloquialisms that are not vague can help us to stay away from computer jargon, which tends to be removed from the issue at hand anyway.

Notice the `otherwise:` wildcard that catches all the cases for which the first keyword did not apply. The keyword `otherwise:` itself is quite colloquial and almost informal. However, it is revealing and precise enough for its purpose.

For the sake of completeness, these messages should be extended to three arguments as well.

```
ifAllTrue:ifAnyTrue:otherwise:
ifAllFalse:ifAnyFalse:otherwise:
```

The keyword `otherwise:` avoids `ifAllFalse:` at the end of the three keyword selector, which would look almost redundant.

With these messages now available, it is possible to write the original piece of code from the beginning of the chapter as shown below.

```
^[aHasDot], [bHasDot]
  ifAllTrue: [a < b]
  ifAnyTrue: [bHasDot]
  otherwise: [a > b]
```

Compared to the previous shortest-possible Smalltalk expression, the piece above does not require square bracket or parenthesis nesting, repeats a single condition only once, does not make excessive use of logical connectors, and does not need

fast-out statements.

This represents a lot of improvement. In fact, the style is so much clearer that writing it without the temporary names would not seriously impair its readability.

```
^[a includes: $.], [b includes: $.]
  ifAllTrue: [a < b]
  ifAnyTrue: [b includes: $.]
  otherwise: [a > b]
```

What makes it possible to come up with the code above is a strong desire for crystal clear expression. Combining selectors in a way that avoids syntactic sugar is a very important and useful skill to have — it can easily scratch a lot of entanglement itch!

But the best part is that this ability is not just one of a kind. There is a whole family of selector design techniques waiting for us to put them to good use. Our boolean expression example still has a lot to give. In fact, we are not even close to being done.

2.2.5 Extended behavior

Let's consider the case of typical messages in the testing protocol. Sometimes, they are implemented in terms of the value of a boolean expression, such as the one below.

```
hasAllProperties

  ^self hasPropertyA and:
    [self hasPropertyB and:
      [self hasPropertyC]]
```

Using complex conditions, the method could be rewritten as follows[2].

```
hasAllProperties

  ^([self hasPropertyA],
    [self hasPropertyB],
    [self hasPropertyC]) allTrue
```

[2]Writing `ifAllTrue: [true] otherwise: [false]` avoids parentheses but looks silly.

Choose your constraints, Luke.

However, the syntactic sugar parentheses are back. Let's examine this issue in detail. A binary message is being used to combine the conditions together. If we want to ask for a result after the individual conditions are combined, it should be done with a keyword message. In this way, the keyword message would be evaluated after the comma message without needing parentheses.

Let's hold on for a moment. We have seen the pattern of *letting the precedence of selector types make parentheses unnecessary* enough times already. It is time to spell it out in general terms.

I am not aware of having seen this style principle anywhere else. Highlight this paragraph.

The names of messages should be chosen such that keyword messages form the skeleton of Smalltalk sentences, while receivers and arguments should be obtained by means of existing names and unary or binary selectors. Doing so tends to eliminate the need for syntactic sugar, and makes it much easier for sentences to read like written English.

This is an extremely important consideration to keep in mind when designing selectors, as the results obtained from following it are of the utmost importance.

Back to our problem. The boolean expression in `hasAllProperties` uses the logical connector `and:`. Somehow, we have to let the complex condition know which logical connector it should use to bind its conditions together. However, that should not be done with a keyword message along the lines of `bindAllConditionsWith:` because it would force us to use parentheses when we follow up sending a message like `ifAllTrue:`. So that approach does not work. Moreover, using a binary message would cause the collection to remember the logical connector, adding unwanted complexity. Finally, unary messages would introduce the kind of syntactic sugar we want to avoid.

Despite how hard we try to design the messages, either the order of evaluation causes problems, or syntactic sugar appears, or the pattern simply does not work at all. No matter what we come up with, the result always seems contrived and unnatural.

To summarize, *using multiple messages is causing too much difficulty.* While this may seem obvious and perhaps a bit harsh, the phrasing is key because it hints at a problem looking for a solution[3].

[3]Note how stating a requirement makes it so much easier to find the answer! There are numerous examples of this in the history of mathematics. For instance, the formula to find the solutions of $ax^2 + bx + c = 0$ was known for many years. But when somebody found that a formula for $ax^3+bx^2+cx+d = 0$ was possible, numerous alternate derivations were "suddenly" found. In a relatively short time, a formula for $ax^4 + bx^3 + cx^2 + dx + e = 0$ was discovered as well. These developments were possible because a clear expression of the requirements was available to people who did not let themselves be intimidated by the challenge.

And indeed, if we examine the situation carefully, we can see the issue is that multiple messages are always more than what is needed to draw a single distinction. This is a concrete example of how much toll booths can have a profound yet almost invisible influence in how effectively we can communicate our intentions.

Why should anyone have to put up with this? What these productivity taxes deserve is a boldly driven bulldozer.

"If you want to become a mathematician, you must learn to become angry."
—Horacio Porta

So now, if we had to make do with one message only, what could we do? It seems the only way out would be to use a keyword message that requests the value of the complex condition when `and:` is used as a logical connector.

Problems are hard to solve when they are left unstated. This is because in the absence of explicit and revealing expression, it is not possible to draw distinctions properly.

There is a feedback cycle between requirements and solutions because the constraints define the size of the answer space in the same way that a system of linear equations implies a vector space of solutions.

If there are too many solutions, it can be a sign of too little expressed intention because the distinguished result set is too large. To address this, you need either more requirements or stricter conditions. In either case, you can typically obtain additional constraints to finetune your solution selection by recognizing a class of unwanted solutions, and then turning its distinguishing characteristics into a requirement of their opposite. On the other hand, if no solution can be found, you must either drop some constraints or replace some requirements with a desirable characteristic of the answers you are looking for. In other words, this is pretty much what you would do when using an index of web pages such as Google.

> *As a rule, said Holmes, the more bizarre a thing is the less mysterious it proves to be. It is your commonplace, featureless crimes which are really puzzling, just as a commonplace face is the most difficult to identify.*
> *—Arthur Conan Doyle*

To summarize, you are managing the size of the solution space by managing the requirements. The difficulty resides in doing this artfully. As far as developers go, we must be skilled at defining issues clearly so that it is easy to find answers, as well as gifted at choosing which particular expression of a problem to state so that the solutions are most beneficial and easiest to find and implement in a computer. In other words, we must learn how to distinguish our constraints by means of our intentions so that we can reach the most interesting emergent properties while minimizing the cost of change.

> *The mere formulation of a problem is far more essential than its solution, which may be merely a matter of mathematical or experimental skills. To raise new questions, new possibilities, to regard old problems from a new angle requires creative imagination and marks real advances in science.*
> *—Albert Einstein*

But if we place this message to the right of all the comma messages, what should its argument be? If we provide the logical connector as an argument, we will cause `ifTrue:ifFalse:` or `perform:` behind the scenes. There is one option left to consider: to place the keyword message *between* all the comma messages. This may seem strange, yet the method below seems quite reasonable.

```
hasAllProperties

  ^[self hasPropertyA] andAllOf:
    [self hasPropertyB], [self hasPropertyC]
```

Note that it is quite advantageous to specify *two* logical connectors. The first connector applies between the receiver and the argument (`and`AllOf:), and the second connector applies between the conditions of the argument (`and`All`Of:`). This suggests the following family of keyword messages.

```
andAllOf:
andAnyOf:
andNoneOf:
orAllOf:
orAnyOf:
orNoneOf:
```

These messages are relatives of `allTrue`, `anyTrue` and `allFalse`. They should be implemented in `BlockClosure` as well as in `ComplexCondition`.

These messages allow mixing the standard `and:` and `or:` logical connectors in a single expression without parentheses or nesting. This is something that regular Smalltalk messages cannot do easily. Let's consider the following piece.

```
hasInterestingProperties

  ^self hasPropertyA and:
    [self hasPropertyB or:
      [self hasPropertyC]]
```

Quick: which conditions are joined by `and:` and which by `or:`? What is the meaning of the expression? Now let's try with five conditions instead...

The care that the code above requires to be understood is not a matter of the particular formatting style chosen to write it. While it seems that rearranging

the square brackets in some way would make the method easier to understand, different formatting rules just cause different types of headaches.

A change in formatting should have no effect on how clearly a piece of code is expressing intention. We all may have our preference, but we should not rely on our style to convince ourselves that what our code means is easier to understand.

Since I do not like the idea of having to deal with random problems caused by random formatting alone, I vote to eliminate these issues outright. Therefore, instead of leaving Boolean Conditionals 101 homework exercises behind, we can just do better and rewrite the whole thing as follows.

```
hasInterestingProperties

  ^[self hasPropertyA] andAnyOf:
    [self hasPropertyB], [self hasPropertyC]
```

Clearly, we can do quite well with just a little bit of effort. In fact, note how the message `and`AnyOf: binds the argument to the receiver, while and`AnyOf`: binds the individual conditions of the argument[4].

It is useful to send these messages to complex condition receivers as well. In this case, we could arrange that the first logical connector of the selector also applies between the conditions of the receiver. Therefore,

```
hasInterestingProperties

  ^(self hasPropertyM and: [self hasPropertyN]) and:
     [self hasPropertyB or: [self hasPropertyC]]
```

is equivalent to

```
hasInterestingProperties

  ^[self hasPropertyM], [self hasPropertyN] andAnyOf:
    [self hasPropertyB], [self hasPropertyC]
```

[4]By letting `true` and `false` understand `andAnyOf:`, we could avoid square brackets around `self hasPropertyA`. However, this slight lack of homogeneity causes problems when it is time to maintain the code. For example, rearranging conditions around messages such as `andAnyOf:` would cause us to remember to add or remove syntactic sugar. Therefore, we choose not to implement those messages for boolean objects.

The processes of avoiding syntactic sugar, and designing selectors such that keyword messages are naturally evaluated last, have a profound impact on how well our intentions are communicated by means of code. They are critical factors that account for the difference between *just good* and *outstanding*.

These considerations imply an ideal shape for statements, in which keyword message sends are rarely inlined inside each other as arguments by means of parentheses. The shape of an ideal statement is shown below.

```
newDistinction := existingNames, unary or binary expressions
  receivingKeywordMessagesWithArgumentsLike: namedDistinctions
  or: binary + expressions
  or: unary expressions
```

This is such a powerful pattern to follow when writing code, that it is often preferable to add temporary names instead of using parentheses[5].

2.2.6 Flexibility

Nearly every message implemented in `ComplexCondition` eventually depends on the messages `allTrue`, `anyTrue`, `anyFalse` or `allFalse`. These, in turn, are implemented in terms of four kernel messages.

```
allTrue

  ^self ifAllEvaluateTo: true answer: true


anyTrue

  ^self ifAnyEvaluatesTo: true answer: true


anyFalse

  ^self ifAnyEvaluatesTo: false answer: true
```

[5]Of course, sentences should still be bound by the limit of 7 distinctions per statement. This restriction makes it prohibitive to abuse receiver or argument expressions by means of keyword message inlining or sending several unary or binary messages.

```
allFalse

    ^self ifAllEvaluateTo: false answer: true
```

At first, it may seem that these kernel messages could be improved. We need to distinguish 4 different logic cases corresponding to `allTrue` through `allFalse`. However, the message sends in the methods above allow two cases by means of `All` and `Any` quantifiers in the selectors, times four cases by means of two sets of `true` and `false` arguments, for a grand total of $2 \times 4 = 8$ distinct logic cases. Even though it is an obviously inefficient choice of selector arrangement, by a factor of two nonetheless, let's resist the temptation to refactor for a moment.

Sometimes, the value of boolean conditions depends on whether names refer to objects that are `nil` or `notNil`. And, typically, the implementation of each such condition ends with an `isNil` or `notNil` question. However, if these questions are asked across all conditions, they become mandatory syntactic sugar[6].

Complex conditions can be easily extended to avoid this kind of redundancy. Instead of checking for `true` and `false`, let's create new messages that check for `nil` and `notNil`.

The boolean family of messages exploit the fact that the one word literals `true` and `false` do not contain logical operators themselves. This allows to create selectors which are clear and easy to understand.

On the other hand, using `nil` and `notNil` will create selectors containing logical operators like the ones we have been trying to eliminate with complex conditions. Moreover, selectors such as `ifAllTrue:ifAnyTrue:otherwise:` have more than enough quantifiers and logical values already. Adding `not` to the mix is not going to help. But the worst problem, what really adds insult to injury, is that there is no literal counterpart to `nil` in the system dictionary.

So instead of using `nil` and `notNil`, let's use the fact that the class of `nil` is `UndefinedObject`. If `nil` is undefined, then clearly everything else is defined. Following this pattern produces the selectors below.

```
allDefined
anyDefined
anyUndefined
allUndefined
```

[6] As developers, we should have a Pavlovian reflex associating *perceiving repetitive code* with *make a framework*. It really ought to be a no brainer.

```
ifAllDefined:
ifAnyDefined:
ifAnyUndefined:
ifAllUndefined:


ifAllDefined:otherwise:
ifAnyDefined:otherwise:
ifAnyUndefined:otherwise:
ifAllUndefined:otherwise:


ifAllDefined:ifAnyDefined:otherwise:
ifAllUndefined:ifAnyUndefined:otherwise:


isDefinedLikeAllOf:
isUndefinedLikeAllOf:
```

So let's check how our new selectors work. For example, consider the following method.

```
hasAllRequiredValues

  ^self firstValue notNil
    and: [self secondValue notNil
      and: [self thirdValue notNil]]
```

With our new messages, it can be easily rewritten as shown below.

```
hasAllRequiredValues

  ^[self firstValue] isDefinedLikeAllOf:
    [self secondValue], [self thirdValue]
```

Clearly, these messages have the same benefits of their boolean counterparts.

For complex conditions, the implementation of keyword messages depends on the implementation of unary messages. The defined/undefined family of unary messages is analogous to the boolean family, yet they still rely on the two kernel methods. For example,

```
allDefined

  ^self ifAnyEvaluatesTo: nil answer: false


anyDefined

  ^self ifAllEvaluateTo: nil answer: false
```

The class `ComplexCondition` itself is quite straightforward to implement when designed in terms of an `OrderedCollection` of blocks called `conditions`. For instance,

Strictly speaking, this should be written as `^super new initialize; yourself`.

```
ComplexCondition class>>new

  ^super new initialize
```

If we design `initialize` carefully, we can use it to avoid implementing `with:` on the class side.

```
ComplexCondition>>initialize

  self conditions: OrderedCollection new


ComplexCondition>>, aBlock

  self conditions add: aBlock


ComplexCondition>>ifAllEvaluateTo: aResult answer: aBoolean

  self conditions do:
    [:each |
      aResult = each value
        ifFalse: [^aBoolean not]
    ].
  ^aBoolean
```

```
ComplexCondition>>ifAnyEvaluatesTo: aResult answer: aBoolean

  self conditions do:
    [:each |
      aResult = each value
        ifTrue: [^aBoolean]
    ].
  ^aBoolean not
```

Using `[aBlock value]` instead of `aBlock` avoids VisualWorks' compiler warnings about non-optimized compiled methods.

```
ComplexCondition>>ifAllTrue: aBlock otherwise: otherwiseBlock

  ^self allTrue
    ifTrue: [aBlock value]
    ifFalse: [otherwiseBlock value]


ComplexCondition>>
  ifAllTrue: allBlock
  ifAnyTrue: anyBlock
  otherwise: otherwiseBlock

    ^self
      ifAllTrue: allBlock
      otherwise:
        [
          self
            ifAnyTrue: anyBlock
            otherwise: otherwiseBlock
        ]
```

Finally, messages in `BlockClosure` typically turn the receiver into a complex condition and resend the message. This avoids unnecessary code duplication. For instance,

```
BlockClosure>>orNoneOf: aComposedCondition

  ^self asComplexCondition orNoneOf: aComposedCondition
```

2.2.7 Summary

Regular Smalltalk boolean expressions become intention obscuring when many conditions are involved. The symptoms are deep block or parenthesis nesting, and code redundancy. Techniques such as fast-out logic and truth table analysis are necessary to simplify or even maintain these expressions. These take time and effort to apply properly.

Complex conditions offer a better way to write these expressions. Nesting and code repetition are avoided. The resulting code is inherently intention revealing, smaller, easier to read, and easier to maintain. The implementation is both light and flexible. Above all, complex conditions can save time and effort to a point where regular Smalltalk idioms cannot.

Since polymorphism has a tendency to make design time boolean expressions superfluous, complex conditions tend to be most useful when distinctions must be drawn at run time.

2.3 Exercises

Exercise 2.1 [07] Use complex conditions to rewrite the following method.

```
EligibilityValidator>>validateSomethingHasChanged

  (model determination = model currentEligibility
    and: [model effectiveDate = model currentBeginDate
      and: [model determinationDate = model currentReviewDate
        and: [model reason = model currentOpenReason]]])
          ifTrue: [self complain]
```

Exercise 2.2 [21] How could you simplify boolean expressions built with nested `xor:` messages such as the one below?

```
(((self conditionA xor: self conditionB)
  xor: self conditionC)
  xor: self conditionD)
    ifTrue: [self parityIsOdd]
    ifFalse: [self parityIsEven]
```

Hint: all four conditions are needed, do not focus on the statement's truth table.

Exercise 2.3 [16] Simplify the expression below.

```
(aHasDot and: [bHasDot]) ifTrue: [^a < b].
(aHasDot not and: [bHasDot not]) ifTrue: [^a < b].
^bHasDot
```

Exercise 2.4 [17] Implement the message `implies:` for boolean receivers and block arguments without using conditional or boolean logic statements.

Exercise 2.5 [14] Extend the message `implies:` from the previous exercise by adding keywords so that sending `ifTrue:ifFalse:` does not require parentheses.

Exercise 2.6 [14] Implement the message `implies:` from the previous exercise in terms of a binary message so that sending `ifTrue:ifFalse:` does not require parentheses.

Exercise 2.7 [31] Find good `nil` family selectors equivalent to `andAnyOf:` and `orAllOf:`.

Exercise 2.8 [28] What interesting transformation could be performed on the piece of code below?

```
[
  self conditionA
    and: [self conditionB
      and: [self conditionC]]
]
  whileTrue: [self action]
```

The current implementation implies that things like `true xor: aBlock` would appear to work, yet the answers from `xor:` would be misleading!

Exercise 2.9 [18] In VisualWorks, `xor:` is implemented as follows.

```
Boolean>>xor: aBoolean

  ^(self == aBoolean) not
```

Reimplement the behavior above without sending `ifTrue:ifFalse:` nor using parentheses or comparison messages such as `==`.

2.4 SUnit tests for ComplexConditions

Certain Smalltalk messages are regularly assumed to be infallible. When we use the debugger to examine a piece of code, for example, one never doubts that `ifTrue:ifFalse:` is working properly.

This assumption does not require a lot of faith. Evidently, if something like `ifTrue:ifFalse:` were broken, the image would crash quickly.

Since the functionality offered by complex conditions is so close to that of `ifTrue:ifFalse:`, their behavior should also be impeccable. `SUnit` can be used to establish this beyond all reasonable doubt[7].

At first, it seems that a few well selected tests can do. It is easy to imagine a test written to stress a particular algorithm, a specific design choice, or even a critical private method. However, these choices would almost certainly be made with knowledge of the particular implementation. Decisions like these imply that a change in how complex conditions are written could render the tests worthless. This is not acceptable because it greatly increases the cost of change.

The argument above leads to the realization that tests should be designed to *define* the acceptable behavior, and then *enforce* it. No `SUnit` tests should ever be written with the goal of ensuring the correctness of a particular implementation, because tests should never cross the boundary between external behavior and internal details.

That is a very subtle distinction to make. Nevertheless, it is an inescapable conclusion once one remembers that everything that happens in Smalltalk is a message send. As you can see, the principles upon which Smalltalk is built have far reaching consequences.

From a more pragmatic point of view, sooner or later the objects being tested will be used as part of a project. At that point, they will be sent messages. Since that is the only thing that can happen, the particular way in which the behavior is accomplished does not matter.

Thus, in order for complex condition to earn a level of confidence comparable to that of the trustworthy `ifTrue:ifFalse:`, the behavior of all the complex conditions messages should be tested. In addition, all the possible boolean logic cases for each message should be verified as well.

In this kind of situation, one's first tendency might be to write a large amount of test methods by means of copy & paste. That is exactly what should be avoided

[7]This does not mean we should write `SUnit` tests after the functionality exists. The fact that tests are being presented after complex conditions have been described is due to a personal preference in presentation style for this particular material.

at all costs. On the contrary, this is an ideal setup to exercise our design muscles and see where our newly acquired insight will take us.

2.4.1 Unary message tests

Let's begin by considering how the unary messages should be tested. Keeping in mind that four different messages will need to be tested, we could cover all possibilities by evaluating the result of sending the messages `allTrue`, `anyTrue`, `anyFalse` and `allFalse` to all complex conditions consisting of two blocks.

```
[true], [true]
[true], [false]
[false], [true]
[false], [false]
```

Since it appears we will be referring to these conditions quite a bit, let's give them the following names.

```
ComplexConditionTestCase>>true

  ^[1 + 2 = 3]


ComplexConditionTestCase>>false

  ^[1 + 2 = 4]
```

This is a perfect example of how implementing a message is being treated in terms of naming a specific object or a particular bit of behavior.

At first, these methods may seem to be too much pain for too little gain. However, they eliminate the need to write any redundant code and unnecessary square brackets in the methods below.

```
ComplexConditionTestCase>>allTrue

  ^self true, self true


ComplexConditionTestCase>>anyTrue

  ^self true, self false
```

```
ComplexConditionTestCase>>anyFalse

  ^self false, self true


ComplexConditionTestCase>>allFalse

  ^self false, self false
```

Since each condition has an intention revealing name, *it is no longer necessary to evaluate code in one's mind to figure out what is going on.* For example, it is immediately obvious that `allTrue` is the complex condition containing `true` and `true`.

Following the same argument, messages named with plural nouns tend to suggest their answer is a collection. How irritating it is when they also answer `nil`!

Incidentally, since in Smalltalk everything is a distinction and all distinctions can be named, it follows that blocks are distinctions and thus can be named.

The implementation of these methods is quite evident in the context of the message name. When names are carefully chosen, the end product tends to avoid unnecessary homework for whoever needs to read or maintain the code later on.

This is a very dramatic example of the benefits that can be reaped when we *use* the naming capabilities available to us in Smalltalk.

We are creating our own language of expression! From this point of view, using Smalltalk is like doing research.

Back to our problem. Now that we have named all possible complex conditions by means of selectors, we can use the resulting messages to write our first test method. A decent first attempt might look like the one shown below.

```
ComplexConditionTestCase>>testAllTrue

  self assert: self allTrue allTrue.
  self assert: self anyTrue allTrue not.
  self assert: self anyFalse allTrue not.
  self assert: self allFalse allTrue not.
```

This is a strikingly obvious implementation, which by means of its carefully chosen message names avoids all unnecessary square brackets and parentheses. It clearly shows some of the benefits that come with proper selector design.

Let's pause for a moment to put something we already know in a new light. On average, natural language sentences rarely depend on parenthesis nesting to convey their meaning, regardless of the language in which they are written. The evolution of languages shows that, while parentheses have a place, they are

not necessarily frequent. But if that is the case, why should the product of our work be expressed in a non-natural way only because it is *stored* in a computer? That does not make any sense! Thus, instead of rejecting the language design knowledge painstakingly obtained by thousands of years of trial and error, we should try to mimick well-articulated language when writing Smalltalk sentences.

Let's concentrate on our first attempt now. It does not have parentheses, it clearly expresses what it is doing, and it reads well. But things are still far from perfect. The problem is that, even though the implementation of the message above is beautiful, we would have to repeat it pretty much verbatim for the other unary messages.

```
ComplexConditionTestCase>>testAnyTrue

  self assert: self allTrue anyTrue.
  self assert: self anyTrue anyTrue.
  self assert: self anyFalse anyTrue.
  self assert: self allFalse anyTrue not.


ComplexConditionTestCase>>testAnyFalse

  self assert: self allTrue anyFalse not.
  self assert: self anyTrue anyFalse.
  self assert: self anyFalse anyFalse.
  self assert: self allFalse anyFalse.


ComplexConditionTestCase>>testAllFalse

  self assert: self allTrue allFalse not.
  self assert: self anyTrue allFalse not.
  self assert: self anyFalse allFalse not.
  self assert: self allFalse allFalse.
```

Evidently, these methods scale only by means of copy & paste. Even worse, anybody could teach a code generator how to write them. That speaks loudly of their entropy.

Adding unnecessary complexity to eliminate automated competition is not the solution we are after either. In the end, that kind of job security will make

our lives miserable. In fact, it makes our project more likely to fail because it increases the cost of change. We need an approach that provides freedom instead.

Let's examine the similarities between those methods. All of them have four lines. The message being sent to the conditions remains constant within each method. When considering all the methods together, it turns out that all unary messages are sent to all possible conditions. Some occurrences of `not` come and go. What is going on? Could this be... the verification of a truth table like the one at the beginning of the chapter?

Condition	`allTrue`	`anyTrue`	`anyFalse`	`allFalse`
`[true], [true]`	`true`	`true`	`false`	`false`
`[true], [false]`	`false`	`true`	`true`	`false`
`[false], [true]`	`false`	`true`	`true`	`false`
`[false], [false]`	`false`	`false`	`true`	`true`
Logic gate equivalent	AND	OR	NAND	NOR

Indeed it is! It immediately follows that tests could be written more clearly in terms of assertions such as *`allTrue` behaves like the AND gate*. Even better, such assertions sound quite obvious when stated that way.

To facilitate being as expressive as we should, we need to be able to refer to more objects. Since we are going to be talking about boolean function behaviors, let's give them a name. For example,

At first, these methods seem to be missing enough sophistication. I tried implementing `TruthTable` but it felt like overkill at that point, though. This feature can wait until version 2.0.

```
ComplexConditionTestCase>>andBehavior

  ^#(true false false false)
```

In addition, since we are going to be talking about the conditions to which certain messages are sent, let's give them a name as well.

```
ComplexConditionTestCase>>booleanConditions

  ^Array
    with: self allTrue
    with: self anyTrue
    with: self anyFalse
    with: self allFalse
```

Now that we have the elements with which we will implement our tests, we have to find a good selector for our assertions. At this point, it is *extremely*

important to avoid the temptation to translate our clear intent into something that has to do with implementation details, such as shown below.

```
ComplexConditionTestCase>>testAllTrue

  self
    selector: #allTrue
    hasBehavior: self andBehavior
    on: self booleanConditions
```

Clear expression of intentions should always have top priority. With this scale of values in mind, the first thing to notice in the method above is that selectors are the *names* of bits of behavior. Selectors do not behave by themselves. Therefore, the keyword `selector:` is misleading. In addition, the keyword `on:` does not seem to be a good choice considering the intentions that should be suggested to potential readers. So far, no good.

Let's look at the bigger picture for a moment. The point of the test message is that `allTrue` is going to be sent to all of the `booleanConditions` so the results can be compared to the behavior of the AND logic gate. Why should it be hard to say such a simple thing with a selector? Let's throw away all our inhibitions and *write in English for a change*.

```
ComplexConditionTestCase>>testAllTrue

  self
    theMessage: #allTrue
    shouldBehaveLike: self andBehavior
    whenSentToEachOf: self booleanConditions
```

This explicitly illustrates the fact that, especially when using Smalltalk, it is not necessary to translate our thoughts into computerese before the CPU can do the job. This is very important because clear expression of intention reduces the cost of change: we need less time to maintain and write new code, and we are less likely to make mistakes. The customers win because they get what they need sooner, and we win as well because our project does not fail.

But we all know where we got the idea of using single letter names. I do not mean to blame all those computer science or mathematics classes, because they have a lot of value on their own. Without them, we would have no computers to begin with! The issue is that, while their typical notation serves them very well

for their purposes, it is simply not suitable for building large programs. There is no reason why we should continue to punish ourselves by ignoring this most evident fact. It is *us* who have to deal with our own dogfood here, and oh by the way the caviar tray is within reach!

It takes considerable unlearning to write code in plain English naturally. If you push yourself through it, the exchange ideas with others will be easier, and you will definitely not lose your hacker points. By taking advantage of the numerous opportunities you have to put your expressions skills to work, such as every time you write a method, you will achieve liberation in no time.

And certainly, the computer could not care less about how you write your code. Remember: you are catering to your peers.

Let's finish our exploration of unary message test selectors. With the pattern above in mind, the rest of the tests follow suit with ease.

```
ComplexConditionTestCase>>testAnyTrue

  self
    theMessage: #anyTrue
    shouldBehaveLike: self orBehavior
    whenSentToEachOf: self booleanConditions


ComplexConditionTestCase>>testAnyFalse

  self
    theMessage: #anyFalse
    shouldBehaveLike: self nandBehavior
    whenSentToEachOf: self booleanConditions


ComplexConditionTestCase>>testAllFalse

  self
    theMessage: #allFalse
    shouldBehaveLike: self norBehavior
    whenSentToEachOf: self booleanConditions
```

Now it is hard to see how we could be more clear. We have avoided copy & paste, which is always a good move regarding the cost of change. We have also expressed our intentions clearly through the process, which further reduces the cost of change. And there is more: this scheme also addresses the defined/undefined version of the unary messages just as easily! Here are some sample methods.

If a block should evaluate to `nil`, then write it explicitly. It is faster to recognize `nil` than it is to remember what `[ ]` evaluates to.

```
ComplexConditionTestCase>>defined

  ^[17]

ComplexConditionTestCase>>undefined

  ^[nil]

ComplexConditionTestCase>>allDefined

  ^self defined, self defined


ComplexConditionTestCase>>anyDefined

  ^self defined, self undefined


ComplexConditionTestCase>>anyUndefined

  ^self undefined, self defined


ComplexConditionTestCase>>allUndefined

  ^self undefined, self undefined


ComplexConditionTestCase>>nilConditions

  ^Array
    with: self allDefined
    with: self anyDefined
    with: self anyUndefined
    with: self allUndefined
```

```
ComplexConditionTestCase>>testAnyDefined

  self
    theMessage: #anyDefined
    shouldBehaveLike: self orBehavior
    whenSentToEachOf: self nilConditions


ComplexConditionTestCase>>
  theMessage: aMessage
  shouldBehaveLike: expectedResults
  whenSentToEachOf: aCollection

    | results |
    results := aCollection
      collect: [:each | each perform: aMessage].
    self assert: results = expectedResults
```

2.4.2 Keyword message tests

Let's tackle the tests for messages like `ifAllTrue:`. If we fast forward through the discussion from the previous section and apply its conclusions, we reach a truth table again. However, in this case, the table tells whether the argument for each keyword is evaluated or not.

Condition	`ifAllTrue:`	`ifAnyTrue:`	...
`[true], [true]`	`true`	`true`	...
`[true], [false]`	`false`	`true`	...
`[false], [true]`	`false`	`true`	...
`[false], [false]`	`false`	`false`	...
Logic gate equivalent	AND	OR	...

Following the pattern of what we did before, the assertion could go along the lines of *when this selector is sent to these conditions, the argument is evaluated according to this behavior.*

Before we choose the selector, however, what will happen when we want to test messages that take more than one argument? Certainly, tests for messages which have two and three keyword selectors will have to enforce truth tables similar

to the one above. There is not much difference between testing the evaluation behavior of the first argument and testing the evaluation behavior of the third argument. Therefore, let's pass which argument should be tested as an argument to the helper message.

We need to be very careful now. This kind of selectors require the utmost care, in particular with regards to the order of the arguments. For example, passive voice statements frequently run into trouble because they tend to need a piece of selector after the last argument.

You could use active voice and avoid the problem altogether in this particular instance. However, the passive voice example will become useful soon.

```
self makeSure: aPatient "wasExamined"
```

Rearranging the words does not help in these cases because it typically creates improper sentences. If the message has more arguments, however, sometimes it is possible to put passive voice keywords first so their trailing elements can be attached to the next keyword.

```
self
  makeSure: aPatient
  wasExaminedBy: aDoctor
```

Keyword reorderings like the one above often result in much better selectors. With that in mind, let's examine a first attempt at expressing the assertion *when this selector is sent to these conditions, the argument for one of the keywords is evaluated according to this behavior.*

```
self
  selector: #ifAllTrue:
  block: 1
  of: 1
  hasEvaluationBehavior: self andBehavior
  on: self booleanConditions
```

Sigh... the first and fourth keywords give the impression that the selector has an evaluation behavior, when evidently it is a matter connected to the block being passed as an argument. So in reality, the first keyword should be more like `forSelector:`. However, further attempts at shoring up the mess take too much energy. When this occurs, it is a sign of things being broken beyond repair.

The next couple of arguments indicate which block of how many in total is the one being tested. But the receiver could figure out the number of arguments

the message needs by counting the occurrences of `$:` in the selector. This would reduce the rather high count of arguments and make things easier to understand.

Let Smalltalk do the homework. Why should you *count the occurrences of `$:`? Besides, providing the count as an argument is redundant because the selector itself is an argument as well.*

Finally, the keyword `on:` is not clear at all. It is possible to deduce what it stands for by elimination, but its meaning should be spelled out explicitly.

The lengthy discussion the selector above requires is proof that it is not designed well enough. In short, it is *intention obscuring*. Let's try again.

```
self
  when: #ifAllTrue:
  isSentToEachOf: self booleanConditions
  argument: 1
  hasEvaluationBehavior: self andBehavior
```

This improved selector shows significant progress. Notice how passive voice is used in the first two keywords while active voice is used in the last two keywords. This has the effect of making the whole selector look like a logical implication: *if these conditions are met, then this should happen.*

In the context of `SUnit`, the selector even gives the impression of being some sort of `assert:`. This is absolutely correct, and we should recognize it as evidence that shows we are on the right path. By paying attention to these details, we can learn to get an idea of how well we are doing.

Although it may be nice already, this selector can be improved further still. Note that the last two keywords imply that an argument has an evaluation behavior. However, the original assertion implies that the behavior belongs to the evaluation act itself. Since we have already seen how to correct this problem, we make a note to address this in our next iteration.

But why do we make a note instead of fixing this deficiency immediately? Because there could be more issues to address, and we are perfectly able to queue up things we know we should do as long as our to do buffer is able to hold them. When our action item list finally fills to capacity and we are forced to empty it, we will only pay the price for one reimplementation effort to take care of all of the things we noticed, instead of having to perform each individual action on one by one basis. More importantly, we might even be able to skip doing some changes by refactoring code in our minds. If we get used to changing code this way, we can train ourselves to become much more efficient — in particular, because with practice our output buffer will grow in size and thus we will be able to perform more work without needing to spend energy creating and maintaining an external representation of what we are thinking about.

So, what else can we point out about our deficient selector? Well, let's see... the keyword `argument:` seems to imply that what follows will be some actual argument connected to the selector provided next to the first keyword. but that is not the case, so again the selector is a bit misleading. Fortunately, a simple suffix can take care of that. Let's gather up our notes from before and try again.

```
self
  when: #ifAllTrue:
  isSentToEachOf: self booleanConditions
  evaluationOfArgumentNumber: 1
  behavesLike: self andBehavior
```

The selector above shows that we have succeeded in squelching the obsfuscated expression tendencies present in our first attempt. In fact, it is even possible to read this sentence aloud without needing to paraphrase it into English. The way the words that make up the selector have moved since the beginning through each stage of improvement merits special attention.

There is a last detail to take care of. This selector should be similar to our previous testing helper selector. Thus, here is the final version.

```
self
  whenTheMessage: #ifAllTrue:
  isSentToEachOf: self booleanConditions
  theEvaluationOfArgumentNumber: 1
  behavesLike: self andBehavior
```

It is difficult to envision a clearer selector now. Compare the message send above with the first attempt. The difference in clarity is striking. Developers do not have to waste their time translating to and from English anymore. Comments become redundant, useless, and futile. These are some of the powerful consequences that come out from paying attention to seemingly unimportant details.

The test helper message we have arrived at allows to implement a multitude of complex condition tests. The necessary tests for one, two and three argument keyword messages, for both boolean and defined/undefined varieties, can be easily implemented in terms of this single message. In fact, the original implementation of `ComplexConditionTestCase` takes 36 sends divided in 20 test methods to cover all possible cases. Note that if there was any repetition of superfluous expression, it would be magnified with these large amplification factors, and this would bring on rather negative consequences for code maintenance.

The implementation of the test helper message is shown below. Note that the arguments for the message being tested are built in a separate method. This has the effect of keeping the complexity at a manageable level.

```
ComplexConditionTestCase>>
  whenTheMessage: aMessage
  isSentToEachOf: receivers
  theEvaluationOfArgumentNumber: interestingArgumentIndex
  behavesLike: expectedResults

    | arguments executed |
    arguments := self
      noActionArgumentsFor: aMessage
      exceptAt: interestingArgumentIndex
      put: [executed := true].
    receivers with: expectedResults do:
      [:eachCondition :eachExpectedResult |
        executed := false.
        eachCondition
          perform: aMessage
          withArguments: arguments.
        self assert: executed = eachExpectedResult
      ]
```

Notice how what used to be called `aMessage` *in a context where a message is performed (above), is now called* `aSelector` *in a context where the structure of the name is what matters (below).*

```
ComplexConditionTestCase>>
  noActionArgumentsFor: aSelector
  exceptAt: anIndex
  put: anObject

    | argumentCount arguments |
    argumentCount := aSelector occurrencesOf: $:.
    arguments := Array new: argumentCount.
    arguments atAllPut: [nil].
    arguments
      at: anIndex
      put: anObject.
    ^arguments
```

Finally, here is a sample test method for a three argument keyword message. Note how the expected behavior changes with the argument number, and how clearly this is shown in the implementation.

Depending on the intention one wants to suggest, one could also write `shouldBehaveLike:` *instead of* `behavesLike:`.

```
ComplexConditionTestCase>>
  testIfAllDefinedIfAnyDefinedOtherwise

    self
      whenTheMessage: #ifAllDefined:ifAnyDefined:otherwise:
      isSentToEachOf: self nilConditions
      theEvaluationOfArgumentNumber: 1
      behavesLike: self andBehavior.
    self
      whenTheMessage: #ifAllDefined:ifAnyDefined:otherwise:
      isSentToEachOf: self nilConditions
      theEvaluationOfArgumentNumber: 2
      behavesLike: self xorBehavior.
    self
      whenTheMessage: #ifAllDefined:ifAnyDefined:otherwise:
      isSentToEachOf: self nilConditions
      theEvaluationOfArgumentNumber: 3
      behavesLike: self norBehavior
```

2.4.3 Extended keyword message tests

We still have to cover the tests for the extended keyword messages such as `andAnyOf:`. The previous test helper messages do not quite help us in this case. For instance, the message

```
ComplexConditionTestCase>>
  theMessage: aMessage
  shouldBehaveLike: expectedResults
  whenSentToEachOf: aCollection
```

does not allow providing the necessary arguments to `andAnyOf:`. The same occurs with our other four argument test helper message. What we have done so far does not allow providing different arguments to the messages that need to be tested. Since we have new requirements, we have to design another test helper message.

Given a fixed receiver such as `[true]`, what should be verified is the answer to the keyword message when it is sent with different arguments. The assertion has the following appearance: *this condition behaves in this way when it receives this message sent with each of these arguments.*

Unfortunately, the first assertion expression attempt shows clear symptoms of intentions having been translated into computer jargon.

```
self
  block: self true
  hasBehavior: self andBehavior
  whenSent: #andAllOf:
  with: self booleanConditions
```

While it is true that a block will be receiving a message when the code is executed, the keyword `block:` is not a good choice. The test message could easily work for complex condition arguments because of their polymorphism with block closures. In those cases, the receiver would not be a block, yet the message send would make sense. Thus, the keyword `block:` is intention obscuring because it implies implementation details or limitations that are not there, instead of limiting itself to addressing the description of behavior. This is a perfect example of how much damage a somewhat misdesigned selector can do.

We are not done adding action items to our queue yet. As with previous first attempts, this selector asserts that blocks themselves have certain behavior when what is being tested is the answer to a message. Furthermore, the keyword `with:` gives the impression that all of the boolean conditions will be the single argument of `andAllOf:`, when obviously this is not so. But we can use the technique of appending a suffix to address this issue. Let's try again.

Interesting: the message is being treated as a function that assigns behavior to objects. In fact, take a moment and compare the acts of evaluating a function and sending a message. All messages return an answer... a function is defined over its domain... if D is the set of all distinctions, then a behavior β is just a humble function such that $\beta : D \to D$.

```
self
  condition: self true
  behavesLike: self andBehavior
  whenSent: #andAllOf:
  withEachOf: self booleanConditions
```

We are becoming quicker at achieving good progress. Now that we have addressed the selector's shortcomings, we should improve it further by following the pattern of our previous selectors more closely. Let's keep in mind that, as opposed to the previous test message helpers, this selector should highlight the fact that there is a single receiver.

```
self
  whenTheMessage: #andAllOf:
  isSentWithEachOf: self booleanConditions
  toTheCondition: self true
  itBehavesLike: self andBehavior
```

While it seems better, an extraneous `it` has appeared in last keyword of the selector. This is usually a sign of poor keyword order. In the selector above, the sentence progresses too much in the first three keywords and leaves the last one stranded. This is why the last keyword ends up needing to reference the first keyword by means of `it`.

The previous discussion regarding which keyword `it` is referencing sounds quite reasonable. Unfortunately, it is not the only reasonable explanation: isn't `it` referencing the third keyword instead? This ambiguity is an issue because whoever reads the code later on will get stuck pondering things like *what is the reason behind this implementation?*, *what was the author thinking?*, or my favorite *why was it necessary to be clever?*.

Repeatedly imposing this type of homework on your coworkers will cause your name to frequently appear in sentences best left unspoken[8].

Selector keywords are much more than placeholders for arguments.

The problem, as far as our selector goes, is that we cannot tell what is going on because `it` is not specific enough. Let's assume it is indeed referencing the third keyword instead. In that case, the third keyword should read `theCondition:` instead of `toTheCondition:`, because otherwise the sentence as a whole would not look right. But if that change was made, then the focus of the message would shift from *sending a message to an object* to *an object receiving a message*. This would cause even more changes to propagate through the selector.

Messages are much more than C functions.

With these considerations, since what `it` references is already referenced by the rest of the selector, let's try tweaking the sequence of keywords while focusing on the condition receiving a message. The ultimate goal is to make `it` disappear.

[8]Roger Whitney comments the following: keep in mind that even large class assignments are tiny compared to code written in industry. As a result software development in industry is different than completing school assignments. One can survive school using practices that are unworkable in industry, so it is important to pay attention to how you develop code. One big difference between school and industry is that in industry a program is read many more times than in school. In industry other programmers read your code to use it, to find and fix bugs, to modify it and to maintain the code. Often the only ones to read a student program is the instructor or grader. As a result students often do not realize what issues are important in making a program readable.

```
self
  theCondition: self true
  behavesLike: self andBehavior
  whenReceivingTheMessage: #andAllOf:
  sentWithEachOf: self booleanConditions
```

Much better! This improved message enables us to write the remainder of the tests, a sample implementation of which is shown below.

```
ComplexConditionTestCase>>
  theCondition: aCondition
  behavesLike: expectedResults
  whenReceivingTheMessage: aMessage
  sentWithEachOf: arguments

    arguments with: expectedResults do:
      [:eachCondition :eachExpectedResult |
        | result |
        result := aCondition
          perform: aMessage
          with: eachCondition.
        self assert: result = eachExpectedResult
      ]


ComplexConditionTestCase>>testAndAllOf

  self
    theCondition: self true
    behavesLike: self andBehavior
    whenReceivingTheMessage: #andAllOf:
    sentWithEachOf: self booleanConditions.
  self
    theCondition: self false
    behavesLike: self falseBehavior
    whenReceivingTheMessage: #andAllOf:
    sentWithEachOf: self booleanConditions
```

```
ComplexConditionTestCase>>testMultipleAndAllOf

  self
    theCondition: self allTrue
    behavesLike: self andBehavior
    whenReceivingTheMessage: #andAllOf:
    sentWithEachOf: self booleanConditions.
  self
    theCondition: self anyTrue
    behavesLike: self falseBehavior
    whenReceivingTheMessage: #andAllOf:
    sentWithEachOf: self booleanConditions.
  self
    theCondition: self allFalse
    behavesLike: self falseBehavior
    whenReceivingTheMessage: #andAllOf:
    sentWithEachOf: self booleanConditions
```

2.5 Exercises

Exercise 2.10 [18] Examine the consequences of adding an argument to the message below to provide the total count of arguments that `aMessage` needs.

```
ComplexConditionTestCase>>
  whenTheMessage: aMessage
  isSentToEachOf: receivers
  theEvaluationOfArgumentNumber: interestingArgumentIndex
  behavesLike: expectedResults
```

Exercise 2.11 [15] How can `defined` be implemented in more direct terms, without referencing literals or global names?

Exercise 2.12 [24] Find an active voice equivalent to `isDefinedLikeAllOf:`.

Exercise 2.13 [20] Refactor the `answer:` keyword out of the kernel messages without using parentheses or negation.

Exercise 2.14 [10] Examine the selector below.

```
addChannelWith: anInteger
bitsPerSymbolCodingStrategy: aCodingStrategy
andDeltaStrategy: aDeltaStrategy
```

As you can see, it does not read well because, looking at it as if it were a sentence, it needs a comma between `Symbol` and `CodingStrategy`. Change it so that it does not suffer from this defficiency.

Exercise 2.15 [35] We typically use booleans to represent whether a particular feature is turned on or off. Sometimes, we also talk about things being enabled or disabled. But `true` and `false` do not go well with the phrases *turn on* or *enable*. Find a way to easily deal with this issue.

Exercise 2.16 [18] *(Howard Oh)* Simplify the expression below.

```
[:x | x sqrt] value:
  ([:x | x + 5] value:
    ([:x | x * x] value: 2))
```

Exercise 2.17 [14] Further simplification of the expression at the beginning of the chapter is possible. Can you find how?

Exercise 2.18 [12] Simplify the expression below.

```
| index1 index2 index3 index 4 |
index1 := aCollection indexOf: oldObject1.
index2 := aCollection indexOf: oldObject2.
index3 := aCollection indexOf: oldObject3.
index4 := aCollection indexOf: oldObject4.
aCollection at: index1 put: newObject1.
aCollection at: index2 put: newObject2.
aCollection at: index3 put: newObject3.
aCollection at: index4 put: newObject4
```

Exercise 2.19 [20] Using the selector `new` to get instances is confusing when the class always answers a singleton. Using the message `singleton` instead causes

maintenance problems when the class stops answering a singleton (e.g.: when you want multiple parallel processes running on their *own* singleton), and it also sprinkles knowledge of which class is using a singleton all over the place. Find a more neutral selector for these cases.

Note: the selector `observe` does not qualify as an answer.

Exercise 2.20 [30] *(Roger Whitney)* Print out the source code of one of your projects. On this copy circle all instances of the following problems:

- Poor names of variables and methods.
- Long methods.
- Places were your code is not properly indented.
- Places were you have cut and pasted code.

If you do not think you have any such problems, then circle the code that you think I would complain about. Now put the code aside for 6 months.

Six months from now, take out your code and refactor it. This will tell you a lot about your comments, documentation and the quality of the names in your code.

Exercise 2.21 [14] *(Roger Whitney)* It should be clear by now that the careful construction of selectors greatly diminishes the potential value of code comments. Rate the usefulness of the comments in the code snippets shown below from this point of view, and improve the code to eliminate the need for comments when necessary.

"Do not patch poorly written code with comments. Make the code better."
—Roger Whitney

```
BankAccount>>setBalance
  "Set the initial balance of the account to zero"

  balance := 0


SomeClass class>>new
  "Create a new instance"

  ^super new initialize
```

```
SomeCollection>>size
  "Returns the size of the collection"

  "Answer the size of the collection"
  ^size


"Check to see if the string ends with A or E"
((someString endsWith: $A)
  or: [someString endsWith: $E])
    ifTrue: [...]


WebClient class>>server: aServer port: aPort
  "aServer must be the string address of the server,
  and aPort must be the port number"

  "^self instanceInitializedSomehow"


Integer>>factorial
  "Do an iteration from 1 to self and answer the product"

  | answer |
  answer := 1.
  1 to: self do: [:each | answer := answer * each].
  ^answer


Magnitude>>between: lowValue and: highValue
  "Magnitude>>between: lowValue and: highValue
  Answer if the receiver is between lowValue and highValue"

  "^theAnswer"
```

Chapter 3

On `CharacterArray>>match:`

3.1 Preliminaries

This chapter illustrates the valuable benefits of letting the meaning of messages be defined by the context in which they are received.

Relentlessly applying the techniques discussed in the previous chapters can yield impressive results and extremely intention revealing code. Moreover, almost as a side effect, quite dramatic performance improvements are easy to achieve without greatly increasing the cost of change.

The behavior of `match:` provides numerous opportunities for exploring this topic. To get a better grasp on the issues and how to solve them, the following sections will describe the process of creating a reasonably good implementation of `match:` from scratch. This approach will pay off when we examine other existing implementations.

Since we will have several new versions of `match:` around, the first thing we will do is to create an `SUnit` test suite so we can make sure they are working properly.

3.2 `SUnit` tests for `match:`

3.2.1 Behavior recap

The message `match:` provides a convenient way of finding out whether a (rather limited) regular expression pattern matches a particular string. This message is understood by `CharacterArrays`. The classes `String`, `ByteString` and others inherit from `CharacterArray`. Thus, `match:` is implemented in a proper place.

In VisualWorks, two metacharacters are supported in pattern strings. The metacharacter `$*` serves as a multicharacter wildcard which can also match zero characters, while the metacharacter `$#` works as a single character wildcard. An example of their functionality is shown below.

```
'prefix*middle#suffix' match: 'prefixABCmiddleXsuffix' "true"
```

The selector `matchOn:` suffers from the same imperative problem.

The astute reader may have already noticed that `match:` is not a good selector. On one hand, if it is to be interpreted as an order to the receiver, then it is bad because it represents the developer speaking in the first person. On the other hand, if it is meant more as a test message, then the resulting statements are not proper English. There is no way to win here! Indeed, it seems the message should be called `matches:` instead.

This is good opportunity to highlight the importance of choosing proper names once more. How is it possible for a selector that could force developers to use a language incorrectly to have survived so long?

Of course, because in that way there will be a difference in value!

However, not all is lost. If we name our alternate implementation `matches:`, then it will be easy to distinguish it from `match:`.

3.2.2 Feature independent tests

Since there are no test cases available for `match:`, we should start by writing one. Let's consider a sample test method.

For the sake of simplicity, we will prefix the names of our test cases with `String` instead of `CharacterArray`.

```
StringMatchTests>>testEqual

  self assert: ('test' match: 'test')
```

It is easy to come up with several of these test messages. But once we are done writing them, what will happen when we need to test `matches:` instead? None of them will do the job because they reference `match:` directly!

The problem is that when tests specify behavior by making direct reference to the terms of a particular implementation, this dependency prevents the test from exercising different expressions of the same feature. Therefore, if we had to keep several implementations of the same functionality, we would also have to maintain several versions of the same test cases. This would have the effect of multiplying our maintenance homework, to the delight (ahem!) of our customers.

To make things even worse, let's consider the potential situation in which the implementation of a feature and the `SUnit` test case are so dependent on each

other that changes in one cause changes in the other. What kind of protection does running the tests offer now? In some sense, the implementation of the feature becomes a test case for the SUnit test. If anything goes wrong, the test case and the feature could break symmetrically: it might happen that the shortcoming of one is not exercised by the other. In this way, hidden problems can easily creep all the way into production code while we get a false sense of security by looking at the green indicator in the test runner.

But the idea is to design tests so that they clearly fail when something goes wrong. To make proper distinctions between proper and improper behavior, the test case should not be contaminated by artifacts belonging to the feature being tested. If this is not the case, the whole test case could become a tautology.

Because of these reasons, SUnit tests should be designed to be as independent of the actual feature implementation as possible. In particular, having powerful tools such as polymorphism at our disposal, actions like rewriting test methods by means of copy & paste are both unacceptable and inexcusable.

It is actually easy to avoid duplication by separating the behavior specification in SUnit from the actual implementation details. Let's see how.

3.2.3 Equivalent test hierarchies

What do behavior specifications for match: look like? In the previous chapter, helper messages were modeled after assertions made about bits of behavior. In our particular case, the assertions *the pattern matches the string*, and *the pattern does not match the string*, seem quite reasonable. A first attempt to express them is shown below.

```
StringMatchTests>>testEqual

  self assert: (self pattern: 'test' matches: 'test')
```

The method above reads well because we can use matches: instead of match: in our implementation-independent selectors. Note how the specific implementation details are pushed into pattern:matches:. This new message can be refined by subclasses of StringMatchTests, thus allowing the behavior specifications to apply to any implementation we wish to test.

Test case inheritance really pays off here.

However, the method has inlined keyword message sends, which introduces syntax sugar in the form of parentheses. Moreover, self appears as a receiver twice in the same statement. Sometimes, this is a hint of things not having been thought out thoroughly. Using a deficient pattern to write a large number of

methods will amplify all of its expression shortcomings. This is not acceptable because it increases the cost of change.

But we could use the technique of combining consecutive keyword message sends into one to avoid these issues, and thus produce the following improved method.

```
StringMatchTests>>testEqual

  self assertPattern: 'test' matches: 'test'
```

In this way, we prevent any repetitive occurrences of parentheses and `self`. We can easily extend `assertPattern:matches:` to handle negative assertions as well, thus eliminating the need for superfluous occurrences of `not`.

```
StringMatchTests>>testDifferent

  self assertPattern: 'test' doesNotMatch: 'abcdxyztes'
```

The implementation of `assertPattern:matches:` and its negative counterpart requires some care. For example, consider the naïve approach below.

```
StringMatchTests>>
  assertPattern: aPattern
  matches: aString

    self assert: (aPattern match: aString)
```

Note that the test case is being divided into a specification of acceptable behavior, and a specification of interface. In terms of refactoring, the idea is to write the interface specification once and only once.

This would force us to essentially repeat the whole message in each subclass of `StringMatchTests`, when the only part that changes is the message sent to `aPattern`. But we can avoid this unnecessary code redundancy by adding one more layer of indirection.

```
StringMatchTests>>
  assertPattern: aPattern
  matches: aString

    | patternMatches |
    patternMatches := self pattern: aPattern matches: aString.
    self assert: patternMatches
```

```
StringMatchTests>>
  pattern: aPattern
  matches: aString

    ^aPattern match: aString
```

By writing test helper methods in this manner, the one and only message that needs to be refined by subclasses of `StringMatchTests` is `pattern:matches:`. In addition, the message to be refined has just one message send.

Upon first examination, this strategy may look like overkill. However, it has many benefits. Code redundancy is eliminated. Because it is not necessary to copy & paste behavior specifications, the probability of having inconsistent tests due to an inadvertent mistake while duplicating test methods becomes zero. Unless it is otherwise specified by subclasses, any number of different implementations will be consistently tested against a fixed set of behavior specifications.

Mutations happen.

Finally, note how much the upper bound for the cost of change is lowered. Testing a different implementation is achieved simply by subclassing the main test case and refining the rather tiny implementation of just *one message*. It does not get much simpler than that! For example, the behavior of `matches:` can be exercised by means of a test case implementing the single message shown below.

DRY principle, good object orientation, and good programming in general.

DRY: Don't Repeat Yourself

```
StringMatchesTests>>
  pattern: aPattern
  matches: aString

    ^aPattern matches: aString
```

Incidentally, this will create a `SUnit` test that breaks because we have not written `matches:` yet. Now that we have a safety net, it is a good time to switch from behavior specification to feature implementation.

3.3 Some design principles

3.3.1 Minimum required complexity

Let's not jump right into throwing an algorithm together just yet. Instead, let's take a moment to examine our situation. We have a behavior specification, and

we are about to write an implementation that satisfies it. At this point, without any other requirements to abide by, *anything* that makes the `SUnit` test pass is, by definition, *correct*. Let's take advantage of this opportunity to be extremely clear about our intentions.

Consider any solvable problem, and compare some of its solutions. Chances are that some solutions will be more cumbersome than others. However, there must be at least one solution whose complexity reaches a minimum in the sense that a simpler solution is impossible. Thus, the complexity of a problem can be thought of in terms of the complexity of its simplest possible solution[1].

Lorenz & Kidd have an excellent book on this subject: Object-Oriented Software Metrics.

There is a place for measuring simplicity in terms of bytes. However, it is not the most helpful metric when working with others or in long term projects, because achieving this kind of simplicity tends to make changes very expensive.

When we write a piece of code that will be read and maintained by others, *simple* does not necessarily equate to few parts, such as in *bunch of long and heavily inlined methods*, or *least number of classes*. Instead, we would be better off by letting simplicity be inversely proportional to the associated cost of change.

"Part of what I do is a craft, but part of what I do is a science. And I guess the craft comes in knowing what science to use and what science not to use."
—Robert Moog

Thus, if we are to create an *elegant* network of objects that interact together in such a way that their equillibrium point has an interesting emergent property, such as the ability to match regular expressions, we need to begin by drawing enough distinctions to support a carefully designed system of mutual influences whose overall behavior can be easily changed.

In Smalltalk, the point of creating classes is to give rise to different contexts in which messages can be received. These contexts describe a way to see and interpret things, and to react to messages. As we localize and concentrate this knowledge and behavior in classes, instead of spreading it ever so thinly on top of every possible C style function we can think of, we get the benefit of describing it once and only once thus lowering the cost of change.

We cannot experience the world without drawing distinctions. Hence, there is nothing wrong with creating classes, because they represent the manifestation of intention and the observation of a difference in value[2]. Our ability to perform these acts is one of our distinguishing characteristics. In fact, we should ask ourselves how is it possible that so many people do their job while stubbornly

[1] Incidentally, when elegance is inversely proportional to size, it is impossible to prove any given solution is the most elegant one. For a more detailed description of these ideas, see The Limits of Mathematics, by Gregory J. Chaitin. Also, Richard Gabriel has an excellent article on *the quality with no name*.

[2] But creating classes is not the *only* way. Classes are an artifact of how behavior specification is supported and implemented in Smalltalk. For an example of how to describe behavior with prototypes instead, take a look at Self.

fighting against their most unique natural skills.

3.3.2 Programming techniques and psychology

It is natural to think of several contexts receiving the same message, and then interpreting it in different ways according to their particular idiosyncracies. And indeed, the point of polymorphism is to let actions be carried out in terms of the context in which they have to be performed.

This technique, as well as others, has an interesting psychological connection. To obtain the most benefits from polymorphism, the contexts from which the polymorphic message is sent ought to not care how the message is implemented. In other words, good use of polymorphism is a matter of being able to let go of control[3].

This section is about having a point of view that is conducive to getting the most value from polymorphism, as it will become crucial very soon.

Why is it so easy to make a connection between a computer programming technique and a psychological trait? The reason is that Smalltalk does not have features that excessively interfere with getting things done. Instead, it is designed to provide quick feedback to our actions. Because of this, it truly is a very clear mirror in which we can see ourselves. This is why it is often said that Smalltalk code *talks* to you.

It takes considerable practice, true dedication, and being honest with oneself to master the interaction with Smalltalk because it has the tendency to not make concessions. In fact, sometimes it may tell us things we do not like to hear[4]. After all, the manner in which programs are written is, to some extent, a reflection of the psychology of their authors.

But always remember: it is a problem, not a person. If you can treat the Smalltalk environment as a peer with whom you are trying to solve a problem, the results can be astonishing. Smalltalk and the developers will become parts of an *interesting* system of interacting parts. The more the developers can make use of Smalltalk's feedback, the faster their system will evolve. When this is taken to its final consequences, it gives a competitive edge that is virtually impossible to match with mainstream development environments.

With this in mind, let's pay close attention and listen to what Smalltalk has to say while we tackle the implementation of `matches:`.

[3]Wikipedia provides the following background: [i]n psychology, [...] control is the attempt to impose excessive predictability and direction on others or on events, often associated with lack of trust or insecurity.

[4]More often than not, the unwanted criticism comes in the shape of a finger silently pointing to the exaggerated and unwarranted size of our narcissism.

3.4 A first approach to an implementation

3.4.1 Distinctions

Take a moment and examine the pattern shown below.

```
<characters*characters*>
```

Clearly, there seem to be two different kinds of *match characters*: the ones that should match literally, and the star with its well known properties. Since they behave quite differently, let's *draw* distinctions between them.

```
<characters|*|characters|*>
```

This divides the pattern in groups of characters with similar characteristics. We will call these groups *tokens*. In this way, a pattern is a sequenceable collection of tokens.

As you can see, there are two kinds of tokens: the ones in which characters should match exactly, and the star. We should name tokens more specifically. Let's call them *match tokens* and *star tokens*, respectively.

With these terms, it is possible to devise an algorithm to check if a pattern matches a particular string. Given any pattern, there are two possibilities: it has star tokens at both ends, or it does not. If there are match tokens at either end, then we should make sure they match the string. For example, if we were evaluating the expression below,

```
<*|test|*|test> matches: 'XXXtestXYZtest' "true"
```

we should make sure that the string ends with the match token `'test'`. If so, we can eliminate the last match token from both the pattern and the string and check whether the expression below holds.

Distinguish and contract — this is extremely powerful stuff.

```
<*|test|*> matches: 'XXXtestXYZ'
```

Now we are in the case where the pattern has star tokens at both ends. Let's work from left to right. Because of the stars, all we need to do is to find the first occurrence of the match token in the string, eliminate the tokens from both the pattern and the string, and continue. Note that if we can do this, we would also eliminate the leftmost star token. The next step would be to check whether the following expression holds as well.

```
<*> matches: 'XYZ'
```

At this point, it becomes obvious that the expression evaluates to `true`, so we finish.

This explanation is (hopefully[5]) very nice and clear. On average, however, the sample implementations that would typically follow are rather obscure and heavily-commented. Almost invariably, they are written to match the underlying hardware to the maximum possible extent.

This is counterproductive because the original distinctions, such as the match and star tokens in our particular case, are thus hardly ever expressed explicitly. Since most problems cannot be flattened without causing huge change costs, the traditional approach just sets people up for failure. This is unacceptable.

"Premature optimization is the root of all evil (or at least most of it) in programming."
—Donald Knuth

3.4.2 Classes

The distinctions we found are interesting because they represent the fact that we perceive different values according to our intentions. In Smalltalk, distinctions are best modeled by means of classes. Our previous discussion suggests that the class `MatchToken` is in order. In addition, instances of `MatchTokenizer` could distinguish tokens in strings, while instances of `MatchTracker` could coordinate the work of the pieces.

At this point we should consider how we are going to structure our objects. We will address the representation of tokens first. Tokens were previously illustrated as `<*|test|*|test>`. If we want them to look like distinctions, however, we need fully enclosing boundaries. In other words, we are really thinking about tokens in these terms: `<` `*` `test` `*` `test` `>`. If that is the case, we can refer to the tokens in a string by specifying where the token starts and where the token ends. Hence, instances of `MatchToken` will have three instance names: `string`, `start` and `stop`.

Where are the instance names `top` and `bottom`?

At this point, it should be easy to imagine a sample implementation of `MatchTokenizer`: set up indices, stream over the characters in the string, create tokens as stars appear, and answer an ordered collection of tokens when done. This is quite straightforward stuff.

But somehow the idea of a generalized `MatchToken` class does not make a lot of sense. In the process of checking whether a pattern matches a string, we

[5]Since no example showing how much more complicated this can get would fit on a single page, I would rather save the space for something more interesting.

would have to see which token is a star. A self evident way of doing this would be to implement `isStar` in `MatchToken`. And you know what comes after that — occurrences of `ifTrue:ifFalse:`!

However, it is evident that the tokenizer will have to determine where are the star characters before it can create star tokens. Hence, if it has distinguished match tokens and star tokens already, why should other objects have to redo that work?

This is a tell-tale of run time distinctions that should be made at design time. Clearly, representing star tokens as instances of `MatchToken` is a bad idea. Therefore, they will be represented by instances of `StarToken`.

Oh boy, our class count is four now. We must be in so much trouble now!

Since now we will have a class for star tokens, perhaps the arrangement to represent match tokens is not appropriate in this case. Let's examine this for a moment. After the tokenizer runs and separates the patterns in a sequence of tokens, we will have a collection of match and star tokens. But as far as we are concerned, *the exact index location* of star tokens in the pattern does not matter. What is important is where the star tokens occur *in relation to match tokens*. Star tokens are what tell us when we should continue searching for a match token if we do not find it at the first place we look at. Therefore, and somewhat surprisingly, instance names are not needed to represent star tokens!

Better to have too many distinctions than too few...

In some way, star tokens are match token modifiers.

At this point, it should be easy to see that all star tokens will behave in the same way because they have no inner distinctions that can be used to tell one star token apart from another. Therefore, we should write `StarToken>>new` so that it answers a unique instance. Note that this implementation of the singleton pattern should not be confused with an attempt to optimize. In this case, having a star token singleton is the crystal clear and explicit expression of our realization that all star tokens are equivalent.

It may not be an optimization, but a beneficial by-product is that it saves time by avoiding the creation of redundant `StarToken` instances.

3.4.3 Orders

Given the pieces we have seen so far, one could easily conceive a reasonable implementation of `MatchTracker`. For example, it could set up a `matchInterval` to represent the portion of the string that has been verified to match the pattern. Then, it would iterate through the collection of tokens trying to verify the match further, until one of two things happen: either all of the string is verified, or a definite mismatch is found.

To do this, the match tracker would first ensure that the last token is a star token. If the last token is a match token, then it is just a matter of verifying that the string's last characters match those of the last token, and changing the

match interval accordingly. At this point, if there are no more tokens, then an answer can be determined by checking whether the match interval has covered all the string.

If there are more tokens, then the match tracker can assume that the last one is a star token now, because the parser should be written such that it will not generate consecutive match tokens. Then, the match tracker would start verifying tokens from left to right. If the first token is a match token, then its characters must match those of the string starting at the next position to be matched.

So let's say that either the first token is a star token, or that via removal of the leading match token the first token is a star token now. Then, the match tracker should remember that the next token could be found at any position within the portion of the string still unmatched, discard the star token, and continue.

This process should stop if a match token cannot be matched, at which point the answer would be `false`, or if the tracker runs out of tokens, because then the answer would be whether all of the string was matched or not.

Unfortunately, this approach achieves the desired behavior by giving orders instead of relying on the interaction between the parts. All control is done within the realm of the match tracker object, while the tokens are just place holders for particular indices. The boolean logic associated with managing these indices and the token collection itself causes heaps of complexity to appear. In other words, there are too many distinctions being drawn in the same place, and the responsibility associated with drawing those distinctions is concentrated within a single object. As such, this implementation strategy is not good. Hence, we will not go down that path. Instead, let's examine how to render the classic computer science version of this algorithm into an object oriented implementation.

3.4.4 Interactions

To illustrate the point made in the previous section in the best manner possible, we will assume we already have a working, computer science style implementation of the algorithm we have been talking about. We will review it with critical eyes and change it according to what we see. This will give us an opportunity to put what we have already learned into practice, and it will also give us a better feel for its usage.

First, after going through the existing code for a bit, we will sooner or later find the main control switchboard in `MatchTracker`. It could look like this (some checks omitted for simplicity).

Meaning `main{...}`*?*

```
self lastToken isStar
  ifFalse: [self processAndRemoveLastMatchToken].
lastTokenWasStar := false.
self tokens do:
  [:each |
    each isStar
      ifTrue: [lastTokenWasStar := true]
      ifFalse:
        [
          lastTokenWasStar
            ifTrue: [self processFirstTokenAfterStar]
            ifFalse: [self processFirstToken].
          lastTokenWasStar := false
        ]
  ]
```

Here you can see the full impact of giving orders. The code above displays many symptoms that we should learn to recognize quickly.

First, note how painful it is to keep track of the knowledge obtained by running the algorithm, and how much energy must be spent into remembering intermediate results. For example, after each statement is executed, the flow of information is such that any bits of information obtained immediately squeeze out of reach through numerous cracks which then must be forcefully plugged. In this case, the plugs are names such as `lastTokenWasStar`.

Even worse, other than the match tracker itself, none of the objects involved are aware of what is going. This forces the match tracker to take on an omniscient role so everybody else can be told exactly what to do. In fact, it could be that the match tracker is taking that role too seriously — it even tells itself what to do every time it sends those `process...` messages to `self`[6].

Although the classes `MatchToken` and `StarToken` exist, the code still needs to ask whether either of them `isStar`. But tokens already know this because of their own identity. Why does this context need to do any work to figure that out *again and again*?

Certainly, the message `isStar` is what is causing most of the trouble. And in fact, if we delegate the knowledge of what to do to the tokens themselves, then

[6]What is the intention being expressed here? What is the implication about the state of mind of the developer?

`ifTrue:ifFalse:` would become unnecessary. Hence, the idea is to replace the occurrences of conditional logic with polymorphic message sends. Just a bit of practice is needed, and soon enough you will be able to obtain highly valuable results while at the same time investing rather minimal efforts.

With that in mind, let's tackle the first occurrence. Originally, the code looked like this.

```
self lastToken isStar
  ifFalse: [self processAndRemoveLastMatchToken]
```

Whether something needs to be done depends on whether the last token is the star token or not. The knowledge of which token is last belongs to the match tracker. This means that the knowledge of what to do at this point should be known by the token. Obviously, the token will have no existential doubts to address regarding its own identity and will just do what is needed.

Since the action still needs to be carried out in the context of the tracker, we will have the tracker ask the token to tell it what to do. Hence, we rewrite the code above as follows.

```
self lastToken processLastTokenOn: self
```

Now we can let double dispatch take care of all our `ifTrue:ifFalse:` needs.

```
StarToken>>processLastTokenOn: aMatchTracker

  ^self


MatchToken>>processLastTokenOn: aMatchTracker

  aMatchTracker processLastToken


MatchTracker>>processLastToken

  "make sure the last token matches the string, then"
  self dropLastToken
```

Note how the meaning of `processLastTokenOn:` depends on the context of the receiver. Previously, `MatchTracker` would resort to `ifTrue:ifFalse:` to create

Note how quickly the derived information was being purposefully thrown away as well!

a context in which knowledge could be derived. But now, since the appropriate context already exists, *it is no longer necessary to create it to access the meaning it gives to the messages it receives*. In other words, it runs faster[7].

Now that the first `ifTrue:ifFalse:` is gone, let's go after the second one. Here is the original section:

```
lastTokenWasStar := false.
self tokens do:
  [:each |
    each isStar
      ifTrue: [lastTokenWasStar := true]
      ifFalse:
        [
          lastTokenWasStar
            ifTrue: [self processFirstTokenAfterStar]
            ifFalse: [self processFirstToken].
          lastTokenWasStar := false
        ]
  ]
```

In this case, the piece of information that needs to be preserved by the evaluation context is whether the previous token is the star token. Hence, we have to let the current token dictate what to do with the next one. If we allow that to happen, we will be able to eliminate the temporary name as well.

Feel the force, Luke!

After a bit of refactoring, you will see that most of the code just evaporates away and *disappears*. Here is the new section:

```
[self tokens isEmpty] whileFalse: [self processNextToken]
```

Compared to the original version, this has no temporary names, no collection iteration, and no `ifTrue:ifFalse:`!

Do the max with the min.

OK, the drastic reduction in code size makes it look like we cheated somehow. Over time, we all develop a sense for the minimum amount of code necessary to

[7]From time to time, you may hear an argument along the lines of "nothing is faster than sending `==`" or some other heavily optimized message. That may be so, but what typically comes after that is `ifTrue:ifFalse:`. I contend programs can run faster by minimizing the amount of run time distinctions they have to draw. Drawing distinctions at design time means that those distinctions must not be drawn at run time, and thus the run time has less stuff to do. Nothing is faster than *doing nothing*.

implement a given feature. The piece above may even feel like it cannot possibly work because there is not enough code.

Actually, that feeling is letting you know the implementation is getting good.

But we did not cheat. To prove our claim, let's follow the code. First, we will examine the match token branch.

```
MatchTracker>>processNextToken

  self firstToken processFirstTokenOn: self


MatchToken>>processFirstTokenOn: aMatchTracker

  aMatchTracker processFirstToken
```

Note how the `First` and `Last` prefixes in messages like the one next to this graffiti carry the proper meaning through the evaluation context.

This is very straightforward code. The match tracker asks a token to tell it what to do on the assumption that it is the first token. Note how the match tracker does not even care about what kind of token receives this request. In the case of match tokens, the action is just to turn around and tell the match tracker to go ahead and process the token. Let's examine the star token branch now.

```
StarToken>>processFirstTokenOn: aMatchTracker

  aMatchTracker processFirstStar


MatchTracker>>processFirstStar

  self dropFirstToken.
  self areThereMoreTokens
    ifTrue: [self firstToken processFirstTokenAfterStarOn: self]
```

The method above uses double dispatch to handle duplicate star tokens. Note how the method below handles that special case without performing any checks whatsoever. The feature is purely a consequence of how the object interaction is designed.

```
StarToken>>processFirstTokenAfterStarOn: aMatchTracker

  aMatchTracker processFirstStar
```

Finally, here are the corresponding match token methods.

```
MatchToken>>processFirstTokenAfterStarOn: aMatchTracker

    aMatchTracker processFirstTokenAfterStar
```

Highlight the last sentence of this paragraph!

This approach will run faster than the original one because the context in which the star token makes sense does not have to be distinguished every time it is necessary to refer to it. In addition to being faster, the amount of lines of code has been trimmed down as well. That is less stuff for you to maintain.

Aside from the pragmatic results which have value on their own, please pay close attention to what has been done: run time switches in code have been expanded into separate design time paths through which information flows. In other words, if we have a set of distinctions, this is about letting values travel freely from distinction to distinction without forcefully telling the signal what to do.

3.5 A first approach to optimization

The previous section claims that better performance can be achieved by replacing `ifTrue:ifFalse:` with polymorphic message sends, because they avoid the need to draw unnecessary distinctions at run time.

This also means that we can use polymorphism to give us feedback on our design. We can even do it mechanically as a matter of principle, with the idea that it will give us information that we may otherwise miss. If you train yourself to do so regularly, and if you pay attention to what Smalltalk tells you, then a sizable portion of your job will be done almost automatically and thus your productivity will rise. Therefore, let's attempt an all out attack to make `matches:` run as fast as possible — not just because of optimization, but as an exercise to see how much work we can move from run time to design time. First, we will briefly discuss a few basic techniques.

3.5.1 Further elimination of `ifTrue:ifFalse:`

One of the first things that stands out in a `TimeProfiler` run is that a large fraction of the time is being spent sending `Character>>==`. With a bit of work, one can trace it to the match token verification method. Eventually, it turns out that characters are sent the message below numerous times.

```
Character>>matches: aCharacter

  ^self == aCharacter
    or: [self == $#]
```

The method above says that characters match other characters when *either the argument matches on its own right, or the argument does not matter because the receiver is the pound character.* At first, this hotspot seems impossible to improve. But what is going on is that since match tokens can have `$#` in them, they are forced to check for pound characters every single time.

The first thing to do in these situations is to realize that methods like the one above are the sign of unaddressed issues. The issue here is the piece that says `self == $#`. It is written in the context of `self`, which is a character. In addition, the argument to the message `==` is also a character. That means that `$#` cannot take advantage of its own identity because it is unaware of it.

Clearly, there is something funny going on with characters because they are not allowed to know who they are. Why shouldn't they? Do you frequently ask yourself *am I myself?*

More often than not, `self whoAmI` is a consequence of deficient design.

In order to fix this without going about changing the implementation of `Character`, we could create the class `PoundToken`. Because of its behavior, it is most convenient to make it a subclass of `StarToken`. Here are a few interesting methods.

```
PoundToken>>processFirstTokenOn: aMatchTracker

  aMatchTracker processFirstPound


PoundToken>>processLastTokenOn: aMatchTracker

  aMatchTracker processLastPound


PoundToken>>processFirstTokenAfterStarOn: aMatchTracker

  aMatchTracker processFirstPoundAfterStar
```

These changes mean the tokenizer now has more work to do. But since the new token eliminates many message sends outright, any test case will run considerably

faster. The numbers at this stage indicated a 50% speed increase in the matching stage due to adding the new token class.

Feel the awesome object power — so much for nothing being faster than ==!

3.5.2 Aggressive inlining

Even with improvements like the one mentioned above, our current parser and token approach to implement `matches:` is still about 10 times slower than the implementation of `match:` that comes with VisualWorks. This is not necessarily a bad thing, because we are currently trading efficiency to obtain a low cost of change in return. Most of our run time is going into tokenizing the pattern and instance creation, which is expensive compared to sending `Character>>==` and other optimized messages.

But if we were to inline our algorithm into a single method, using the existing implementation of `match:` as a model, how fast would our algorithm run? While this exercise is a bit of an apples to oranges comparison because `match:` has an extra feature we did not implement (the ability to evaluate `aBlock` with the start and stop index for the runs matching a `$#` or a `$*`, unused in the base image), the results can give a good indication of where we stand.

Without comparing the inlined methods explicitly since neither of them would fit in one page anyway, here are some `millisecondsToRun:` results comparing VisualWorks' version of `match:` to our case insensitive inlined implementation. Each test was run one million times using `timesRepeat:`[8]. First, the patterns for which the answer is `true`.

Pattern set A	String to match	VisualWorks	Inlined `matches:`
`abc`	`abc`	632	531
`ab*de#fg`	`abczdexfg`	2,198	1,179
`ab*`	`abczdexfg`	537	529
`ab#`	`abc`	664	540
`*ab*`	`xyzabczdexfg`	1,750	1,803
`*ab#`	`xyzabc`	1,984	661
`*fg`	`abczdexfg`	3,326	513
`#ab`	`xab`	656	532
	Totals:	11,747	6,288

For the particular patterns arbitrarily chosen, our method is faster overall because it checks the end of the pattern first, and because it treats `$#` differently. In

[8]The computer used was a Pentium III running at 600 MHz. Evaluating an empty block one million times took approximately 14 milliseconds.

that way it tends to isolate star tokens in the middle, whereas VisualWorks' implementation goes strictly from left to right. Now, a set of patterns for which the answer is `false`.

Pattern set B	String to match	VisualWorks	Inlined `matches:`
`ab*de#fg`	`abczdexfgz`	4,169	507
`xab*de#fg`	`abczdexfgz`	510	491
`xab*`	`abczdexfg`	510	567
`ab#`	`xabc`	510	540
`*ab*`	`xyzahbczdexfg`	5,214	5,494
`*afb*`	`xyzahbczdexfg`	5,225	5,481
`*ab#`	`xyzabcd`	3,070	622
`*fgx`	`abczdexfg`	3,170	498
`#ab`	`xyab`	669	543
	Totals:	23,047	14,743

Again, our method is faster overall, although in the seemingly most common case in which the pattern is of the form `< * matchToken * >`, VisualWorks' implementation maintains a small advantage. Here is a pattern set with multiple stars for which the expressions evaluate to `true`.

Pattern set C	String to match	VisualWorks	Inlined `matches:`
`ab*de*fgz`	`abczdexfgz`	2,639	1,968
`*ab*de*fgz`	`xabczdexfgz`	3,092	2,595
`ab*de*fg*`	`abczdexfgz`	2,471	2,510
`*ab*de*fg*`	`xyzabczdexfgz`	3,737	3,807
	Totals:	11,939	10,880

And here is a similar set for which the expressions evaluate to `false`.

Pattern set D	String to match	VisualWorks	Inlined `matches:`
`ab*de*fg`	`abczdexfgz`	3,329	497
`ab*de*fg`	`abczdhexfg`	3,837	3,334
`ab*de*fg*`	`xabczdexfgz`	529	575
`*ab*de*fg*`	`xyzabczdhexfgz`	5,462	5,494
`*ab*de*fg`	`abczdexfgz`	3,490	502
`*ab*de*fg`	`axbczdexfg`	4,303	3,850
`*ab*de*fg`	`abczdhexfg`	3,978	3,426
	Totals:	24,928	17,678

The conclusion to draw is that if we are going to resort to inlining, it is easier to make a piece of code run faster by first having an easy to change implementation to play with. Thus, only when the human-friendly version runs as fast as possible should we resort to aggressive inlining.

We should only resort to aggressive inlining if performance is lack-luster to begin with. Otherwise, leave it alone!

Note that changing VisualWorks' implementation to check the end of the pattern first would be quite difficult because the existing version of `match:` does not have an easy to change counterpart. Inlining is a mechanical translation exercise. As such, we should try to avoid working directly on the translated version as much as possible.

3.5.3 Reordering conditional expressions

While working on the inlined implementation of our approach, the time profiler can quickly determine that the expression below is a hot spot.

```
patternCharacter asUppercase ==
  (aString at: lastMatchIndex) asUppercase ifFalse:
    [
      patternCharacter == $#
        ifFalse: [^false]
    ]
```

What we should notice is that looking up the uppercase or lowercase version of a character is relatively expensive. Therefore, the expression of conditional checks should put that particular test after anything that takes less time to do.

```
patternCharacter == $# ifFalse:
  [
    patternCharacter asUppercase ==
      (aString at: lastMatchIndex) asUppercase
        ifFalse: [^false]
  ]
```

Now we can reduce the amount of `asUppercase` lookups by using the fact that, in what appears to be most of the time, case insensitive matching can be case sensitive except for a minority of the characters involved. In other words, if the characters are identical, then it is not necessary to perform the more expensive case insensitive lookup.

```
patternCharacter == (aString at: lastMatchIndex) ifFalse:
  [
    patternCharacter == $# ifFalse:
      [
        patternCharacter asUppercase ==
          (aString at: lastMatchIndex) asUppercase
            ifFalse: [^false]
      ]
  ]
```

Since `$#` is rather infrequent in pattern searches, the check for pound tokens should come after the character equality test. VisualWorks' implementation of `match:` uses these techniques as well.

3.5.4 Credits

Credit must be given when credit is due. These improved versions of `match:` were prepared with the invaluable help of Blaine Buxton. I would like to take the opportunity to thank him for making this chapter possible.

3.6 Exercises

Exercise 3.1 [22] Prove that the matching algorithm implemented as `matches:` in this chapter actually works.

Exercise 3.2 [25] Prove that the left-to-right matching algorithm implemented as `match:` in VisualWorks actually works.

Exercise 3.3 [32] Use the `TimeProfiler` to find a place where adding classes to eliminate `ifTrue:ifFalse:` makes something run considerably faster.

Exercise 3.4 [30] Port the best available versions of `matches:` to VisualAge 6, thus fixing its currently broken implementation of `match:`. In addition, make sure the improvements are included in the next VisualAge release.

Exercise 3.5 [33] Extend the functionality of `matches:` and `inlinedMatches:` by allowing a single digit wildcard. Write the `SUnit` tests before implementing this feature.

Exercise 3.6 [22] Modify how the parsing version of `matches:` is implemented such that it is possible to escape the metacharacters. This would allow `$*` and `$#` to appear in match tokens.

Exercise 3.7 [21] Using the messages `match:` or `matches:` almost certainly causes parentheses because once the result is obtained, chances are that what follows is `ifTrue:ifFalse:`. Use the techniques from the previous chapter to eliminate the need for parentheses.

Exercise 3.8 [15] Trace all sends of `match:` to see how many times the answer is `true` and how many times the answer is `false`. Use the results to weigh the time it took to run each test case mentioned in the benchmarks, thus obtaining a weighed total.

Exercise 3.9 [26] How would you change your `SUnit` tests for `match:` if you suspected that some implementations had issues with uppercase and lowercase characters?

Exercise 3.10 [14] In the parsing implementation of `match:`, the `First` and `Last` prefixes in selectors such as `processFirstTokenOn:` can be a bit misleading because the collection of tokens changes over time. Find better prefixes to avoid potential confusion.

Exercise 3.11 [34] If star tokens are match token modifiers, then research how to rework the parsing implementation of `match:` so that match tokens know if they are preceded by a star or not by virtue of being instances to a class corresponding to that situation (e.g., `MatchTokenPrecededByStar`).

3.7 A second approach to optimization

Now that we discussed traditional performance improvement techniques, it is time to take a step back and try something different.

In general terms, aggressive inlining will not work as well as with `match:` because most of the time the problems we will have to deal with are much larger in scope. Even then, the resulting implementation will become very difficult to maintain and change regardless of whether inlining is mechanical or not. There is only so much mileage to be had with this kind of vehicle. So how do we write fast programs without greatly increasing the cost of change by resorting to aggressive inlining or primitives?

Any implementation of a computer program is a reflection of how we think about a problem. Unfortunately, most of us have been taught over and over, and then heavily trained on, how to translate our thoughts into computer languages that support, for the most part, the 10-zillion-commandment point of view. And yet, our discussions are proof that there is a better way to create programs. In fact, our new point of view says that if a program written in a traditional manner is not efficient, chances are what is going on is that there are way too many orders to be followed.

But in Smalltalk it is especially easy to do something other than just giving orders. We can create our own distinctions at *design time* so that the *run time* process does not have to figure out what we already knew. By means of the careful design of the pieces, we can create a network of interactions that will produce an answer as quickly as possible. It will do this for us because we will design it such that it will not be able to help itself. And as the implementation will not have many occurrences of `ifTrue:ifFalse:`, it may become a very clear reflection of our thoughts.

To do this is an art[9], and as such it deserves *serious* commitment.

Warning! We are about to leave Kansas for good now.

3.7.1 Distinctions and circuits

Each time we call a name and each time we send a message we are crossing boundaries. If we picture a piece of paper with several circles drawn in it, what we are doing is to travel from one distinction to another.

For example, consider the following mathematics problem. It is very popular and it is known by many names, one of which is the Syracuse algorithm problem. The problem states that if k is a positive integer, then recursively evaluating the function f as defined below:

$$f(k) = \begin{cases} k/2 & k \text{ even} \\ 3k+1 & k \text{ odd} \end{cases}$$

will eventually reach a point where f evaluates to 1, regardless of which k is chosen. Perhaps you have been exposed to this before. For example, if we start at 23,

[9] Art: The conscious production or arrangement of elements in a manner that affects the sense of beauty. Human effort to imitate the work of nature. The study and product of these activities.

$$
\begin{array}{lcll}
f(23) & = & 3 \times 23 + 1 & = & 70 \\
f(70) & = & 70/2 & = & 35 \\
f(35) & = & 3 \times 35 + 1 & = & 106 \\
f(106) & = & 106/2 & = & 53 \\
f(53) & = & 3 \times 53 + 1 & = & 160 \\
f^5(160) & = & 160/2/2/2/2/2 & = & 5 \\
f(5) & = & 3 \times 5 + 1 & = & 16 \\
f^4(16) & = & 16/2/2/2/2 & = & 1
\end{array}
$$

This fact has been verified up to extremely large values of k (over 2×10^{17}), however there is no available proof that shows it will hold for all $k > 0$.

One could easily think of a way to implement this in Smalltalk in terms of the recursive process of evaluating f. The method below would be a more or less obvious start:

```
Integer>>syracuse

  ^self even
    ifTrue: [self // 2]
    ifFalse: [3 * self + 1]
```

But that is not the kind of solution we are after. What we need to use is that the function $f(k)$ specifies how each k should behave when asked to evaluate f. In other words, instead of each integer having to *think* about its own identity and what that means in terms of f, it should be possible to let *each* integer know what to do. If we could do this, then the implementation above would not need `ifTrue:ifFalse:`. And in fact, integers would not even need to do any calculations whatsoever. For example, 23 would answer 70 due to the fact that it is 23 who is responding to the message. If it were practical to have one class per integer, you could implement methods like the one below.

```
Integer23>>syracuse

  ^70
```

Let's assume for a moment that we can implement all such messages. Then, the behavior specified by f would be expressed solely in terms of the graph connecting all the integers to one another by means of `syracuse`. This implies a circuit or graph, in which the nodes are integers, and the wires are the implementations of `syracuse`.

Thus, instead of considering the properties of the process of evaluating f, we could consider the properties of the directed graph connecting all the integers as specified by f. In other words, finding a proof would become an issue of finding a property of the graph's topology, rather than one of recursively calculating $f(k)$ for all integers k.

The Syracuse problem is one of showing that a traversal of this graph, starting at any distinction in it, eventually reaches the node labelled 1. This is equivalent to showing that the behavior implied by f has only one accumulation point, namely the one labelled 1.

What is interesting about this is that, given the graph, any traversal can be done without performing any arithmetic operations whatsoever! It is just a matter of behavior and topology.

It so happens to be that this approach is exactly the best way to design highly efficient *and* easy to change programs in Smalltalk (or any other language for that matter).

Amazing execution speed is achieved because the run time does *nothing*. As you will see, in some sense this can be astonishing on its own because it implies that, for the most part, only a rather small amount of expressed intention can be enough to solve most of the problems we typically address with computer programs[10]. And because changes in the interaction between the parts resonate and reverberate through the circuit, the butterfly effect can make minuscule changes result in impressive amounts of change.

Good design is recursive.

3.7.2 Processes

A Smalltalk process represents the act of performing a traversal in the behavior graphs we build by means of creating classes and implementing messages. This traversal is done in discrete steps, each of which corresponds to the act of sending a message.

If we see this in terms of forms and distinctions, traversals are performed in a graph of distinctions connected by worm holes. Again, traversals are done in discrete steps, each of which corresponds to the act of crossing a distinction.

[10] In fact, seeing how puzzlingly little code is necessary to achieve most of what we do daily can cause a shock reaction: it may be embarrasing to find that our complexity meter was way off the mark! Situations like that can devaluate how much we think our own work is worth, and as such are typically avoided. But rather than finding a reason why what we do should be kept in the *complicated* bucket, realizing that it belongs to the *trivial* bucket is a sign of ourselves increasing our understanding. Do not let self assigned importance get between you and your own attractor.

To cross a distinction, the distinction must exist to begin with. Distinctions can only be drawn as a consequence of intention. Since the only things that can cross distinctions are messages, it turns out that these phenomena are reflections of our own intentions.

3.7.3 Messages

It does not make a whole lot of sense to think of a traversal without a traveller. In Smalltalk, travellers move around by means of messages. Let's explore this further.

What is the meaning of requiring that every message has an answer? If we see the process triggered by sending a message as a traversal through a circuit, then sending a message is some sort of evaluation of the circuit. Asking that every message has an answer is equivalent to requiring that every circuit can be evaluated. This, already, has far reaching consequences.

To begin with, a process can be seen as a series of nested circuit evaluations, because each message send represents the local intent to evaluate.

Processes end when they are completely evaluated. Essentially all of the Smalltalk processes that run are spawned in response to the virtual machine receiving a signal. But this is not all that can occur. It is entirely possible for these externally triggered processes to spawn internal processes that, unlike them, never produce a value. Here is an example.

```
[[false] whileTrue: [nil]] fork
```

And it is possible to spawn internal processes that feed on themselves as well, like the one that follows.

```
forkRefinement

  [self refine: self initialContext] fork


refine: aContext

  aContext refineWith: self.
  self refine: aContext
```

In this light, the message `fork` gains a whole new meaning: it can start the

evaluation of circuits which are designed to not have a value[11].

Another area that needs attention is that of message arguments. What do they mean as far as traversals go? Let's start with the case in which messages carry no arguments, such as in `syracuse`. Clearly, performing a traversal on the graph for `syracuse` is one of verifying that there is a single accumulation point to which all traversals arrive. In some way, it is a connectedness check. But if that is the case, one could quickly consider the disjoint sets of all nodes in the graph which are connected to each other. Solving the Syracuse problem is equivalent to showing there is only one such set, and that it contains the node labelled 1. Or, in other words, it is equivalent to showing that if one were to apply the intention of distinguishing the classes of equivalence according to the relationship "has the same accumulation point as", over the set of all possible traversals, one would end up with only one equivalent class in the quotient set implied by this relationship[12].

This, as well, can be seen as drawing distinctions. Remember that distinctions represent the perception of differences in value. In this case, the value is the actual accumulation point for a traversal starting at each node of the graph. Here is another instance where it is useful to think of the label and the labelled distinction as separate objects. The label is *the accumulation point for the traversal starting at* k, and hopefully it is the case that the actual distinction being labelled is always the one we also label as 1, regardless of the value of k.

[11] Consider that electrical signals are travelling through the circuit that is your brain. What does it imply about us? Do our evaluations finish and provide a final answer in some context of evaluation, or do we unwind because of an unhandled exception? Is our life just the continuous evaluation of an infinite feedback loop? Does it matter? And which of these scenarios is it: that the intentions behind our process belong to another process, that processes like ours are in fact spontaneous, or both? If the preceding questions are not the right ones to ask, then what are the valuable distinctions we should try to draw in this regard?

[12] Quick classes of equivalence recap. A *relation* R defined over a set X is a set of ordered pairs of elements of X, e.g.: (x, y). The relation R is called *reflexive* when, for all elements $x, y \in X$, the pair (x, x) is in R. The relation R is called *symmetric* when, for every pair (x, y) in R, the pair (y, x) is also in R. Finally, the relation R is called *transitive* when, for every set of pairs of the form $(x, y), (y, z)$ in R, the pair (x, z) is also in R. When a relation R is reflexive, symmetric and transitive, it is called an *equivalence relation*.

Consider the equivalence relation defined by means of "starts with the same letter as", over the set of all alphabetic strings. Clearly, this relationship sorts all possible strings into the buckets conveniently labelled `$a` to `$z`. No string from one bucket can be in a different bucket at the same time, because otherwise the strings from both buckets would belong to a single bucket thanks to the properties of the relationship. Each bucket is called a *class of equivalence*. The set of all classes of equivalence implied by the relation R over the set X is called the *quotient set* of X divided by R.

Looking at this further from the point of view of `syracuse`, a traversal is the set of all values obtained by repeatedly evaluating f on its own output, starting at k. This can be written as $f^\infty(k)$ for convenience. The fact that the message `syracuse` does not carry arguments can be seen in terms of f not needing more arguments than its own output (or the traversal starting point) to produce the next value. In other words, f has no *static* context across evaluations.

Note that for our purposes, $f^\infty(k)$ will be evaluated in discrete steps, not instantaneously.

Of course, not all functions work without static contexts. Let's review the case of a traversal that requires remembering stuff as it travels through the evaluation space. A good example is the process of garbage collection.

Generally speaking, in order to perform a full GC across the entire object memory, the virtual machine traverses (that verb again!) all reachable objects starting at some root, typically the global dictionary `Smalltalk`. As it visits all the reachable objects, it remembers which objects were visited. This allows the virtual machine to determine which objects were not seen during the traversal, and it is how the VM's garbage collector determines what is garbage and what is not.

But this means that the virtual machine is sorting all objects in the image into two classes of equivalence by means of a traversal. In other words, it is performing a connectedness check to find out which equivalence class each object belongs to! Once the VM determines the contents of the class of equivalence corresponding to the objects that are not garbage, it can proceed to dispose of the class of equivalence corresponding to the objects that are garbage.

The static context of this traversal are the contents of each equivalence class determined so far. Let's look at this more formally. We could name the set of objects to keep as K. Initially, K is empty. Then, the virtual machine's actions can be seen in terms of evaluating a traversal function that we could represent as $t(K, R)$. This function adds R to K, and then evaluates itself for all objects reachable from R that are not already in K. When the traversal is done, K contains all objects reachable from R.

In short, if the virtual machine evaluates the function t with the argument R set to `Smalltalk`, then K will contain all the objects reachable from `Smalltalk` when the traversal is done.

Identical as in ==.

Here you can see that, although the value of R changes, the context containing the set K remains identical across all evaluations of $t(K, R)$ that occur as a result of evaluating $t(K,$ `Smalltalk`$)$.

If you look at this in terms of drawing a distinction, you can see that it results in a perfect partitioning of the whole image into exactly two. This allows to keep track of only one side of the distinction, because from the inside we can figure

out the outside and viceversa. And this is not just weird VM stuff. We express this kind of intentions in Smalltalk much more often than it seems. For instance,

```
ByteString allInstances
  inject: Set new
  into:
    [:staticContext :traversalPosition |
      staticContext
        add: traversalPosition;
        yourself
    ]
```

Now we can see what this static context is more easily[13]. It is more than just the arguments to a message. It is *the space of immediately available names*. In Smalltalk, this includes `self`, `super`, instance names, and so on.

What does the static context represent? I contend that it is *the representation of the position of a signal travelling towards its attractor in an information space, the topology of which is defined in terms of behavior as specified by messages and their implementations*.

This is why it is so fundamentally, essentially and critically important to distinguish contexts, not just objects, with the utmost care and consideration. Each behavior implies a topological structure in the information space. Together with that structure and travel come attractors, the accumulation points towards which traversing signals converge when left to wander through the information space. Here are some examples to consider.

- Division by repeated subtraction is a particularly inefficient way to traverse the set of integers towards quotient and remainder attractors.
- The Syracuse algorithm *seems* to imply only one attractor in the set of positive integers. In addition, all the space *appears* to be connected to this single attractor by means of the behavior of the algorithm.
- All sufficiently large accumulations of closely packed mass will rearrange themselves into a spherical shape, possibly flattened by angular momentum up to the point of disintegration.

[13]It is a good habit to read selectors while making full use of our critical thinking skills. As you can see, the message `inject:into:` could be renamed to something along the lines of `traverseWithStaticContext:atEachStopDo:`.

Typically, the answers we look for are related to the attractors for particular locations in the information space. Therefore, the contexts we distinguish must be designed such that they enable signals to efficiently traverse information spaces towards their attractors, both as a matter of time and as a matter of clarity of expression.

Messages represent the intention to switch contexts, to move to a new location in the information space, to change the point of view from which distinctions are drawn. As messages cross boundaries, they carry with them the manifestation of intention. This is why, in object oriented programming, the most important concept is *messaging* instead of *objects*.

"The big idea is messaging *— that is what the kernal of Smalltalk/Squeak is all about (and it's something that was never quite completed in our Xerox PARC phase). The Japanese have a small word — ma — for* that which is in between *— perhaps the nearest English equivalent is* interstitial. *The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be."*
—Alan Kay

To summarize, under the conditions above, if we let a signal travel in an information space structured by behavior, there are only so many things that are interesting to distinguish, namely:

- What are the accumulation points of the space (or a subset of it) when traversed according to the behavior? In other words, how many different values are distinguished by "accumulation points implied by the behavior in the space"?
- Without knowing the accumulation points of the space, are two locations connected by a traversal implied by the behavior? In other words, are their distinguished accumulation points identical?

This covers most (if not all) of what software does. Hence we must become experts in expressing our intentions clearly, both in terms of the language of the problem and in terms of the information space being traversed.

The previous discussion strongly relies on assuming that distinctions are the results of intention, and that we can only draw distinctions when we perceive a difference in value. In more concrete terms, object oriented programming is hardly more expressive than its predecessors when proper messaging technique is not utilized, because message sends are explicit manifestations of drawing distinctions and perceiving differences in value.

3.7.4 Traversals

Now that we know the basics of messaging, such as naming selectors properly and using polymorphism to cross distinctions drawn by means of classes, it is time to use what we have just discussed to implement `matches:` in a completely new way.

First of all, if we are going to let a behavior traverse an information space to answer the question `aPattern matches: aString`, what is the space we are talking about? And given the space, what does asking `matches:` represent?

We can try to guess the space by looking at the static context of `matches:`. In our non-inlined version, the static context was represented by the instance names of the match tracker. These names kept the positions up to which the pattern had matched the string, and the positions up to which the string had been examined according to the pattern. In other words, the static context of `matches:` seemed to be as follows.

- `aPattern`, `patternMatchedInterval`
- `aString`, `stringMatchedInterval`

If the static context above represents the location in the information space, then what is the direction of the traversal implied by `matches:`? Well, it seems to be going from a location (or static context) of non-empty intervals towards one in which the intervals are empty. Thus, if that is the case, the matter of `matches:` appears to imply two issues that need to be addressed. The first one is *do both intervals become empty at the same time?* The answer to this question is the value of `matches:` — assuming the traversal can be performed in the first place! Clearly, the traversal `'abc' matches: 'xyz'` will not get anywhere. Therefore, the second issue is *can the traversal reach a point in which at least one of the intervals is empty?*

We can put those two questions together. Thus, in simpler terms, the value of `matches:` is the answer to the assertion *the traversal reaches a point at which both intervals become empty simultaneously.* Hopefully these points will be the attractors of the information space as implied by `matches:`.

But how come the space looks like a collection of interval pairs and there is no mention of the pattern and the string? Now it seems they do not belong to the space at all. How is this possible?

The behavior of `matches:` is configured by the pattern and the string.

If we examine this situation carefully, we will see that at each point of the space of interval pairs, `matches:` makes a decision about where to go next *based on the individual characters of the pattern and the string, and on the current position in the space.* Oh... so the pattern and the string specify a particular variety of traversal behavior. In other words, `matches:` translates the pattern and the string into a direction vector at each step of the trip!

Or the direction in which mice traverse a maze.

This suggests that in reality the static context is just the pair of intervals, that the space is one of interval pairs connected in the orientation specified

by `aPattern matches: aString`, that we distinguish interesting accumulation points by drawing distinctions regarding the interval sizes, that we are able do so because we can perceive a difference between their possible values, and that the answer we are after is whether the particular traversal ends at an interesting accumulation point or not.

You can also see `matches:` as discrete vector integration of a trajectory in the space of interval pairs.

3.7.5 Contexts

In order to implement `matches:` following the guidelines implied above, we need to distinguish contexts, to carve spaces of immediate name accessibility, such that travelling through the information space is both efficient and intention revealing. How are we going to do that?

The most important idea is that *a context should exist when it is necessary for processes to make a decision*, typically one regarding the next direction to travel towards to in the information space[14]. So, in the case of `matches:`, what are the decisions that need to be made by run time processes? Here are a few of them.

- Whether a pattern character matches a string character...
  - because they are identical,
  - because they have the same uppercase representation, or
  - because the pattern character is `$#`.
- Whether a pattern character is a star.
- Whether the previous pattern character seen was a star.

This seems to imply a lot of `ifTrue:ifFalse:` and an inlined implementation. But that is not necessarily the case. These decisions should be made at design time so that the run time does not have to make them. Once these decisions are made and taken to their final consequences before a single message is sent, contexts need to be designed to just let those predetermined actions take effect efficiently.

But how can we possibly specify a predetermined action based on whether a character is a star or not? We do not know the particular characters in the

[14]This is why creating classes at design time results in faster run times. The contexts needed to make developer decisions are not necessary and thus signals do not need to go through them. Again, nothing is faster than *doing nothing*.

pattern or the string beforehand! Characters do not know who they are, so we are forced to use `Character>>==` and `ifTrue:ifFalse:`, right?

Well, not really. The previous paragraph is actually giving the answer away. The issue is that characters do not know their own identity. So instead of doing their homework ourselves, we need to teach them to do it on their own. In fact, if you pay close attention to them, each of those `ifTrue:ifFalse:` is begging us to create a class so the decision can be made at design time.

This implies what at first glance could seem borderline heretic. It may feel strange if you are not used to it, but this is how it is done in Smalltalk: we need to create multiple character classes.

This is not `'Kansas'` anymore.

Now we know what we have to do. Let's go for it.

3.7.6 Class based characters

Almost immediately we find out that, in VisualWorks, strings typically store characters encoded in a certain format. This means that we will not be able to put instances of our own character classes inside a string. Fortunately this is Smalltalk, so we can simply create another subclass of `CharacterArray`. We will name our new kind of string as `ClassBasedCharacterArray`. Since it is a subclass of `CharacterArray`, it will inherit a lot of useful collection stuff from its superclasses, and thus we will not have to reimplement all of that ourselves.

So, how will we implement our class based characters? And how many classes do we need? It seems we could do well with one class for `$*`, one for `$#`, and one for the rest of the characters. In this way, each of the contexts implied by these classes will support our needs as far as `matches:` is concerned, without needing to resort to `ifTrue:ifFalse:` because of character identity issues. We will refer to this strategy as *condensed class based characters*.

On the other hand, since we are creating multiple character classes, we could take advantage of this to let each individual character know their uppercase and lowercase counterparts directly by means of instance names. In this way, we would get rid of the comparatively expensive table lookups. And there is more. With this approach, we could eliminate essentially *all* occurrences of `ifTrue:ifFalse:` in `Character` as well! Think of those lovely `whoAmI` messages you know so well: `isDigit`, `isSeparator`, `isAlphabetic`... the implementation of every one of those messages would become a properly placed boolean return. In addition, the implementation of messages like `digitValue` would boil down to answering an integer or `nil`. This is very tempting stuff, and we will refer to this strategy as *enumerated class based characters*.

And 64k classes for Unicode.

Unfortunately, enumeration also means we will have to create 256 character classes to implement it. Maintenance of 256 classes without tools is not going to be a lot of fun. Moreover, a subtle side effect is that so many classes might overflow VisualWorks' virtual machine caches[15], so performance could be lower than with condensed class based characters.

Each approach has its advantages and disadvantages, so we will explore both. Let's take a look at what they have in common first.

3.7.7 Strategy independent support

Since we will be using class based character arrays, we will need the ability to convert a regular string into one of ours. Because there are two kinds of class based characters, two conversion messages suggest themselves. We could name them `asCondensedCharacterArray` and `asEnumeratedCharacterArray`, implement them in `CharacterArray`, and refine them as appropriate. And thus we run into trouble. Our class hierarchy looks like this:

```
CharacterArray
  ClassBasedCharacterArray
  String
    ...
  Text
```

Let's say we take a regular string and send it `asCondensedCharacterArray`. We would expect this to trigger the conversion work and answer an instance of `ClassBasedCharacterArray`. But if we send the same message to the class based character array, we would not expect more work to be triggered. In fact we may even want to prevent more work from being done. Hence, the message should be refined in `ClassBasedCharacterArray` to just answer `self`.

[15] A message that has only one implementation is said to be monomorphic. When a message has multiple implementations it is called polymorphic. If a message has more than just a few implementations, it is considered to be megamorphic.

At first, virtual machines were given inline caches (ICs) that kept track of the compiled method for a particular class and selector combination. This saves time because when the information in the IC applies to the message that is about to be sent, then no message lookups are necessary. For polymorphic messages, however, ICs would be reset all the time leading to thrasing. Hence, ICs were extended to hold more than one entry, and thus became polymorphic inline caches, or PICs for short. Eliot Miranda, VisualWorks' virtual machine guru for many years, described ICs tendency to thrash as "a catastrophe of PIC proportions". In particular, VisualWorks' polymorphic inline caches hold "only" 8 entries.

However, if the class based character array contains enumerated characters instead, the refined message cannot meet our expectations. We want it to do one of two things depending on the contents of the receiver. This, more likely than not, will result in `ifTrue:ifFalse:`.

But the problem is not that the message is not making a decision on what to do. The real issue is that the class based character array cannot make assumptions about its contents. Therefore, instead of resorting to `ifTrue:ifFalse:` in the conversion message, we need to create more classes.

```
CharacterArray
  ClassBasedCharacterArray
    CondensedClassBasedCharacterArray
    EnumeratedClassBasedCharacterArray
  String
    ...
  Text
```

This allows each class to answer `self` or do conversion work as appropriate, based on the context of the receiver.

In addition, it would also be desirable that evaluating the print string of a class based character array gives us another class based character array. With two classes instead of one, it becomes possible to implement the `printOn:` messages without using `ifTrue:ifFalse:` either, as shown below.

```
ClassBasedCharacterArray>>printOn: aStream

  super printOn: aStream.
  aStream space.
  aStream nextPutAll: self conversionSelector


ClassBasedCharacterArray>>conversionSelector

  ^'abstract'


CondensedClassBasedCharacterArray>>conversionSelector

  ^#asCondensedCharacterArray
```

```
EnumeratedClassBasedCharacterArray>>conversionSelector

    ^#asEnumeratedCharacterArray
```

Note that by letting the conversion selector be defined in terms of `'abstract'` in `ClassBasedCharacterArray`, as opposed to `self subclassResponsibility`, we will avoid printing failures in inspectors should we create an abstract class based character array by mistake. Although the print string would fail to evaluate because nobody implements the message `abstract`, at least we would be able to see what is going on. Allowing messages like `printOn:` to fail is tantamount to leaving irritating homework for others.

Another interesting detail comes from the implementation of the conversion messages themselves. Since we have two of them, and they do almost the same thing, they cause a good opportunity for unwanted code duplication to occur. For example, a simple approach would be along the lines of the one shown below.

```
CharacterArray>>asCondensedCharacterArray

    | answer |
    answer := CondensedClassBasedCharacterArray new: self size.
    1 to: self size do:
      [:each |
        answer
          at: each
          put: (self at: each) asCondensedClassBasedCharacter
      ].
    ^answer
```

Somehow this is not right. The message `asEnumeratedCharacterArray` would be almost the same, including the duplication of the index management code. What is wrong here, what do we really mean when we write a method like the one above? Most likely, we are very familiar with an extremely similar message already.

And indeed, upon closer examination, it seems we want the behavior of `collect:`, with the difference that the result needs to be of a class specified by the sender. But if we can easily and clearly express that in English, then what prevents us from doing the same thing in Smalltalk? Let's boldly give a name to our intentions.

```
CharacterArray>>collect: aBlock into: aCollection

  self doWithIndex:
    [:each :eachIndex |
      aCollection
        at: eachIndex
        put: (aBlock value: each)
    ].
  ^aCollection
```

This message can be implemented higher in the class hierarchy, but for now it serves our purposes in `CharacterArray`. With it, we can reimplement the conversion messages in a much more elegant way.

```
CharacterArray>>asCondensedCharacterArray

  | answer |
  answer := CondensedClassBasedCharacterArray new: self size.
  ^self
    collect: [:each | each asCondensedClassBasedCharacter]
    into: answer
```

The enumerated counterpart to the method above just replaces the message sent to each character and the desired class of the answer. In this way the index management part does not need to be duplicated, and at the same time we gain the advantage of having a new message to express ourselves clearly.

Regarding our characters, it is useful to make them compatible with regular characters. For example, they should be able to respond to `asInteger`. This makes it easier to convert to and from one another. In addition, by implementing messages like `isPound`, we will be able to run our old tokenizer and match tracker with our new characters and character arrays as well[16].

Our new class based characters themselves should be singletons just like the star and pound tokens were. The mechanisms to create, initialize and keep track of them are quite straightforward.

[16]Using parallel test hierarchies, just 29 test cases are run in 12 different contexts. This yields a total of 348 tests run for the whole test case suite. Adding support for a new context is just a matter of creating a subclass and requires no changes to the tests themselves. New tests become effective immediately for all contexts with zero effort.

While most condensed class based characters will need an instance name to hold the answer to `asInteger`, enumerated ones can simply answer an integer instead because they have full knowledge of their identity.

The hierarchy of class based character classes is below, with instance names shown in parentheses.

```
ClassBasedCharacter
  CondensedClassBasedCharacter (asInteger)
    CondensedPoundCharacter
    CondensedStarCharacter
  EnumeratedClassBasedCharacter (asUppercase asLowercase)
    ClassBasedCharacter000
    ...
    ClassBasedCharacter035 "Pound"
    ...
    ClassBasedCharacter042 "Star"
    ...
    ClassBasedCharacter255
```

3.7.8 Condensed implementation strategy

Now that all the supporting pieces are in place, we are in a position to finally approach the actual implementation of `matches:` as a circuit, the evaluation contexts of which reside in class based characters.

The interesting aspect of implementing circuits with multiple inner contexts is that their very structure suggests the order in which each of their pieces should be created. In other words, the point of viewing the circuit as a whole is to evaluate it for some arguments. If we see the circuit's inner parts in terms of a graph, implementing the circuit is equivalent to coloring each node of the graph while traversing it. Therefore, a suitable implementation technique in these situations is to take enough example arguments so that their corresponding evaluations will require the circuit's graph to be completely painted with our implementations.

We play the role of a JIT compiler.

This is what we did when we went over an example of how our bidirectional approach to `match:` would work earlier in this chapter. So again, it is all about some sort of connectedness check in a graph, or traversing an information space either towards its attractors or visiting all possible locations. The Laws of Form are present everywhere.

So, for our concrete purposes, we will concentrate on the expression below.

```
'a#c*fg#ij*x#z' matches: 'abcQQfgfghijQQxyz'
```

Watch! A constraint has been chosen and now the rest of the implementation will literally weave itself out of thin air because it will have no choice other than to do so!

For this to work, we need to implement `matches:` in our new character arrays. Since the first thing that has to be done is to match backwards from the end of the pattern and the string, and since each and every decision needs to be made in the context of the individual characters of the string, we need a way to both start the process to match backwards and also to let characters determine what to do next.

While we do not have a clear picture of how does that look in the context of the characters yet, we can actually implement `matches:` at this point since the intention that needs to be expressed is already known to us: matches should start backwards.

Well, but if one can say *start* matching backwards, eventually one will also say *continue* matching backwards. Therefore it is not necessary to distinguish between starting and continuing, and so we choose the verb *continue* for the selector.

So if one were to say *continue*, one would eventually need to mention *where to continue from*. Clearly, if `matches:` is going backwards, there will be some tracking needed to see how far back it has been able to travel. As the intervals can shrink at each character matched, it follows that the tracking mechanism representation must always be present in the current context. After a bit of thought, this becomes self-evident because this tracking mechanism represents the position in the information space that `matches:` is traversing. It is easy to see that a new position can only be derived from the current position, and thus the current position should be immediately accessible at all times.

Since `matches:` is going backwards, we can let the interval starting points be fixed at the beginning of the pattern and the string, namely 1. This means we do not need to explicitly represent or transmit this information, as we can let the implementation assume those values.

Regarding the end points of the intervals, while at first their values are the sizes of the pattern and the string, their values may be decremented during the traversal. This means the end points' values must be explicitly represented and transmitted, so that they are always available in the context that makes traversal decisions.

Keeping in mind these conditions, we reach the conclusion that the first thing that needs to be said is that `matches:` should continue matching backwards starting at the sizes of the receiver and the argument. In other words,

```
ClassBasedCharacterArray>>matches: aString

  ^self
    startingAt: self size
    continueBackwardsMatches: aString
    from: aString size
```

This implementation is not bad at all considering we still have no clue as to how `matches:` will be ultimately implemented! However, by implementing this message, we have also adopted a constraint. Any further messages implemented must play nicely with this one.

We may not be able to predict the path to the attractor, but since we know where the attractor is the actual traversal does not matter as long as we continue to zero in on our target efficiently and effectively.

As if by mathematical induction, each step we take gives us what we need to take the next one. Although we cannot predict consequences far away from this initial constraint, we now have enough information to implement the message sent above. Spelling out the message shape as soon as possible provides useful argument names with which we can think more easily about how to write the actual method.

```
ClassBasedCharacterArray>>
  startingAt: patternIndex
  continueBackwardsMatches: aString
  from: matchIndex
```

So, if one just said *continue from here*, how could `matches:` continue? Well, first of all, did we reach a point where we can go no further? What happens if, for example, `patternIndex = 0`? In that case, `matches:` would be out of pattern characters. That is one of the conditions to detect an interesting attractor. If that were to happen, then the answer to `matches:` would be `true` if and only if we had run out of characters to match as well. Therefore, we can write the first line of the method:

```
ClassBasedCharacterArray>>
  startingAt: patternIndex
  continueBackwardsMatches: aString
  from: matchIndex

    patternIndex = 0 ifTrue: [^matchIndex = 0].
```

Here we are using the fact that 0 is the first value that is less than the intervals' start points, which are assumed to be 1.

But what if `matchIndex = 0`? At this point, we must have `patternIndex > 0` because otherwise we would have returned from the method in the line above. So, initially, it appears the answer would be `false`, except in the case when all that remains in the pattern are stars. Therefore, we can write the second line of the method as well:

```
ClassBasedCharacterArray>>
  startingAt: patternIndex
  continueBackwardsMatches: aString
  from: matchIndex

    patternIndex = 0 ifTrue: [^matchIndex = 0].
    matchIndex = 0
      ifTrue: [^self hasStarsFrom: 1 to: patternIndex].
```

If `matches:` has not reached an interesting attractor yet, then we are at a point where we know we can move on and take a step in the information space of pairs of intervals. The position in the space is known because we have `patternIndex`, `matchIndex`, and we know from the selector that `matches:` is going backwards. We just need to find out the direction in which to move. For this, we need to ask the characters themselves.

At this point, there are three different cases to consider: that the current pattern character is a star, that the current pattern character is a pound, and that the current pattern character is something other than a star or a pound. Of course, since we have a distinct class per case, we will let polymorphism deal with this. But how do we phrase the question that needs to be asked? We should ask ourselves first what is it that needs to be distinguished.

From a simplistic point of view, we are moving the execution of `ifTrue:ifFalse:` down to the implementation of PICs in the virtual machine.

Most interestingly, upon quick inspection, we will find that every potential direction in which to move is known at this point. Actually, there is a quite limited amount of them. So much so, in fact, that there is just *one* of them: the only thing that can happen is that `patternIndex` and `matchIndex` are decremented, because `matches:` is going from right to left. If this step can be taken, then `matches:` would continue matching backwards with different end points. If it cannot be taken, it is because the context is a star character and matching needs to continue in a left-to-right fashion from the beginning of the pattern and the string. Therefore, we should let the corresponding characters tell the string what

The constraints squeeze the answer, quite literally, from nothing. It is almost a miracle.

to do according their identity. In other words,

```
ClassBasedCharacterArray>>
  startingAt: patternIndex
  continueBackwardsMatches: aString
  from: matchIndex

    patternIndex = 0 ifTrue: [^matchIndex = 0].
    matchIndex = 0
      ifTrue: [^self hasStarsFrom: 1 to: patternIndex].
    ^(self at: patternIndex)
      ifYouMatch: (aString at: matchIndex)
      let: self
      startingAt: patternIndex - 1
      continueBackwardsMatches: aString
      from: matchIndex - 1
```

And that is it. Yes, this rather tiny bit of behavior completes the implementation of the message `startingAt:continueBackwardsMatches:from:`. We can do this because the complexity of the problem is encoded in the information space being traversed. Note how the particular structure of the space lets our implementation decide quite simple things. Given a *unique* traversal direction, should it be taken? Not only that is a rather simple thing to determine — the decision is done by polymorphism so `ifTrue:ifFalse:` does not even enter the picture at all! There is nothing to say because there is nothing to do. This is a mark of proper, artful design.

We could even start suspecting that this implementation might be quite fast. But let's not get ahead of ourselves. We still need to complete the implementation traversal of `matches:` first. Now we need to implement `ifYouMatch:...` for stars, pounds, and the other characters.

And here, it is at this exact point that the full benefit of classes comes to fruition. Because it is due to the existence of the classes we created, and to the fact that the contexts they imply are specific enough, that behavior can proceed *without asking any unnecessary questions*. It just runs like electrons flowing down a copper wire. They do not ask for permission. They do not get into `ifTrue:ifFalse:` existential considerations. They just behave that way because *they cannot help themselves*.

This is how we have designed our implementation. Now we will see our guiding principles in action. As before, let's spell out the shape of the message we are

trying to implement first, so we can refer to things by their names.

```
CondensedClassBasedCharacter>>
  ifYouMatch: aCharacter
  let: aPatternString
  startingAt: aNewPatternIndex
  continueBackwardsMatches: aMatchString
  from: aNewMatchIndex
```

This message will have three implementations: one for "ordinary" characters, implemented in the class shown above, one for pound characters implemented in the class of pound characters, and one for star characters implemented in the class of star characters.

So, if the receiver is a pound character, how should this be implemented? Clearly, no checking needs to be done because the pound character matches everything. In this case, the unique traversal step can always be taken. Therefore, we simply let the intervals shrink and pass the baton to the string again. In other words,

```
CondensedPoundCharacter>>
  ifYouMatch: aCharacter
  let: aPatternString
  startingAt: aNewPatternIndex
  continueBackwardsMatches: aMatchString
  from: aNewMatchIndex

    ^aPatternString
      startingAt: aNewPatternIndex
      continueBackwardsMatches: aMatchString
      from: aNewMatchIndex
```

This implementation does pretty much nothing. And here, nothing is meant quite literally. In particular, note that *there is not even a single send of* `==`! Also, pay attention to how the existence of a context representing the pound character makes it possible to let things happen in a *mandatory* fashion, without it being necessary to forcefully tell objects what to do nor to draw any distinctions whatsoever.

But what if the receiver was an "ordinary" character? In that case, the context may determine that the step should not be taken. This would occur only when

the receiver and the character passed as an argument do not match. Therefore, a simple refinement to the implementation above would let the traversal carry on without difficulty.

```
CondensedClassBasedCharacter>>
  ifYouMatch: aCharacter
  let: aPatternString
  startingAt: aNewPatternIndex
  continueBackwardsMatches: aMatchString
  from: aNewMatchIndex

    ^(self matches: aCharacter) and:
      [
        aPatternString
          startingAt: aNewPatternIndex
          continueBackwardsMatches: aMatchString
          from: aNewMatchIndex
      ]
```

Again, not a single occurrence of `ifTrue:ifFalse:`. Now, it could argued that eventually there will be sends of `==`, happening somewhere in the implementation of `matches:` for characters. However, they will only occur because the value of the comparison cannot be determined at design time. In other words, those `==` sends will draw distinctions the value of which can only be known at run time.

Finally, how should this message be implemented for star characters? Clearly, the pattern should be told to continue matching forward, but what about the interval end points? By the time this message is evaluated, the indices will have been decremented on the assumption that backwards matching will continue. To address this now invalid assumption, we can simply increment the indices by 1 before being passed back to the pattern. This needs to happen because when reverting directions due to a star, our bidirectional approach to `matches:` will expect star characters to serve as end-of-pattern terminators. In addition, it would be improper at this point to assume that the star matches at least one of the characters in the string. If we did not reincrement the end point of the string interval, then the example below would fail.

```
'ab*c' matches: 'abc'
```

Therefore, no more characters should be matched from the string. At this point,

the star character should tell the string to start matching forward — unless `aNewPatternIndex = 0`, in which case we detect another interesting attractor and are thus done. In other words,

```
CondensedStarCharacter>>
  ifYouMatch: aCharacter
  let: aPatternString
  startingAt: aNewPatternIndex
  continueBackwardsMatches: aMatchString
  from: aNewMatchIndex

    ^aNewPatternIndex = 0 or:
      [
        aPatternString
          startingAt: 1
          upTo: aNewPatternIndex + 1
          continueForwardMatches: aMatchString
          from: 1
          upTo: aNewMatchIndex + 1
      ]
```

This completes the implementation of backwards `matches:`.

Let's take a quick break and go over our example to see how what we wrote so far would work.

```
'a#c*fg#ij*x#z' matches: 'abcQQfgfghijQQxyz' "true"
```

The pattern would receive `matches:`. As per the implementation of `matches:`, the pattern would begin matching backwards. The first step of this process is to ask the pattern's `$z` to continue backwards matching if it matches the string's `$z`. The character receiving this request is an ordinary character because it represents `$z`. Because of how ordinary characters are designed to fulfil the request from the pattern, the pattern's `$z` checks whether it matches the string's `$z` or not. Since it does, it tells the pattern to continue backwards matching within the constrained range provided by the pattern in the first place. Thus, our example has been reduced by one character, as shown below.

```
'a#c*fg#ij*x#' matches: 'abcQQfgfghijQQxy'
```

The pattern now continues walking from right to left, starting at the decremented rightmost position. Thus, the next character in the pattern is `$#`, and it is asked to continue backwards matching if it matches the string's next character to match, `$y`. The receiver is the pound character, and because of how it implements the message sent from the pattern, it does not perform any checking whatsoever. Instead, it immediately turns around and tells the pattern continue backwards matching with the already decremented indices. This can be done because it relies on assumptions that can be made *in the context of the pound character*. As a result, our example is reduced by another character.

```
'a#c*fg#ij*x' matches: 'abcQQfgfghijQQx'
```

The pattern continues matching backwards as instructed by the pound character. The next character in the pattern is `$x`, and as such it is told to let the pattern continue matching backwards if it matches the string's last character, `$x`. But the pattern's `$x` matches the string's `$x`. Therefore, it tells the pattern to continue matching backwards using the indices decremented by the pattern in the first place. Thus, one more character is dropped from both the pattern and the string.

```
'a#c*fg#ij*' matches: 'abcQQfgfghijQQ'
```

At this point, the last character in the pattern is `$*`. Therefore, it is asked to let the pattern continue backwards matching if it matches the string's last character, `$Q`. The receiver is a star character, and provides its own implementation of this request: it increments the interval end points by 1, and then tells the pattern to start matching forward instead.

Here is where the change in direction occurs. As we just saw, star characters can easily take care of this — and there are no occurrences of `ifTrue:ifFalse:` to be seen! Magic. By means of polymorphism alone, the matching direction has been switched from backwards to forward. This can be done because, as all potential cases are modeled by classes, it is possible to just do what needs to be done for every particular situation without needing to do any thinking whatsoever — in other words, *it is possible to take action without having to rediscover what the context is, because that knowledge is implied by the very context in which action takes place to begin with.*

Now that matching backwards is taken care of, we need to implement how patterns will match forward. In order to continue painting the implementation of `matches:`, then, the first thing we need to do is to provide a method for the message below.

```
ClassBasedCharacterArray>>
  startingAt: patternIndex
  upTo: maxPatternIndex
  continueForwardMatches: aMatchString
  from: matchIndex
  upTo: maxMatchIndex
```

The first we need to do is to distinguish whether the traversal has arrived at an interesting attractor or not. Fortunately, we can easily adapt the checks we performed when the traversal was going backwards. In a similar manner, we can reuse our strategy for letting the characters make a decision. Therefore we write:

```
ClassBasedCharacterArray>>
  startingAt: patternIndex
  upTo: maxPatternIndex
  continueForwardMatches: aMatchString
  from: matchIndex
  upTo: maxMatchIndex

    patternIndex > maxPatternIndex ifTrue: [^true].
    matchIndex > maxMatchIndex ifTrue:
      [^self hasStarsFrom: patternIndex to: maxPatternIndex].
    ^(self at: patternIndex)
      ifYouMatch: (aString at: matchIndex)
      let: self
      startingAt: patternIndex + 1
      upTo: maxPatternIndex
      continueForwardMatches: aString
      from: matchIndex + 1
      upTo: maxMatchIndex
```

Hmmm... this is getting on the verge of being too hard to understand! Let's keep an eye on the amount of names in the current context to make sure it does not grow any further.

The context is being deliberately flattened in a calculated trade off for speed.

As in the backwards case, we have three possible receivers for the forward version of the message `ifYou:...`: ordinary characters, pounds and stars. The first two are so similar to the backwards ones that their implementation is not worth reviewing. The necessary work for star receivers, however, is a completely

different story. Let's fast forward to the first star receiver in our example.

```
'*fg#ij*' matches: 'QQfgfghijQQ'
```

What should the star receiver do? It could start scanning forward, but that would essentially follow the steps of our previous inlined versions. Then the question becomes how could the star know the proper answer to the question of whether the rest of the pattern matches the string?

It cannot know the answer. And since it cannot, because the information to answer that question is not immediately available, it will have to delegate the work back to the pattern — being careful enough to let the pattern know that a star character has just been seen! In other words,

```
CondensedStarCharacter>>
  ifYouMatch: aCharacter
  let: aPatternString
  startingAt: aNewPatternIndex
  upTo: maxPatternIndex
  continueForwardMatches: aMatchString
  from: aNewMatchIndex
  upTo: maxMatchIndex

    ^aPatternString
      startingAt: aNewPatternIndex
      upTo: maxPatternIndex
      continueStarForwardMatches: aMatchString
      from: aNewMatchIndex - 1
      upTo: maxMatchIndex
```

Note that, while the string's interval start index is backspaced once, the pattern's interval start index is left as it is. In other words, the traversal goes over the star and changes matching behavior, but does not consider characters in the string just yet.

It may feel like we have been thrown back into the context of the pattern. In fact, we may even fear that our approach cannot possibly work. But it is very important to remain calm, and continue our traversal as if all those medieval looking circumstances did not exist, even if we cannot see an exit yet. Again, we should give names to the message and its arguments so we can think of them in terms of names.

```
ClassBasedCharacterArray>>
  startingAt: patternIndex
  upTo: maxPatternIndex
  continueStarForwardMatches: aString
  from: matchIndex
  upTo: maxMatchIndex
```

First things first: have we reached an interesting attractor? How can we tell? Let's say we ran out of pattern characters. That means the last one was a star, and in turn it means that the last star matches anything left in the string. This is true even if there are no more characters left in the string, as the star character matches zero characters too. That gives us one of our interesting attractor checks. The other interesting attractor check is to see whether we have run out of string characters. If that is the case, we can just make sure anything remaining in the pattern are stars, like we did before. Finally, we should let characters decide as usual, taking care of letting them know that a star has just been seen in the pattern. Therefore,

```
ClassBasedCharacterArray>>
  startingAt: patternIndex
  upTo: maxPatternIndex
  continueStarForwardMatches: aString
  from: matchIndex
  upTo: maxMatchIndex

    patternIndex > maxPatternIndex ifTrue: [^true].
    matchIndex > maxMatchIndex ifTrue:
      [^self hasStarsFrom: patternIndex to: maxPatternIndex].
    ^(self at: patternIndex)
      ifYouMatch: (aString at: matchIndex)
      afterStarLet: self
      startingAt: patternIndex + 1
      upTo: maxPatternIndex
      continueForwardMatches: aString
      from: matchIndex + 1
      upTo: maxMatchIndex
```

By now, panic might be rampant. We have done nothing other than message

passing, we are not getting any closer to solve the problem, we need to inline everything away!... no, not really. All that dark are paraphernalia is actually not there. Look around. Do you see any dragons, tigers, or dirty castles in the dark? We have to insist and to keep pushing because we have not reached a dead end yet. We have to let the traversal take us where it wants us to go. We are not trapped in a corner. We must keep painting.

So, once more, we have three possible receivers. First, let's consider the star case in terms of our example:

```
'*fg#ij*' matches: 'QQfgfghijQQ'
```

If we reach another star after having seen a star, then clearly all that is between the current star and the previous one matched. So we just let the pattern know that another star has occurred, and delegate the job again — unless we have run out of pattern characters, in which case we would be done. In other words,

```
CondensedStarCharacter>>
  ifYouMatch: aCharacter
  afterStarLet: aPatternString
  startingAt: aNewPatternIndex
  upTo: maxPatternIndex
  continueForwardMatches: aMatchString
  from: aNewMatchIndex
  upTo: maxMatchIndex

    ^aNewPatternIndex > maxPatternIndex or:
      [
        aPatternString
          startingAt: aNewPatternIndex
          upTo: maxPatternIndex
          continueStarForwardMatches: aMatchString
          from: aNewMatchIndex - 1
          upTo: maxMatchIndex
      ]
```

See? That was not that hard, really. In fact, our implementation handles multiple consecutive stars for free! Maybe this is not that scary after all.

Let's consider the case of pound receivers now. We are in a state where we know we have seen a star, and now we see a pound. Well, if there are enough

characters, how could we possibly conclude the answer is `false`? Because clearly, the pound character will match anything, and this particular context cannot see any information that could be used to determine the pattern does not match the string! There is nothing to distinguish because we cannot perceive a difference in value at this point. Therefore, we decide to delegate the problem back to the pattern. In other words,

```
CondensedPoundCharacter>>
  ifYouMatch: aCharacter
  afterStarLet: aPatternString
  startingAt: aNewPatternIndex
  upTo: maxPatternIndex
  continueForwardMatches: aMatchString
  from: aNewMatchIndex
  upTo: maxMatchIndex

    ^aPatternString
      startingAt: aNewPatternIndex
      upTo: maxPatternIndex
      continueStarForwardMatches: aMatchString
      from: aNewMatchIndex
      upTo: maxMatchIndex
```

Things look decent so far. But you may have noticed that these objects are being extremely lazy. Where is the piece that actually *does something*?

Good object oriented design is lazy. It is called late binding of making decisions.

Well, here it is. Let's examine ordinary character receivers now. We are in the context of having just seen a star, and now we have two possible outcomes: either the receiver matches `aCharacter`, or it does not. If it does not, we cannot conclude anything because it may be that we need to go over a few characters to find a match. So what we need to do is try to skip one character in the string and try again by delegating the job back to the pattern. This is perfectly fine because we have *just* seen a star. The receiver knows this because it has received *this* message, not *any other* message.

But what if the receiver matches `aCharacter`? If that is the case, then we cannot be lenient and skip characters anymore. But this seems to be the only thing we can determine in this context. As with the star character before, there is no access to more characters so that the answer can be determined, and hence we will just delegate the work back to the pattern indicating that, after a star,

the next ordinary characters have matched. Matches ought to be literal up to the next star. Therefore, we express these intentions as follows.

```
CondensedClassBasedCharacter>>
  ifYouMatch: aCharacter
  afterStarLet: aPatternString
  startingAt: aNewPatternIndex
  upTo: maxPatternIndex
  continueForwardMatches: aMatchString
  from: aNewMatchIndex
  upTo: maxMatchIndex

    (self matches: aCharacter) ifFalse:
      [
        ^aPatternString
          startingAt: aNewPatternIndex - 1
          upTo: maxPatternIndex
          continueStarForwardMatches: aMatchString
          from: aNewMatchIndex
          upTo: maxMatchIndex
      ].
    ^aPatternString
      startingAt: aNewPatternIndex
      upTo: maxPatternIndex
      continueLiteralForwardMatches: aMatchString
      from: aNewMatchIndex
      upTo: maxMatchIndex
```

Back in the context of the pattern, now we have to determine what to do when we have to enforce literal matching. At least in principle, we already know how to do that because we already specified how to continue forward matching. But what if forward matching fails? It means that we need to skip more characters and try to find a literal match further down. We can see that in our example:

```
'*fg#ij*' matches: 'QQfgfghijQQ'
```

Not only will `matches:` have to skip through the first two occurrences of `$Q` — in addition, it will have to recover a false literal match start after the first occurrence of the substring `'fg'`.

Let's provide a method for this message. As usual in the context of the class based character array, we start our implementation with the usual attractor checks, and then simply write down what we discussed in the previous paragraph: if pursuing a literal match fails, then skip one character in the string and try again.

```
ClassBasedCharacterArray>>
  startingAt: patternIndex
  upTo: maxPatternIndex
  continueLiteralForwardMatches: aString
  from: matchIndex
  upTo: maxMatchIndex

    patternIndex > maxPatternIndex ifTrue:
      [^matchIndex > maxMatchIndex].
    matchIndex > maxMatchIndex ifTrue:
      [^self hasStarsFrom: patternIndex to: maxPatternIndex].
    ^((self at: patternIndex)
      ifYouMatch: (aString at: matchIndex)
      afterLiteralLet: self
      startingAt: patternIndex + 1
      upTo: maxPatternIndex
      continueForwardMatches: aString
      from: matchIndex + 1
      upTo: maxMatchIndex) or:
        [
          (self at: patternIndex - 1)
            ifYouMatch: (aString at: matchIndex)
            afterStarLet: self
            startingAt: patternIndex - 1
            upTo: maxPatternIndex
            continueForwardMatches: aString
            from: matchIndex
            upTo: maxMatchIndex
        ]
```

Finally, all we need to do is teach characters how to continue literal forward matching in terms of forward matching. And thus, the implementations of the

message

```
ifYouMatch: aCharacter
afterLiteralLet: aPatternString
startingAt: aNewPatternIndex
upTo: maxPatternIndex
continueForwardMatches: aMatchString
from: aNewMatchIndex
upTo: maxMatchIndex
```

are equal to those of the message

```
ifYouMatch: aCharacter
let: aPatternString
startingAt: aNewPatternIndex
upTo: maxPatternIndex
continueForwardMatches: aMatchString
from: aNewMatchIndex
upTo: maxMatchIndex
```

except when the receiver is a star, in which case the implementation should check whether there are any more pattern characters to match.

This completes the implementation for forward matching, as well as the rest of the implementation for `matches:` for condensed class based characters. The implementation of `caseInsensitiveMatches:` is now trivial.

3.7.9 Enumerated implementation strategy

Once the implementation for condensed class based characters becomes available, producing an equivalent implementation for enumerated class based characters is quite simple.

All the messages implemented for `ClassBasedCharacterArray` stay the same. The messages implemented in `CondensedClassBasedCharacter` can be pushed up into `ClassBasedCharacter` so they are not duplicated.

Finally, the messages implemented in the classes `CondensedPoundCharacter` and `CondensedStarCharacter` need to be copied to `ClassBasedCharacter035` and `ClassBasedCharacter042`, respectively.

This completes the implementation of `matches:` for enumerated class based characters.

3.7.10 Benchmarks

Now it is the time to see if our efforts paid off. And since we have made our implementations very flexible, it turns out that we will be able to compare six different approaches against each other.

The first approach is the implementation currently available in VisualWorks. We will refer to this version as **VW**. The second one is our improved inlined version. We will refer to this version as **VW+**. The third one is the improved inlined version running on condensed class based characters. We will refer to this version as **VW+c**. The fourth one is the improved inlined version running on enumerated class based characters. We will refer to this version as **VW+e**. The fifth one is our distinction based implementation running on condensed class based characters. We will refer to this version as **Dc**. Finally, the sixth one is our distinction based implementation running on enumerated class based characters. We will refer to this version as **De**.

Before going through the numbers, we should keep in mind a few things that make direct comparisons a bit difficult. To begin with, class based character arrays do not naturally occur in a VisualWorks image. As such, it would be unfair to include the time it takes to create them starting from regular strings. Therefore, the run time will only include the time it takes to run `matches:` or any of its varieties.

Also, as mentioned before, our implementations do not offer the possibility to evaluate a block during the process as VisualWorks' implementation of `match:` does. While this causes a small speed penalty for VisualWorks, I do not think it can skew the results very much. In addition, the feature appears to be unused, and hence we did not provide for it in our implementations.

Moreover, VisualWorks will not handle our class based characters as efficiently as regular characters. As such, our class based character implementations take a penalty for this. I will refrain from estimating how much this penalty amounts to. For more details, check the exercises.

Finally, the class based characters implementations used in these tests are not exactly the ones presented in the previous sections. There are a number of improvements that can be made so they run faster. The exercises at the end of this chapter have the details.

The benchmarks were performed by taking each of the cases benchmarked in the previous section, and evaluating them one million times by means of `timesRepeat:` for each implementation not previously tested. For convenience, the times obtained before are reproduced verbatim in the tables that follow.

Pattern set A	String to match	VW	VW+	VW+c	VW+e	Dc	De
abc	abc	632	531	677	841	772	919
ab*de#fg	abczdexfg	2,198	1,179	1,504	2,032	2,145	2,642
ab*	abczdexfg	537	529	673	740	1,014	1,060
ab#	abc	664	540	708	760	754	853
ab	xyzabczdexfg	1,750	1,803	2,216	**1,501**	2,804	2,032
*ab#	xyzabc	1,984	661	845	934	909	955
*fg	abczdexfg	3,326	513	666	734	696	786
#ab	xab	656	532	670	770	744	819
	Totals:	11,747	6,288	7,959	8,312	9,838	10,066
	VW Speed:	100%	187%	148%	141%	119%	117%
	VW+ Speed:	54%	100%	79%	76%	64%	62%

In the table above, the fifth figure in **VW+e**'s column is not a typo. Indeed, enumerated class based characters are involved in the fastest implementation for that particular case. It is also worth noting that both **Dc** and **De** run faster than **VW** in this batch. Other than that, **VW+** dominates these cases.

Pattern set B	String to match	VW	VW+	VW+c	VW+e	Dc	De
ab*de#fg	abczdexfgz	4,169	507	610	**326**	615	337
xab*de#fg	abczdexfgz	510	491	596	**325**	613	328
xab*	abczdexfg	510	567	693	**415**	782	554
ab#	xabc	510	540	714	**795**	764	868
ab	xyzahbczdexfg	5,214	5,494	6,672	**3,901**	13,297	10,939
afb	xyzahbczdexfg	5,225	5,481	6,613	**3,892**	13,406	10,982
*ab#	xyzabcd	3,070	622	758	**548**	798	577
*fgx	abczdexfg	3,170	498	608	**387**	604	384
#ab	xyab	669	543	694	**857**	755	904
	Totals:	23,047	14,743	17,958	**11,446**	31,274	25,873
	VW Speed:	100%	156%	128%	**201%**	74%	89%
	VW+ Speed:	64%	100%	82%	**129%**	47%	57%

Simply amazing. For the cases above, **VW+e** was 29% faster than **VW+**! The only difference between the two is the use of enumerated class based characters. Also, note that **VW+e** was 2 times faster than **VW**.

Pattern set C	String to match	VW	VW+	VW+c	VW+e	Dc	De
ab*de*fgz	abczdexfgz	2,639	1,968	2,766	2,865	5,725	5,956
*ab*de*fgz	xabczdexfgz	3,092	2,595	3,483	3,757	6,619	6,743
ab*de*fg*	abczdexfgz	2,471	2,510	3,373	3,557	6,400	6,392
*ab*de*fg*	xyzabczdexfgz	3,737	3,807	5,148	4,723	8,331	7,931
	Totals:	11,939	10,880	14,770	14,902	27,075	27,022
	VW Speed:	100%	110%	81%	80%	44%	44%
	VW+ Speed:	91%	100%	74%	73%	40%	40%

It seems that **VW+** dominates when expressions evaluate to `true`, while **VW+e** dominates when expressions evaluate to `false`. Will the pattern hold for the last cases?

Pattern set D	String to match	VW	VW+	VW+c	VW+e	Dc	De
`ab*de*fg`	`abczdexfgz`	3,329	497	606	**335**	611	333
`ab*de*fg`	`abczdhexfg`	3,837	3,334	4,137	**3,447**	9,838	8,980
`ab*de*fg*`	`xabczdexfgz`	529	575	610	**364**	611	358
`*ab*de*fg*`	`xyzabczdhexfgz`	5,462	5,494	7,017	**5,001**	19,871	15,958
`*ab*de*fg`	`abczdexfgz`	3,490	502	604	**375**	612	368
`*ab*de*fg`	`axbczdexfg`	4,303	3,850	4814	**3,432**	9,981	9,018
`*ab*de*fg`	`abczdhexfg`	3,978	3,426	4384	**3,747**	17,744	15,800
	Totals:	24,928	17,678	22,172	**16,701**	59,268	50,815
	VW Speed:	100%	141%	112%	**149%**	42%	49%
	VW+ Speed:	71%	100%	80%	**106%**	30%	35%

Indeed, **VW+e** dominates when expressions evaluate to `false`. Finally, here is a summary of the execution times for all six variants of `matches:`.

Pattern set	Expression value	VW	VW+	VW+c	VW+e	Dc	De
Set A	`true`	11,747	6,288	7,959	8,312	9,838	10,066
Set B	`false`	23,047	14,743	17,958	11,446	31,274	25,873
Set C	`true`	11,939	10,880	14,770	14,902	27,075	27,022
Set D	`false`	24,928	17,678	22,172	16,701	59,268	50,815
	Totals:	71,661	49,319	62,859	51,361	127,445	113,776
	VW Speed:	100%	145%	114%	140%	56%	63%
	VW+ Speed:	69%	100%	78%	96%	39%	43%

Let's review these results carefully.

The performance of **Dc** and **De** may seem like a disappointment. However, when one considers that the first non-inlined version of `matches:` was a full order of magnitude slower than **VW**, excluding the time needed to parse the pattern to create tokens, it turns out that **Dc** and **De** offer a quite impressive performance improvement factor of at the very least 5. Moreover, they achieve this without resorting to inlining, which maintains their associated cost of change extremely low by comparison.

Indeed, if we keep in mind the amount of message sends, and the number of arguments in each of those message sends, the fact that overall they run less than 2 times slower (and in some cases actually faster) than **VW** is quite

commendable! This allows us to conclude that *it is not always necessary to resort to inlining when higher performance is required.*

Especially when one looks at the performance of **VW+c** and **VW+e**, it is easy to appreciate the full impact of representing design time distinctions with classes. Even though the implementation is essentially the same, using class based characters allows these two approaches to clearly surpass **VW**. In fact, **VW+e** even beats **VW+** about half of the time — sometimes by a large margin!

In some way, **VW+** is lucky in that although it is difficult to properly inline `matches:` in a single method, it is still doable. But what would happen when the problem required a larger context in order for it to be inlined? Eventually, we would not be able to deal with the associated complexity, and inlining would become impossible. It is here, in the vast majority of cases where inlining is costly, where spending enough time designing the pieces so that they interact together to efficiently reach the answer sought really pays off.

In real life, software development challenges are hardly ever as simple and limited in scope as `matches:`. And, quite frequently, the success of the projects in which the challenges appear depends more on the cost of change being low rather than performance being extremely high.

When it comes to raw performance, however, **VW+** is the fastest overall performer of the pack. The bidirectional, greedy methodology allows it to be 45% faster than **VW** in the tests we performed. However, it claimed first place by a slim margin. We should remember that it is not clear whether **VW+e** would still be slower than **VW+** should VisualWorks' bias in favor of regular characters were to be accounted for.

We should also keep in mind that **VW+**'s performance comes at the expense of clarity of expression. Implementations based on long inlined methods are very hard to maintain. One should be able to write an implementation like **Dc** or **De**, and only inline when absolute necessary[17].

Overall, these results are fantastic already because we have clearly shown it is possible to do much better than **VW**. And yet, we have missed something. One of the advantages of enumerated class based characters is that the `asUppercase` lookups are reduced to a reference to an instance name. Since all our patterns and strings have the same case, we have not exercised this feature at all. Hence, we will rerun all the benchmarks and examine how performance changes when case lookups are necessary.

[17]Ideally, we should let the environment optimize and inline behind the scenes so we do not even have to deal with this error prone activity. Hopefully, it will become possible to do this in Smalltalk not long after this book is published.

3.7.11 Mixed case benchmarks

These tests were performed in the same conditions as the previous ones, with the exception that the string to match was upcased first in order to exercise the case insensitive matching features of each approach.

Pattern set A	String to match	VW	VW+	VW+c	VW+e	Dc	De
`abc`	`ABC`	1,150	1,242	1,443	**764**	1,539	**868**
`ab*de#fg`	`ABCZDEXFG`	3,168	2,451	3,080	**2,085**	3,736	**2,664**
`ab*`	`ABCZDEXFG`	896	963	1,194	**732**	1,538	**1,055**
`ab#`	`ABC`	1,015	1,037	1,192	**756**	1,246	**906**
`*ab*`	`XYZABCZDEXFG`	1,978	2,063	2,775	**1,581**	3,378	**2,182**
`*ab#`	`XYZABC`	2,140	1,092	1,419	**1,134**	1,405	**1,112**
`*fg`	`ABCZDEXFG`	3,306	975	1,268	**872**	1,245	**947**
`#ab`	`XAB`	1,010	1,019	1,308	**813**	1,287	**811**
	Totals:	14,663	10,842	13,679	**8,737**	15,374	**10,545**
	VW Speed:	100%	135%	107%	**168%**	95%	**139%**
	VW+ Speed:	74%	100%	79%	**124%**	71%	**103%**

These are incredible results. First of all, **VW+e** was clearly faster than **VW+** with expressions that evaluate to `true`. This shows, again, the value of creating classes to represent design time distinctions.

But there is more, because **De** was faster than **VW+**. In other words: a fully inlined and optimized method was *slower* than multiple classes and polymorphic message sends! Inlining may not lose every time, but these results categorically show how beneficial proper context design can be. Let's go to our first batch of cases for which `matches:` evaluates to `false`.

Pattern set B	String to match	VW	VW+	VW+c	VW+e	Dc	De
`ab*de#fg`	`ABCZDEXFGZ`	4,935	453	575	**387**	564	**357**
`xab*de#fg`	`ABCZDEXFGZ`	474	459	569	**375**	570	**359**
`xab*`	`ABCZDEXFG`	470	530	659	**534**	792	**568**
`ab#`	`XABC`	493	946	1,405	**838**	1,262	**823**
`*ab*`	`XYZAHBCZDEXFG`	5,175	5,093	6,694	**3,744**	12,878	9,243
`*afb*`	`XYZAHBCZDEXFG`	5,166	5,096	6,647	**3,840**	12,884	9,389
`*ab#`	`XYZABCD`	3,320	577	730	**542**	735	**509**
`*fgx`	`ABCZDEXFG`	3,454	466	574	**427**	575	**385**
`#ab`	`XYAB`	655	958	1,326	**935**	1,280	**845**
	Totals:	24,142	14,578	19,179	**11,622**	31,540	**22,478**
	VW Speed:	100%	166%	126%	**208%**	77%	**107%**
	VW+ Speed:	60%	100%	76%	**125%**	46%	65%

And again, **VW+e** simply achieved a level of performance that **VW+** could not. Even **De** gives an astonishing performance by beating **VW**, and sometimes being faster than **VW+e**!

Pattern set C	String to match	VW	VW+	VW+c	VW+e	Dc	De
`ab*de*fgz`	`ABCZDEXFGZ`	3,833	3,346	4,631	**3,097**	7,452	6,180
`*ab*de*fgz`	`XABCZDEXFGZ`	4,258	3,926	5,360	**4,014**	8,301	6,873
`ab*de*fg*`	`ABCZDEXFGZ`	3,431	3,562	4,881	**3,404**	7,818	6,283
`*ab*de*fg*`	`XYZABCZDEXFGZ`	4,535	4,741	6,482	**4,555**	9,628	7,816
	Totals:	16,057	15,575	21,354	**15,070**	33,199	27,152
	VW Speed:	100%	103%	75%	**107%**	48%	59%
	VW+ Speed:	97%	100%	73%	**103%**	47%	57%

By now, it is evident that **VW+e** is dominating the mixed case benchmarks. And while **De** falls short in this case, it does so by no more than a factor of 2 — still very impressive!

Pattern set D	String to match	VW	VW+	VW+c	VW+e	Dc	De
`ab*de*fg`	`ABCZDEXFGZ`	4,209	447	585	**404**	570	403
`ab*de*fg`	`ABCZDHEXFG`	3,987	4,064	5,260	**3,269**	11,419	9,446
`ab*de*fg*`	`ZABCZDEXFGZ`	470	522	573	**361**	568	393
`*ab*de*fg*`	`XYZABCZDHEXFGZ`	5,426	5,507	7,094	**4,518**	20,956	16,016
`*ab*de*fg`	`ABCZDEXFGZ`	4,403	457	564	**351**	585	316
`*ab*de*fg`	`AXBCZDEXFG`	4,031	4,023	5,259	**3,176**	11,351	8,345
`*ab*de*fg`	`ABCZDHEXFG`	4,121	4,129	5,416	**3,452**	19,334	15,285
	Totals:	26,647	19,149	24,751	**15,531**	64,783	50,204
	VW Speed:	100%	139%	108%	**172%**	41%	53%
	VW+ Speed:	72%	100%	77%	**123%**	30%	38%

Once more, **VW+e** flies by **VW+**, while **De** holds its ground. This is reflected in the summary below.

Pattern set	Expression value	VW	VW+	VW+c	VW+e	Dc	De
Set A	`true`	14,663	10,842	13,679	**8,737**	15,734	10,545
Set B	`false`	24,142	14,578	19,179	**11,622**	31,540	22,478
Set C	`true`	16,057	15,575	21,354	**15,070**	33,199	27,152
Set D	`false`	26,647	19,149	24,751	**15,531**	64,783	50,204
	Totals:	81,509	60,144	78,963	**50,960**	145,256	110,379
	VW Speed:	100%	136%	103%	**160%**	56%	74%
	VW+ Speed:	74%	100%	76%	**118%**	41%	54%

What can we learn from these tests? First of all, that the true best performer is **VW+e**. Since its implementation is the same as that of **VW+**, we also see that *when classes are used to represent design time distinctions, it is possible to obtain significant performance boosts* — even when considering VisualWorks' preference for regular characters. A perfect example of this fact is that, without resorting to inlining, **De**'s performance reached 74% of the speed of **VW**, while that of **Dc** fell behind by less than a factor of 2.

The final conclusion below may seem counterintuitive, and for understandable reasons, but the numbers support it beyond any degree of skepticism. We had previously concluded that inlining was not necessarily the only way to improve performance. And now, in addition, we conclude that *creating classes makes it possible to obtain valuable performance improvements on top of those obtained by means of inlining*. In other words, even when favoring speed to maintainability, inlining alone is not the *best* answer.

3.7.12 Reflections

These results have strong, far-reaching consequences. Do you remember all those carefully inlined methods you have seen? Well, they are immediately suspect of being far from optimal. And while the cause seems to be a matter of optimization technique, it is not exactly the case.

The issue here is that by means of reflecting upon what we already know, and by being able to reify it explicitly in Smalltalk, we allow computer programs to run faster because we provide access to a more sophisticated way of perceiving differences in value.

Many readers are probably familiar with the phrase "intention revealing", as it is used in computer programming particularly when choosing proper names for things. However, it really ought to be interpreted in a much more encompassing context. It should be seen in terms of being successful at expressing intentions. Since intention is what causes distinctions to come into existence, and since in Smalltalk different kinds of distinctions are modeled by different kinds of classes, it clearly follows that revealing our intentions in an expressive manner also implies creating classes for the distinctions we perceive.

This also shows how beautifully Smalltalk works as a mirror in which we can see our own understanding. Because when the feedback cycle is so tight and the switch time is so short, it is almost too easy to observe something outside of yourself, to draw new distinctions based on it, then reverse directions and try to artfully express your intentions once more, only to start the whole process

one more time — as if iterating traversal steps through an information space, in search of your own attractor.

Welcome to a fascinating, life long adventure of self discovery made possible by environments like Smalltalk.

3.8 Exercises

Exercise 3.12 [05] Prove that the relationship *is covered by the GPL just like*, over the set of all pieces of software, is an equivalence relation.

Exercise 3.13 [08] A *partial, strict order relation* is a relation R over a set A, such that it is areflexive (R has no pairs of the form (x, x)), antisymmetric (if the pair (x, y) is in R, then the pair (y, x) is not in R), and transitive.

Show that the relationship *is a derivative work of*, over the set of all pieces of software covered by the GPL, is a partial, strict order relation.

Exercise 3.14 [16] Express the intentions of the piece below in a less verbose manner.

```
| wordString |
wordString := wordStream contents.
wordString isEmpty ifFalse: [^Word on: wordString]
^nil
```

Exercise 3.15 [15] Let's say your internal process decides to go to the train station and catch the next train. Trains are not running on anymore, but this knowledge is not accessible to you. Exactly how do *you* recover from this situation and avoid waiting forever?

Exercise 3.16 [50] Find an interesting property of the topology for the circuit implied by the Syracuse function, and solve the open problem of whether all traversals will eventually reach 1.

Exercise 3.17 [32] *(Blaine Buxton)* The implementations of both **Dc** and **De** feature messages that at time require numerous arguments. What could be done to alleviate this problem?

Exercise 3.18 [27] Use class based characters to simplify the piece below.

```
nextWord

  | return |
  wordStream := String new writeStream.
  return :=
    [
      | wordString |
      wordString := wordStream contents.
      wordString isEmpty ifFalse: [^Word on: wordString]
    ].
  input do:
    [:next |
      next isAlphaNumeric
        ifTrue: [wordStream nextPut: next]
        ifFalse: [return value]
    ].
  return value.
  ^nil
```

Exercise 3.19 [50] Find a polynomial time traversal of the integers such that if you start at $N > 1$, you either reach proper factors of N, or conclude that N is prime.

Exercise 3.20 [38] *(Leandro Caniglia and Valeria Murgia)* You are given two points in a discrete plane, an origin point and a destination point. From the origin point you need to find a traversal through the plane such that you arrive at the destination point with the following restrictions. You can only exit the origin point in a certain direction, and you can only reach the destination point from a certain direction. There is also a minimum length constraint for each segment in the traversal (for instance, you could have to ensure all segments measure at least 3). The only directions allowed for travel are up, down, left and right.

For example, if you were to exit `0@0` towards the left and arrive at `4@7` from below, with a minimum segment length of 3, a correct traversal would be `0@0`, `-3@0`, `-3@4`, `4@4`, `4@7`. On the other hand, if you had to arrive at `1@1` from below, a correct traversal would be `0@0`, `-3@0`, `-3@-3`, `1@-3`, `1@1`.

Implement a traversal finder for the above scenario such that it finds correct traversal that have a minimum possible number of segments. In addition, the length of the whole traversal must be of the minimum required length.

Exercise 3.21 [12] What is wrong with the method below?

```
Matrix>>isNull

  1 to: self dimension y do:
    [:eachY |
      1 to: self dimension x do:
        [:eachX |
          | eachRow |
          eachRow := self rowAt: eachY.
          (eachRow at: eachX) isNull ifFalse: [^false]
        ]
    ].
  ^true
```

Exercise 3.22 [10] Why are instance names necessary to store uppercase and lowercase counterparts of enumerated class based characters?

Exercise 3.23 [15] Why is the value of the expression below empty?

```
Character allInstances
```

Exercise 3.24 [18] How would you refine the abstract `matches:` test case to support class based character arrays without changing any of the test messages?

Exercise 3.25 [12] Find at least 18 messages currently implemented in the class `Character`, the implementation of which would be simpler with enumerated class based characters.

Exercise 3.26 [30] The astute reader may have noticed that there are redundant comparisons in the implementation of `matches:` for class based characters, as discussed in the chapter. Find and eliminate such superfluous checks.

Exercise 3.27 [25] In addition, the astute reader may have noticed that, in the absence of duplicated checks, some messages become redundant as well. Find and eliminate such superfluous messages.

Exercise 3.28 [41] Use enumerated class based characters to implement all the features of regular expressions.

Exercise 3.29 [13] It is claimed that the implementation of **VW+** and **VW+e** are equal, but there is a catch. What has to be different between them? How should this be resolved when implementing **VW+e**?

Exercise 3.30 [35] Examine the game Life. You will see that, for each iteration of the game, it is necessary to sum the neighbors of each cell. A traditional approach would be to use small integer arithmetic. However, this will cause `ifTrue:ifFalse:` in the form of a switch statement to choose between death by starvation, survival, spawning, or death by overpopulation. In addition, you may encounter further logic depending on whether the sum corresponds to a live or a dead cell.

Find out if it is possible to run these calculations faster by using a class hierarchy such as the one shown below.

```
AbstractNeighborCount
  ZeroNeighbors
  OneNeighbor
  TwoNeighbors
  ThreeNeighbors
  TooManyNeighbors
```

Exercise 3.31 [13] Simplify the method below.

```
Behavior>>depth

  | answer |
  answer := 0.
  self superclass notNil ifTrue:
    [answer := self superclass depth + 1].
  ^answer
```

Exercise 3.32 [10] Is the fact that numbers of many classes understand the message `squared`, which is implemented only once, an example of polymorphism?

Exercise 3.33 [14] What messages must be understood by every object?

Exercise 3.34 [12] Make the following method run faster.

```
Behavior>>includesBehavior: aClass

  ^self == aClass or:
    [
      superclass notNil and:
        [superclass includesBehavior: aClass]
    ]
```

Chapter 4

Validation revisited

For some of you, the previous chapter may have been the first exposure to a fundamentally different way of thinking about designing and creating computer programs. While we will have a chance to take another look at it, let's take a break for now while it sinks in — so how about some good old refactoring to refresh ourselves?

4.1 Motivation

Domain object validation is important in any application. Since validation rules change frequently due to application enhancements and revisions of business rules, the actual validation work should be performed by a light and flexible framework.

Agile!

While this may sound reasonable on paper, unfortunately it seems to me that things hardly ever turn out that way in real life. This observation may be just a matter of unlucky twists of my own experience, and as such could very well be biased. Nevertheless, for illustration purposes, here are some brief descriptions of conventional approaches to validation I have seen applied in practice.

4.1.1 Testing message protocol

One way to handle validation of domain objects is to implement messages such as `DomainObject>>isValid`, and have these return `true` or `false`. This causes problems very quickly, because what one needs to do right after asking `isValid` is to complain if validation fails. In other words,

```
SomeUIClass>>takeAction

  self domainObject isValid ifFalse:
    [
      self complainBecause: 'Validation failed'.
      ^self
    ].
  self basicTakeAction
```

The first observation to make is that if `isValid` provides boolean results, any potential complaints cannot be very specific as to what went wrong. This is a problem that needs to be addressed, because it will be hard for whoever receives these messages to track down the cause of the validation failure if all they are told is, essentially, to try again.

Unless we are talking about a slot machine, of course.

At this point, we have at least two options. The first one is to break the `isValid` methods into a multitude of smaller ones like `isFirstDetailValid`,... , `isNthDetailValid`. Once this is done, then we can write more complicated logic to determine which message to show the user. But that gets out of hand immediately. At the same time, it increases the cost of change to unacceptably high levels. Some of the issues with it include: having to make sure each and every place where validation is used sends *all* the validation messages (unless some are not appropriate for the particular context where validation is performed, in which case one has to document their purposeful omission), having the same pattern of code repeated over and over again, and the inability to specify the failure text once and only once. Further inquiry only reveals further problems. Clearly, this approach is going nowhere.

The other option is to push the act of complaining to the context where the problem is known: the `isValid` message itself. However, rearranging things to obtain locality of information in this way carries an expensive price tag. The consequence is that the implementation of the `isValid` messages becomes a sequence of blocks like the one shown below.

```
self instanceName isValidAccordingToSomeCriteria ifFalse:
  [
    Dialog warn: 'Do not do that again'.
    ^false
  ]
```

And the problem with this approach is that domain object validation, on its own, ought not to depend on the existence or functionality of a user interface. Why should some window need to be open when trying to figure out if some object is in a valid state or not?

One could go ahead and hack `Dialog class>>warn:` to do nothing if running headless, but that is not the point.

The inability to provide a boolean answer without opening a dialog or causing any other user interface side effect is a problem not only because of our desire to have a reasonable distribution of responsibilities. Server side programming typically involves performing work without a user interface — in other words, it runs *headless*. Allowing these direct reference to GUI widgets increases the cost of change and limits the usage scope of our application for no good reason.

Oh, and by the way, each of those code blocks represents a distinction which is now blurred because all the blocks are in the same method. Lovely.

To make the story short, this approach just does not work.

4.1.2 Querying message protocol

In order to get around these limitations, sometimes it may seem like a good idea to implement messages such as `validationErrors` instead. These messages typically answer a collection of error message strings which can be reused by the user interface. If the program is running headless, then at least it is possible to tell whether an object is valid or not by ensuring that `validationErrors` answers an empty collection.

While this approach is a big improvement over `isValid` messages, it is still deficient. To begin with, it does nothing to prevent code repetition. The template for `validationErrors` would look like this.

```
SomeDomainObject>>validationErrors

  | answer |
  answer := OrderedCollection new.
  self firstInstanceName isValidAccordingToSomeCriteria
    ifFalse: [answer add: 'First instance name is wrong'].
  .
  .
  .
  self nthInstanceName isValidAccordingToSomeCriteria
    ifFalse: [answer add: 'Nth instance name is wrong'].
  ^answer
```

This is still contains a lot of redundancy. All we seem to have done is to replace `Dialog warn:` with `answer add:`. Moreover, it may be quite useful to know more details about each particular failure. For example, if the instance name that fails validation were known, we could arrange that the UI (if open) highlights the corresponding widget. Alas, the needed information remains hidden. Answering just strings is not very informative from a reflection point of view.

While we could begin to address this shortcoming by having the message `validationErrors` answer a collection of validation failure objects instead of plain strings[1], we would still fail to prevent code repetition.

But to really add insult to injury, most validation complaints are very similar to each other. Here are a few message patterns that occur quite frequently.

- The value of some instance name is required.
- The value of some instance name is not formatted as required.
- The value of some instance name violates business rules.
- The value of some instance name is outside its allowed range.

Do you realize that the one who will type all of these cookie cutter messages, one by one, is *you*?

In short, there is just way too much duplication. The implementation itself is redundant. The validation failure messages are repetitive as well. Maintenance and standard compliance work are error prone and time consuming *by design* in this context. It just seems like we cannot possibly win with these deficient strategies.

What these problems suggest quite strongly is that we need a new way to do validation. Now we need to find it.

4.2 Intention

Exactly what is validation? What is this sequence of checks and complaints we saw in the previous section? What is the *purpose* of validation? What is our *intention* when we do it?

[1] Associations of the form `instanceName` → `failureMessage` are sometimes used to do this. Of course, sending `value` or `key` is not intention revealing at all. In this case, using associations is equivalent to using arrays of size 2.

We could try to guess the answers to these questions by taking a look at the complaints we show the users when something fails validation. Such and such *is required*. So and so *must be in the right format*. These things we show to users when there is a problem seem to be rules, and as such validation becomes the process of enforcing rules.

Validation is the object police!

Moreover, as far as object validation is concerned, there are typically multiple rules that need to be satisfied for validation to pass. So, for the purposes of validation, there is this sort of checklist consisting of individual rules.

So if we have a checklist and validate an object, we would obtain some sort of diagnostic as our answer: which rules passed and which rules did not. If our validation is good, perhaps our answer will include some explanation as to why failures occurred. And if our validation is *really* good, it might even tell us what pieces of validation blew up and did not finish at all.

But we have already seen this picture before!!!

	Validation	`SUnit`
Rules cluster in	checklists	test cases
A result is obtained by	evaluating the checklist	running the tests

And therefore, implementing validation in terms of `SUnit` seems very reasonable — and even irresistible. Validation *is* testing. When we think about testing in terms of compliance to the rules, we see that many things like validation or even security enforcement can be put in terms of test cases.

The main advantage of `SUnit` is that it is an excellent place from which to go about enforcing rules. This is because it was designed to do exactly that. Thus, instead of using the intention obscuring techniques we reviewed earlier, we could have each validation checklist represented by a test case. Every one of the individual validation rules, typically represented by the template blocks we saw earlier, would become an individual test message in the test case. Since these small test messages will look like many others, we will be able to factor the frequently used rules out of these methods and write them once and only once as helper messages. In other words, *using `SUnit` for validation enables refactoring across the whole application.*

Test cases full of test methods represent our checklist approach very well. Updating the rules would become extremely easy: just implement, update or delete small methods. And since user interface handling will be out of the picture, and because `SUnit` handles failures for us[2], the validation methods will contain

[2]In addition, by handling every exception, `SUnit` based validation will resist things like MNUs and primitive failures.

the rules and nothing but the rules. The cost of change associated with managing this approach is almost negligible.

It seems to be that we have the blueprint of a validation framework in place. The only obstacle is that `SUnit` is typically considered a development-time tool, while we would like to use it for runtime purposes instead. To avoid showing the test runner to the user, we will need to examine how `SUnit` works so we can take advantage of it by running it headlessly.

4.3 Adapting `SUnit` to validation

Before we jump right into implementing `SUnit` based validation, let's briefly go over how `SUnit` accomplishes its goals.

4.3.1 `SUnit` behavior recap

As we know, test cases implement test methods that are executed one at a time. A test suite is a collection of test case instances. These test case instances are set up to execute only one of their test methods, by storing the selector in an instance name called `testSelector`.

When the test runner executes one or more test cases, it builds a test suite containing test case instances for all test messages involved. A test suite may also contain test suites. This arrangement is useful for running whole hierarchies of test cases.

A test result has the responsibility of collecting the results of executing a test suite. It registers whether each test passed, failed, or ended in error, by sorting individual test cases into passed, failed, or ended in error buckets. This information is what is presented to us by typical test runners.

Although this is not exactly what we need, it is quite close. We should make the distinction between ***`SUnit` failures*** (which occur as the result of asserting `false`), and *validation failures* (which occur when the validation rules are not met). A validation failure should know which object was being tested, what aspect of the object was found to be defective, and some text description of the problem.

Since `SUnit` failures and validation failures are quite different, it would be desirable to have the test result record them separately. In this way we avoid being cornered into sending `areYouThisOrThatKindOfFailure` messages, followed by `ifTrue:ifFalse:`.

Regular test failures are detected by SUnit when TestFailure, the SUnit failure exception, is raised. In the same manner, SUnit detects errors when any other unhandled exception occurs. As we want test results to distinguish between validation failures and errors, we will have to introduce our own validation failure exception.

SUnit terminates test methods when any exception goes unhandled. This avoids the situation in which the developers would have to explicitly trap and take care of all possible failures and errors in each test method. In turn, this allows us to write test methods as if exceptions never occurred[3], which results in code that is much easier to read. To take advantage of this design feature, we will let the execution of the test method be interrupted when a validation failure occurs.

It may be a good idea to make it easy to distinguish between regular test cases and validation test cases. To accomplish this, we could have the validation test selectors start with the prefix validate instead of the default SUnit prefix test. It is more intention revealing to the eye as well.

Finally, it would be great if we had a convenient validate: message on the class side of our validation test cases. In this way, we will be able to easily perform validation on objects without having SUnit's internal details in the way.

By subclassing SUnit's classes instead of changing their behavior, we can confidently rely on regular SUnit for guidance while we implement all the new functionality. If we need to do some refactoring in SUnit itself, we can verify the refactoring is correct using SUnit's own SUnit tests.

Since the reader surely has written tests for the features we discussed (cough! cough!), we will now move on to making those tests pass.

4.3.2 Validation failures

When SUnit runs a test case, it wraps its execution with two exception handlers. This is done in the message TestResult>>runCase:. The innermost exception handler is set up against TestFailure, and keeps track of which test cases ended up asserting false. Wrapped around that handler, there is a more generic handler that detects errors by catching anything the first handler does not take care of. If no exceptions are left unhandled during the execution of the test case, SUnit records a pass. The implementation is equivalent to the piece shown below.

[3]By definition, exceptions model the occurrence of *exceptional* events.

```
TestResult>>runCase: aTestCase

  | testPassed |
  testPassed :=
    [
      [
        aTestCase run.
        true
      ]
        on: TestFailure
        do:
          [:ex |
            self recordFailureOf: aTestCase.
            ex return: false
          ]
    ]
      on: Error
      do:
        [:ex |
          self recordBreakdownOf: aTestCase.
          ex return: false
        ].
  testPassed ifTrue: [self recordSuccessOf: aTestCase]
```

As we discussed, we will need our validation failures to have a bit of specific behavior. At first sight, it may seem convenient to enhance `SUnit`'s own test failure exception, `TestFailure`. However, this causes problems. If we enhanced `TestFailure` to take care of our validation needs, we would be forced to write something like `isTheRightKindOfException` — which would be immediately followed by `ifTrue:ifFalse:`. Moreover, we would be forced to put all that mess in places such as inside the test failure exception handler shown above. We just do not need that kind of thing. That method is elaborate enough already — and besides, why do we need to draw design time distinctions at run time?

This hints at creating a new exception for `SUnit` to handle. If we created a subclass of `TestFailure` directly, however, we would have to be particularly careful when writing the exception handlers. This is because the handler for `TestFailure` would also handle our subclassed exceptions. Since the hierarchy relationship between the classes would not be explicitly represented in the method

containing the handlers, the use of three nested exception handlers would turn out to be a good opportunity for confusion to arise.

So, instead of refining `TestFailure`, we will distinguish validation failure exceptions by means of new subclass of `Error`. We will conveniently call it `ValidationFailureException`. These exceptions will refine `Error` by adding an instance name to hold the validation failure associated with them. In this way, we will be able to keep this information available long after failures occur.

All this means that we will have to add another nested exception handler so `SUnit` can record validation failures. To preserve the functionality of vanilla `SUnit`, we will subclass `TestResult` so that we can refine `runCase:` freely. In the same way that `TestResult` maintains counts of test failures and errors, instances of `ValidationResult` will also have instance names dedicated to keep track of validation failure counts.

4.3.3 Validation suites

Instances of `TestResult` are created by the test suite in order to run itself. Since we should preserve the behavior of `SUnit`, we will also need to make a subclass of `TestSuite` to accommodate the creation `ValidationResult` instances. We will call this subclass `ValidationSuite`.

This, in turn, causes us to modify the creation of test suites. Suites are created by test case classes. To allow for this modification, we will create a subclass of `TestCase`. We will call this subclass `AbstractValidator`. All validation test cases will be subclasses of this class.

Now we need to make our abstract validator test cases search for selectors starting with the prefix `validate` instead of `test`. It turns out that the pattern is mentioned in two messages implemented by `TestCase class`. We could easily refine them in `AbstractValidator class`, but there is a better solution that makes `SUnit` easier to change for others as well: why should that constant be referenced in more than one and only one place?

That is the issue, precisely. Therefore, we should put this prefix in a message called `selectorPattern`, for example. By refactoring `TestCase class` so that the selector prefix is referenced once and only once, we just need to refine the message `selectorPattern` in `AbstractValidator class`[4].

[4]Do you remember how having a single reference to the implementation specific details allowed us to build parallel test hierarchies? From a more profound point of view, the issue is that *the value of the call made again is the value of the call*. Since the duplicated references provide no additional value, they should be avoided.

Refactoring in this way eliminates code duplication — both the repetitiveness of having to specify the same constant multiple times, as well as the redundancy caused by having to copy a whole method from a superclass in order to change a tiny fraction of it. This kind of refactoring also makes things easier for us in the long run because, should maintenance become necessary, there will be no duplicated code to maintain[5]. In addition, it is a more refined expression of our understanding. Change can be accomplished so easily in `Smalltalk` that there is no valid excuse to let go of these benefits.

Back to our problem. Validation test cases will have an instance name called `object`. This instance name will hold the object being validated. This means that before validators run, they need to be provided with the object being validated. Adding these small changes to our subclasses is very straightforward. When the suite is being built at validation time, we can send it the message `object:` with the object being validated as an argument. The suite will in turn send `object:` to all of its individual test cases. If it contains other suites too, the same implementation will make sure that all individual test cases get what they need before the suite is run.

For convenience, we should implement the message `validate:` on the class side of `AbstractValidator`. This message will create the validation suite, run it, and answer the corresponding validation result.

4.4 A migration example

Overall, it seems that the cost of change that comes with this framework is almost ridiculously low. It may even feel like it cannot possibly work because it is too simple. That is why we should suspect it is on the right track. To ensure that this is so, we should take a sample implementation of `isValid` and rewrite it in terms of `SUnit` validation.

Let's think one up. We could be validating support calls, for example. For business reasons, these calls should be attached to a client. In addition, we should make sure that support calls tell which representative took care of them. These calls should be categorized in some manner, and for some of those categories we could easily have subcategories. Finally, the date should also be required, and in addition this date should not be in the future. The method below summarizes these requirements.

[5]Writing things once and only once also means any necessary change will have to be done once and only once.

Yes, you can do it. Take a deep breath and read on.

The code does not look very readable with that compact formatting. Those blocks could use the lessons learnt best by using curly braces. Unfortunately, if the implementation were expanded, it would not fit on the page. Sorry!

```
SupportCall>>isValid

  self client isNil ifTrue:
    [Dialog warnWithBeep: 'Client is required'.
    ^false].
  self representative isNil ifTrue:
    [Dialog warnWithBeep: 'Representative is required'.
    self highlightWidgetFor: #representative.
   ^false].
  self supportCategory isNil ifTrue:
    [Dialog warnWithBeep: 'Category is required'.
    self highlightWidgetFor: #supportCategoryDescription.
    ^false].
  self supportSubCategory isNil ifTrue:
    [| subCategories |
    subCategories := self possibleSubCategories.
    subCategories isEmpty ifFalse:
      [Dialog warnWithBeep:
        'A subcategory is required in this case'.
      self highlightWidgetFor:
        #supportSubCategoryDescription.
      ^false]].
  self date isNil ifTrue:
    [Dialog warnWithBeep: 'Date is required'.
    self highlightWidgetFor: #date.
    ^false].
  self date > Date today ifTrue:
    [Dialog warnWithBeep: 'Date cannot be in the future'.
    self highlightWidgetFor: #date.
    ^false].
  ^true
```

Ahhh, I agree: how very painful. The domain object requires that there is a user interface open so that dialogs can be opened. The `isNil` checks are shamelessly copied & pasted. The boolean returns, the widget highlighting requests (which makes the dependency on the user interface even stronger)... even the text messages are repeated.

Now imagine this for *all your domain objects*, and think of the assertions upon which methods like the one above were written. Surprise, they could change. What if users did not like beeping anymore, or wanted it depending on the kind of problem encountered? Or what if the text messages needed to follow a different standard? What if dates could be up to one day in the future instead? What if we needed to use `isValid` for some internal conversion process that had to run headlessly? Do you see yourself going through all the implementors of `isValid`? The lack of coherency means that the cost of change would be enormous. In short: what a mess!

All these potential issues imply that the meat of the refactoring problem we face is that we have to enforce unique expression of intention, distinction and behavior for validation purposes. Based on this constraint, let's see where the traversal takes us in the space of all possible validation implementations.

4.4.1 Migrating `nil` checks

The first piece we have to convert is below.

```
self client isNil ifTrue:
  [
    Dialog warnWithBeep: 'Client is required'.
    ^false
  ]
```

While it seems more or less obvious that we will have to provide a method for a message called `validateClient`, at this time it is not clear how the message would be implemented. A rough transcription would be as follows.

With a strong iterative and contractive process, any *traversal will inevitably converge towards its attractor regardless of the starting point.*

```
SupportCallValidator>>validateClient

  self object client isNil
    ifTrue: [self failValidationBecause: 'Client is required']
```

Our first iteration seems to provide almost no benefits regarding code size or the elimination of redundancy. However, a significant piece has appeared: the message `failValidationBecause: aString`. Clearly, we should avoid polluting every single validation method with the creation of `ValidationFailure` objects, so this is something to keep.

But let's leave that in the back burner for now and concentrate on what else

we could remove from the method above. After a bit of observation, it seems that there is this idea of a generalized `nil` check that fails whenever the argument given to it is undefined. Clearly, this check will be used in numerous validation methods. As such, *it needs to be refactored into its own unique representation.* In other words,

```
SupportCallValidator>>validateClient

  (self generalizedNilValidationWith: self object client)
    ifFalse: [self failValidationBecause: 'Client is required']
```

Iterate, contract, repeat.

Good, now we do not see the actual `nil` check anymore. What could we eliminate next? Well, if we invoke the generalized `nil` check, our intent is to complain if it fails. Then why should we have to send the message `failValidationBecause:` every single time? Let's add that to the generalized `nil` check instead. After all, the validation failure exception will be caught by `SUnit` regardless of where it is raised from. Therefore,

```
SupportCallValidator>>validateClient

  self
    generalizedNilValidationWith: self object client
    andMessage: 'Client is required'
```

This is definite improvement. Our validation methods now have no occurrences of `ifTrue:ifFalse:` nor syntax sugar parentheses. However, there is still much duplication left to be taken care of.

For example, what about the failure message? If we mean to validate the *client*, then the failure message would be "*Client* is required". If we mean to validate the *user*, then the failure message would be "*User* is required". The "is required" part stays the same, and the only thing that changes is the beautified version of the instance name in question.

Of course we always choose good names, right?

Did we just say that since we chose good instance names for our objects to begin with, now we can reuse them so we do not have to explicitly write boilerplate validation failure messages ever again? Yes, exactly.

And since this is Smalltalk, we have enough reflection powers to take care of this once and for all. A way to make this work is to supply the instance name to the generalized `nil` check so it can both retrieve the value to validate and build the failure message. In other words,

```
SupportCallValidator>>validateClient

  self generalizedNilValidationForAspect: #client
```

Even though this looks better, we can still improve upon it. What if we have to perform further validation once we know that `client` is not `nil`? Providing the aspect name as an argument every single time would become repetitive:

```
SupportCallValidator>>validateClient

  self generalizedNilValidationForAspect: #client.
  self generalizedSomeOtherValidationForAspect: #client
```

We could avoid sending the same object instance name argument over and over again if we kept it in an instance name of the validator. Therefore, we will add the instance name `aspect` to `AbstractValidator` and rewrite our validation method one last time.

```
SupportCallValidator>>validateClient

  self aspect: #client.
  self valueIsDefined
```

Wow, that even looks like an *assertion*! How interesting that we end up making assertions in the context of `SUnit`. And the implementation reads very nice as well! You can even hear the conversation going... *When are the clients of a support call considered valid? Well, of course, when the aspect* `client` *is defined.* It does not get much more clear than that.

Let's take a look at `valueIsDefined` now. Since all that changes is the failure message, we could easily implement it as follows.

```
AbstractValidator>>valueIsDefined

  self value isNil
    ifTrue: [self failValidationBecause: self isRequired]
```

Note the care taken not to duplicate `perform:` sends across all validation helper methods by delegating the work to the message `value`.

```
AbstractValidator>>value

  ^self object perform: self aspect
```

Also, since the aspect being validated is remembered in an instance name, then it becomes part of the static context. As such, it is not necessary to pass it as an argument to `isRequired`.

```
AbstractValidator>>isRequired

  ^self aspect prettyPrint, ' is required.'
```

This takes care of all the `nil` checks. Now we can take the next step and examine a slightly more complicated case.

Note that `SUnit` Based Validation, by its very design, is a framework that fosters and facilitates the creation of a library of code — in this case, of validation checks. Very few frameworks accomplish this.

4.4.2 Migrating composite validation rules

Let's consider what we could do with the piece below. Note that in this case, the second block depends on the first one preventing `nil` date values from reaching it.

```
self date isNil ifTrue:
  [
    Dialog warnWithBeep: 'Date is required'.
    self highlightWidgetFor: #date.
    ^false
  ].
self date > Date today ifTrue:
  [
    Dialog warnWithBeep: 'Date cannot be in the future'.
    self highlightWidgetFor: #date.
    ^false
  ]
```

Following the pattern we established for `nil` checks, all we need to do is just to write a generalized *not in the future date* check and use it. This is almost too easy to accomplish — the cost of change is essentially zero! In other words,

```
AbstractValidator>>valueIsNonFutureDate

  [self value isKindOf: Date],
  [self value <= Date today] ifAnyFalse:
    [self failValidationBecause: self cannotBeInTheFuture]


AbstractValidator>>cannotBeInTheFuture

  ^self aspect prettyPrint, ' cannot be in the future.'
```

Now we can use the check we wrote, and the validation for `date` ends up as follows.

```
SupportCallValidator>>validateDate

  self aspect: #date.
  self valueIsDefined.
  self valueIsNonFutureDate
```

Since `SUnit` will stop execution of methods as soon as an exception is not handled, we do not have to worry about `valueIsNonFutureDate` if `valueIsDefined` fails. No `ifTrue:ifFalse:` or early returns from the method are required.

4.4.3 Migrating nested validation rules

Finally, we are in a position where we can confidently tackle the last piece of our original `isValid` method. For convenience, its first part is reproduced below.

```
self supportCategory isNil ifTrue:
  [
    Dialog warnWithBeep: 'Category is required'.
    self highlightWidgetFor: #supportCategoryDescription.
    ^false
  ]
```

This seems like a regular `nil` check. There is an important difference, however. The user interface was built so that a description of the category is shown in the drop down list. These descriptions are not the categories themselves, and as such

the instance name `supportCategory` does not match the widget aspect in the user interface, `supportCategoryDescription`. What to do?

Well, we could get things going if we simply instructed `SUnit` validation to complain about an aspect other than the one it is looking at. Maybe we should just do that. In other words,

```
SupportCallValidator>>validateCategory

  self aspect: #supportCategory.
  self aspectForReporting: #supportCategoryDescription.
  self valueIsDefined
```

Done, problem out of the way. Note that by using accessors and then invoking the desired bit of behavior, we do not have to create messages with numerous keyword arguments — much less for all the possible argument subsets, which would quickly become unmanageable. This approach of pushing all the buttons (or loading all the registers) and then turning the crank allows us to write very little code. Now we can tackle the final piece of `isValid`, shown below.

```
self supportSubCategory isNil ifTrue:
  [
    | subCategories |
    subCategories := self possibleSubCategories.
    subCategories isEmpty ifFalse:
      [
        Dialog warnWithBeep:
          'A subcategory is required in this case'.
        self highlightWidgetFor:
          #supportSubCategoryDescription.
        ^false
      ]
  ]
```

We have two options at this point. The first one is to make a new validation method to validate the subcategory. However, it does not make much sense to validate the subcategory when the category itself could be invalid. Hence, we will go with the second option: add the subcategory validation piece to the category validation method. After all, if the category validation fails, then the validation of the subcategory will not run.

```
SupportCallValidator>>validateCategories

  | subCategories |
  self aspect: #supportCategory.
  self aspectForReporting: #supportCategoryDescription.
  self valueIsDefined.
  subCategories := self object possibleSubCategories.
  subCategories isEmpty ifFalse:
    [
      self aspect: #supportSubCategory.
      self aspectForReporting: #supportSubCategoryDescription.
      self valueIsDefined
    ]
```

This completes the conversion of the original `isValid` method. Even without considering the benefits arising from having an explicit validation framework, the number of statements required to implement the validation behavior have gone from 26 down to 15, a nice reduction of about 40% in code size.

4.4.4 Enhancing failure messages

We were lucky because our instance names were user-friendly. However, it is entirely possible that we could have to provide custom pretty-printable versions of instance names. In our load the registers and turn the crank model, this is extremely easy to do: we just add an instance name to validators and proceed as follows.

```
AbstractValidator>>prettyPrint

  ^prettyPrint ifNil: [self aspect prettyPrint]
```

Once we do this, the failure message templates do not need to reference the `aspect` anymore.

```
AbstractValidator>>isRequired

  ^self prettyPrint, ' is required'
```

This additional layer of indirection allows us to enhance our validation methods as shown below.

```
SupportCallValidator>>validateCategories

  | subCategories |
  self aspect: #supportCategory.
  self aspectForReporting: #supportCategoryDescription.
  self prettyPrint: 'Category'.
  self valueIsDefined.
  subCategories := self object possibleSubCategories.
  subCategories isEmpty ifFalse:
    [
      self aspect: #supportSubCategory.
      self aspectForReporting: #supportSubCategoryDescription.
      self prettyPrint: 'Subcategory'.
      self valueIsDefined
    ]
```

Cost of change: essentially zero. Customer satisfaction: absolutely priceless.

4.4.5 Handling validation failures

Earlier on, we ran into the message `failValidationBecause:` but did not specify its implementation. In simple terms, we need to create the validation failure object and raise the necessary exception with it. Well... that is simple enough!

This is one of my all time favorite selectors.

```
AbstractValidator>>failValidationBecause: aString

  | validationFailure exception |
  validationFailure := ValidationFailure new
    object: self object;
    aspect: self aspectForReporting;
    description: aString;
    yourself.
  exception := ValidationFailureException new.
  exception validationFailure: validationFailure.
  exception raise
```

This message will raise exceptions that have the knowledge of what went wrong with validation. They know this because they are holding on to validation failure objects. The corresponding `SUnit` style exception handler will catch these and detect validation failures. But since this exception handler will have access to the exceptions in question, then it also has access to the validation failure objects while the exceptions are being handled. As such, the handler can instruct the validation result object to remember the validation failure objects so they can be used later.

4.5 Using `SUnit` based validation

Now that we can obtain meaningful validation result objects, we need to connect validation with the rest of our application in a non-intrusive way. We will begin by assuming there is no user interface.

4.5.1 Headless validation

We mentioned how convenient it would be to have a message called `validate:` implemented in `AbstractValidator class`. This would give us a simple way to obtain validation results.

In order to implement it, we could simply follow the pattern established by regular `SUnit`: build a test suite, run it, and answer the test result object. So in our case, we will build a validation suite, run it, and answer the validation result object.

```
AbstractValidator class>>validate: anObject

  | validationSuite |
  validationSuite := self validationSuiteFor: anObject.
  ^validationSuite run


AbstractValidator class>>validationSuiteFor: anObject

  ^self buildSuiteFromSelectors
    object: anObject;
    yourself
```

Done, no other work is necessary. We should see this as further confirmation of the degree to which simplicity and expressive power are possible with SUnit based validation. These qualities come from the lack of non essential features. As with Smalltalk, *less is more*.

Alas, we are not done just yet. There is more merciless refactoring to be done. For example, consider invoking validation using one of the expressions shown below.

```
SomeValidatorClass validate: anObject
SomeValidatorClass validate: self
```

Sure, they may be quite intention revealing, but following this pattern means that references to the validator class corresponding to whatever object are going to be sprinkled all over the place. When seeing this, our conditioned reflex should be to refactor into compliance of the reference uniqueness pattern.

So, who should know which validator corresponds to whatever object? The object itself, of course. Therefore,

```
AbstractDomainObject>>defaultValidator

  ^AbstractValidator


AbstractDomainObject>>validate

  ^self defaultValidator validate: self
```

In this way, *each and every* domain object will know how to validate itself in an uniform and predictable way. The particular validation rules that apply for an object are extremely easy to specify: create a new validator, and implement `defaultValidator` in the corresponding domain object.

Again, note the extremely low cost of change, and for how long this cost has remained low since we started to adapt SUnit for our purposes. This is an indication of both flexibility and robustness. It may even start to seem that this way to do validation can take anything we can think of. Good, that is the idea.

Finally, once we have a validation result, we can find out if validation passed or not by sending a message such as `hasValidationFailures`. In the same way, we can find out what are the validation failures by sending `validationFailures`. Once we know all this, there should be nothing we cannot do. Let's verify this assertion by tackling validation in the context of a user interface.

4.5.2 Headful validation

Earlier in the chapter, we briefly went over a typical user interface validation invocation.

```
SomeUIClass>>takeAction

  self domainObject isValid ifFalse:
    [
      self complainBecause: 'Validation failed'.
      ^self
    ].
  self basicTakeAction
```

So how could we rewrite this using `SUnit` based validation? The first thing we need to consider is that since this is a message that is sent as a response to an action, we should minimize the amount of code necessary to invoke validation from a context like this. This is because, clearly, there will be many such action messages. The last thing we need is to duplicate an inefficient expression of intention along with all its deficiencies.

The short version of what we have to do is that we need to add some sort of gatekeeper before `basicTakeAction` is sent. Then this is what we should do, preferably in a single line. The sample implementation below seems like a good way to do it.

```
SomeUIClass>>takeAction

  self domainObjectPassesValidation ifFalse: [^self].
  self basicTakeAction
```

Notice how the task of displaying visual cues was implicitly delegated to a context where this behavior can be implemented once and only once. Let's explore this in more detail.

In order to provide the correct answer, the implementation of the message `domainObjectPassesValidation` will have to examine a validation result. At this point, there are two alterntives. If there are no validation failures, the answer is `true` and then there is nothing else to do. But if there are validation failures, the answer is `false` and then there is some further work that needs to be done before answering because this is a user interface context. In other words,

```
AbstractUIClass>>domainObjectPassesValidation

  | result |
  result := self domainObject validate.
  result hasValidationFailures ifFalse: [^true].
  self reactToValidationFailures: result validationFailures.
  ^false
```

Although there are other approaches worthy of consideration, a simple way to deal with multiple failures is to complain about the first one. Therefore,

```
AbstractUIClass>>reactToValidationFailures: aCollection

  | failure |
  failure := aCollection first.
  Dialog warn: failure descriptionText.
  self highlightWidgetFor: failure aspectForReporting
```

This completes the sample user interface use of SUnit based validation.

4.5.3 Unexpected consequences

As you can see, the implementation of SUnit based validation is extremely simple and straightforward. To my surprise, however, this refactoring exercise led to the whole concept of validation having a much more profound impact than I had envisioned at first.

In 2004, I quite informally described this validation framework to Leandro Caniglia, who had been leading a team of developers working on an oil field simulation application for a number of years. Nothing seemed to happen out of this conversation. Then, in the middle of 2005, and completely out of the blue, he got back to me with quite interesting news. He had achieved great things with SUnit based validation, and wanted to know if I knew who had thought of the approach. I had neglected to tell him it had been me.

He urged and convinced me to write about this new approach to validation. But as I was pondering what would be the best way to describe them, I realized it would be a good opportunity to write about many other things as well. I collected all the material I had and began writing this book. Credit must be given when credit is due. I would like to express my gratitude to Leandro for starting the

process by which it is possible for this page to exist.

Now, without further digression, here are some of the experiences and new uses of `SUnit` based validation that Leandro and his team independently found.

4.6 Extreme validation

The underlying premise in this section is that the process of validating objects is a truly pervasive aspect of application development, and that great benefits become available when its explicit expression is taken to its final consequences. In this regard, the practice of applying validation techniques becomes *extreme*. The following are some of the usage patterns of *extreme validation*.

4.6.1 Pervasive UI validation

At first glance, validators might be seen as a way to factor out repetitive checks that are usually present in the UI. In fact, that was the original motivation for them. Validators turned out to be clearly useful for implementing UI related behavior, for example in the area of context free validation services.

Nevertheless, experience has shown that validators are extremely helpful to identify and solve a wider range of problems. Even when looking at them from a UI perspective, validators have many more interesting and profound applications than data entry quality control. In particular, they provide an alternate solution to the ubiquitous problem of warning the user about invalid or inconsistent states in the system.

Compare this to the situation in which a parent forces a child to guess the right answer, ridiculing mistakes at each wrong attempt. The only "reward" available to the child is an end to the torture. There is no incentive for creative, independent thought.

Under the conventional approach, dialogs unexpectedly jump onto the screen to inform the user about possible or actual problems. This invariably occurs *after* the issue was created in the first place. Because of the modal character of the dialogs, the user must memorize the message, find out where in the application the offending object can be edited, and finally attempt a fix. In other words, *dialogs force the users to make the right decision before they are able to proceed.* It is, evidently, a punitive approach to interaction with people.

With validators, on the other hand, there is no need to force the user to fix a defect immediately. If something is wrong, then the model will know because its validation will fail. `SUnit` based validation allows an application to check the consistency of the model as many times as required. The validation results can be obtained at any moment by simply evaluating the expression below.

```
validationResult := self model validate
```

Once the information the users need is known in the form of a validation result, it is trivial to provide a dedicated place in the UI where the relevant validation details are visible all the time. In this way, the user always has a clear idea about the model's health.

4.6.2 Pervasive abstraction validation

Surprisingly, the main use of validators does not come from the UI but from the model itself. An application that has validators for all its objects can easily accomplish many non-trivial tasks safely.

For instance, validators can detect problems and point out their causes when importing data from external sources. This is best accomplished if the external data is first read into an intermediate object representation before it is added to the application model. A validation on the intermediate representation would identify any invalid or missing components of the data, and prevent any defects from being propagated to the actual domain objects.

Let's take a look at this more concretely. If the intermediate representation is supposed to hold information coming from a spreadsheet, then each of the representations of every spreadsheet cell should have the properties expected from them. In addition, the spreadsheet representation itself should be consistent and be free of any defects such as having missing cells, not having a rectangular shape, and the like.

After the domain objects are created and populated with data coming from the validated intermediate representations, we can validate them as well to make sure the imported data is valid according to the corresponding rules for the business.

This arrangement prevents situations such as having an import batch fail in the middle. If some of the information was added to the intended target database before the failure, it may not be easy, feasible, or even possible to undo the damage. What is left to do now? Fix the problem and import the whole dataset again, hoping it will not crash this time? In this regard, the use of extreme validation practices provides immediate and precise feedback as to the quality of data before it even has a chance to cause problems.

A particularly interesting use of validators in this context arises when it is time to migrate from older to newer versions of the application. After reading

old objects, the application could inform the user about any defects that resulted from the automatic upgrade.

Equally important is the consistency verification that validators provide in concurrent environments, such as those supported by GemStone databases. When the participants of such environment finish a transaction, they get a fresh view of the shared repository. At that moment, a complete validation of the *local* model would detect any inconsistencies caused by the concurrency with other sessions. Once this is known, it is trivial to make the information available to the users.

A model that can be validated at any time can be implemented in terms of simpler, less defensive code as well. For instance, assume that the user wants to delete an object. The conventional approach consists in making sure that the deletion will not break the rules of the model. If it does, then the user will be warned with a dialog. This methodology is defensive and punitive, and as such it will cause inconveniences to the user.

But it is possible to do better than that. If the deletion of object A requires the deletion of object B, then instead of trying to enforce that invariant with warnings and prohibitions, we could check later, at validation time, whether the user broke the rules or not. If the user removed A and not B, then validation will fail. However, if the user ended up removing both objects before clicking on the [`Apply`] button, then validation will pass. A knowledgeable user will not be interrupted, and a misinformed one will be advised.

Validators are useful for developers, too. By adding a [`Validate...`] menu item to the inspectors and to the debugger, the validation of any object in the system will be just one click away. In this way, the health of the objects involved can be examined with just a few mouse gestures. This can almost effortlessly pinpoint the cause of a problem. In particular, developers will become users of the validation framework, and therefore they will have a strong interest to enrich it further with more and more incisive tests.

Typically, when `SUnit` unit tests are written for domain objects, the actual tests fall into two categories. On one hand, some checks are not very demanding and thus apply to every possible domain object. On the other hand, some other tests are stricter and may test non trivial relationships between objects. Quite frequently, it becomes necessary to hand build domain objects for the more strict test cases. In either case, the objects tested by `SUnit` probably do not represent the kind of objects that occur in practice.

With `SUnit` based validation, therefore, testing is taken a significant step forward because every domain object that occurs in practice is subject to being validated. This is why validation is much less forgiving than unit tests, and as

such is a very powerful and valuable tool for developers to have.

Finally, it is clear that tests that check the consistency of the information available to the application are a good thing to have. But where do they provide the most value?

If these verifications were implemented as unit tests, their failure would mean that the production system is probably inconsistent by now. The issue may be so widespread that damage might already be beyond repair. While it is good to find out about potential problems, this information could be a bit too little too late. Not only unit tests do not tell you which objects in the production system may be broken — their failure does not prevent further aggravation from occurring.

But if the same tests ran in the context of validation, failures would keep the production system in a consistent state while providing instant debugging material. This implies that validators are much more valuable than unit tests, especially when their implementation is equivalent. In fact, it even suggests that most unit tests could be implemented in terms of performing validation on a particular set of domain objects.

In fact, note that `SUnit` based validation will be constantly verifying that the code we as developers consider correct actually produces correct results. The moment this does not happen, the failure of the validation process will prevent the damage from spreading from a contained set of domain objects to the entire database. This is particularly true because with extreme validation, the idea is that objects are validated in a variety of contexts, not just after the user enters new information.

4.6.3 Pervasive source code validation

One of the problems addressed by `SUnit` is that of incorporating code from several developers into the official version of the software. The idea is that only code that passes all tests is acceptable. With validators, it is possible to extend that premise by declaring that only validated methods and classes are acceptable. The key concept is that of validating the source code itself[6].

Let's take a look at source code validators in practice. For example, we could search for occurrences of returns in exception handlers, such as the one below.

[6]Blaine Buxton has independently researched this subject as well. For more information, links to books, articles and further material, visit his webpage on "Automated Standards Enforcement" at http://www.blainebuxton.com/ase.

```
[self doSomething]
  on: Error
  do: [:ex | ^nil]
```

These returns can cause issues that can be quite hard to debug. This is because the return could be performed in the process handling the exception, not in the process in which the exception occurs[7]. The corresponding validator methods, shown below, can help making sure code patterns like the one above do not reach the production system.

```
validateOnError

  | sendsOfOnDo |
  sendsOfOnDo := self parseTree messageSends select:
    [:any | any selectorNode value = #on:do:].
  sendsOfOnDo do:
    [:eachSend |
      | exceptionHandler returnsFromHandler |
      exceptionHandler := eachSend arguments second.
      returnsFromHandler := exceptionHandler allNodes select:
        [:any | any isReturn].
      returnsFromHandler notEmpty ifTrue:
        [
          self failValidationBecause:
            self shouldNotReturnFromExceptionHandler
        ]
    ]


shouldNotReturnFromExceptionHandler

  ^self prettyPrint, ' returns from an exception handler.'
```

Source code validators are easy to implement because of the reflection capabilities built in Smalltalk. As you can imagine, the example above is just the tip of the iceberg. For example, by examining the parse tree, it is also possible to detect

[7]The particular misbehavior depends on the Smalltalk being used, and on the particular implementation of their exception framework. Some resist this kind of abuse better than others.

when a method `super`-sends a message other than the one being executed. In other words,

```
initialize

  super differentInitialize.
  self localInitialize
```

The corresponding validator methods are shown below.

```
validateSuperSend

  invalidSuperSends := self parseTree messageSends select:
    [:any |
      any receiver isSuper and:
        [any selector ~= self selector]
    ].
  invalidSuperSends notEmpty ifTrue:
    [
      self failValidationBecause:
        self shouldNotSuperSendDifferentMessage
    ]


shouldNotSuperSendDifferentMessage

  ^self prettyPrint, ' super sends a different message.'
```

Other code patterns that can be easily detected by validators include:

- Files opened in one method and closed in another one.
- Methods that use magic constants.
- Domain objects that open dialogs.
- Objects that refine the implementation of the message `=`, but still inherit the implementation of `hash` from `Object`.

By extension, one could add specialized validators for the source code of the user interface. For instance, it is possible to validate that all events triggered by the UI are handled by the UI and not by its model.

More profoundly, specialized validators could check that some piece of code conforms to a given programming or design pattern. This higher level source code verification would be a particularly interesting application of validators. Despite the fact that design patterns have become very popular as theoretical instruments and as terms of discussion, they are rarely honored in actual implementations. An appropriate validation suite could ensure that at least the fundamental principles of a given pattern are present in the code.

4.6.4 Collapsed test hierarchies

Typically, validators check each of the instance names of the object in a one by one fashion. These checks usually make sure that the contents of the object in question are defined, within an acceptable range of values, not empty, and so on. In addition, a validator can also verify the relationships, if any, between two or more of these instance names. Notwithstanding, the main flow of the validation implementation is firmly dictated by the structure of the object being validated. This must be considered carefully, as it could negatively impact the structure of the validation hierarchy.

For the sake of illustration, let's assume a simple domain object hierarchy with a root class and some subclasses, such as the one below (instance names shown in parentheses).

```
RootClass (a b)
  SubclassOne (x)
  SubclassTwo (y)
    RefinedSubclassTwo (z)
```

A naïve validation implementation would result in a validator hierarchy that duplicates the one of the domain objects.

```
RootClassValidator
  SubclassOneValidator
  SubclassTwoValidator
    RefinedSubclassTwoValidator
```

In turn, since each of these validator classes would correspond to a particular domain object class, the validators would implement validation messages such as the following.

```
RootClassValidator>>validateA, validateB
  SubclassOneValidator>>validateX
  SubclassTwoValidator>>validateY
    RefinedSubclassTwoValidator>>validateZ
```

This would happen even when the subclasses of `RootClass` are very similar to their superclass. In well designed class hierarchies, that would frequently be the case. Yet, as the same pattern repeats itself, it duplicates the maintenance cost of the domain objects. A domain object class hierarchy change would be mimicked by the validator class hierarchy as well. This duplication of work appears to be a shortcoming. Thus, one arrives at the conclusion that the validation hierarchy should not necessarily reproduce that of the domain objects, especially when many classes are involved. A better design must overcome the inability of scale of the naïve approach.

The issue seems to be that we have too many classes. While sometimes it is good to create many classes to reproduce our understanding of a problem in fine resolution, creating a parallel hierarchy of validators would not use the fact that the objects being validated are very similar to each other. This seems to be a case in which using coarser resolution validation classes appears to be better. The fact that it seems possible to do it without losing expressiveness makes the approach worth considering.

So, if we wanted less validator classes, we could refactor a hierarchy of closely related validators so that their implementation is collapsed into the hierarchy's root validator class. In our example, we would push all the validation methods up until they are implemented in `RootClassValidator`.

Now, of course, this makes sense for our particular example because we have just a handful of classes. In general, the idea is to find particular branches of the class hierarchy that allow this validator class collapse without making the resulting validator a conglomerate of extremely dissimilar validation rules.

This refactoring blurs the distinctions that allow us to tell which validation applies to which object, since the same validator would now apply to many different domain objects. In order to determine which validation sequence to run for each particular object, thus restoring the distinctions lost in the collapse of the validator class hierarchy, we could use the visitor pattern to let objects being validated have a say into what set of validation rules apply specifically to them.

Note that this validation customization would come in the form of messages that specify the list of validation rules for particular domain objects, and that such messages would be implemented in `RootClassValidator`. In this way, the

domain objects do not need to know the individual validation rules that apply to them, thus shielding them from validator implementation details.

This technique requires careful thought and consideration, as it also comes with an associated maintenance cost. With numerous domain object classes and a diverse class hierarchy, however, there will be a point in which collapsed validator hierarchies are easier to maintain than parallel validator class hierarchies.

How would this approach work in the case of our particular example? First, we would have all the validation messages that apply for the domain objects implemented in the usual manner in `RootClassValidator`. To allow for the differences between the root domain object class and its subclasses, we could implement an additional validation message to kick off the visitor pattern:

```
AbstractValidator>>validateSpecialized

  self object validateInTheContextOf: self
```

Then, each domain class would refine the message `validateInTheContextOf:` following the pattern below.

```
Object>>validateInTheContextOf: aValidatorClass

  ^self


SubclassOne>>validateInTheContextOf: aValidatorClass

  super validateInTheContextOf: aValidatorClass.
  aValidatorClass subclassOneValidation


SubclassTwo>>validateInTheContextOf: aValidatorClass

  super validateInTheContextOf: aValidatorClass.
  aValidatorClass subclassTwoValidation


RefinedSubclassTwo>>validateInTheContextOf: aValidatorClass

  super validateInTheContextOf: aValidatorClass.
  aValidatorClass refinedSubclassTwoValidation
```

Finally, we would need to implement the messages that specify the particular validation rules that apply for each domain object. We would implement these in `RootClassValidator`.

```
RootClassValidator>>subclassOneValidation

  self xValidation


RootClassValidator>>subclassTwoValidation

  self yValidation


RootClassValidator>>refinedSubclassTwoValidation

  self zValidation
```

With this design, we only need one validator class per main model hierarchy. The validation of `RootClass` subclasses requires more methods instead of additional validator classes.

Note that the actual specialized validation selectors such as `xValidate` must not begin with the selector prefix `validate`, as they should not be included in the validation suite of the superclasses. These messages will be automatically sent by the double dispatching mechanism kicked off by `validateSpecialized` as discussed above.

The natural grouping of the model in main class hierarchies should provide enough clear feedback to make distinguishing the required collapsed validator classes easy. When the model hierarchy is well designed, the double dispatching simply invokes the validation of the subinstances, avoiding the duplication of the model hierarchy.

4.6.5 Validation delegation

Domain objects typically have a composite structure because they hold on to other domain objects as well. Therefore, it is natural for validators to delegate the validation of each composite part to the parts themselves. For example, if an object has an aspect called `user`, then the object validator would typically try to include the validation of the value associated with that aspect:

```
SomeValidator>>validateUser

  self aspect: #user.
  self valueIsDefined.
  "actual validation of the user object"
```

Since this pattern appears frequently, the mechanism that delegates validation to other objects should be implemented at the framework level to avoid code repetition. This is very easy to do because validation failures are real exceptions, and because exception handling is already being done by `SUnit`. Therefore, we can just do as follows.

```
AbstractValidator>>delegateValueValidation

  self adoptValidationResult: self value validate


AbstractValidator>>adoptValidationResult: aValidationResult

  | exception |
  exception := ValidationResultAdoption new.
  exception validationResults: aValidationResult.
  exception raise
```

By implementing a specific handler for `ValidationResultAdoption` exceptions, so that it just adds passes, failures, validation failures and errors to the main validation results, it becomes easy to adopt the product of nested validators.

Now we can go back to our `validateUser` message and use the framework functionality we just added to rewrite it like this.

```
SomeValidator>>validateUser

  self aspect: #user.
  self valueIsDefined.
  self delegateValueValidation
```

However, note that for validation delegation to work, the `user` aspect must be defined. This `nil` check will have to be performed every time we mean to delegate

validation, since our intention is to delegate validation to a defined domain object. Therefore, we can move the `nil` check to the framework as follows.

```
AbstractValidator>>delegateValueValidation

  self valueIsDefined.
  self adoptValidationResult: self value
```

After these changes, we can implement `validateUser` in very simple terms.

```
SomeValidator>>validateUser

  self aspect: #user.
  self delegateValueValidation
```

This delegation mechanism ties all validators together, greatly simplifying the implementation. Note, however, that this approach leaves the door open to uncontrolled validation recursion. In order to break any circularity in the validation execution, delegation must stop at final nodes or leaf aspects.

There are two kinds of final nodes. The first one is that of aspects that are numbers, strings, characters and the like. In that case, instead of sending the message `delegateValueValidation`, the specific validation rules should be invoked. This is when we would make use of all the validation helper messages provided by the framework: `valueIsDefined`, `valueIsPositive`, etc.

The second kind of final nodes are those that have already been validated. For example, assume we have three objects: `objectA`, `objectB` and `objectC`. We would have a potential validation cycle if `objectB` was an aspect of `objectA`, `objectC` was an aspect of `objectB`, and `objectA` was an aspect of `objectC`. In other words,

$$\ldots \texttt{objectA} \rightarrow \texttt{objectB} \rightarrow \texttt{objectC} \rightarrow \texttt{objectA} \ldots$$

In this case, the object validators should take care of that circularity.

```
objectA validatorClass>>validateAspectB

  self aspect: #aspectB.
  self delegateValueValidation
```

```
objectB validatorClass>>validateAspectC

  self aspect: #aspectC.
  self delegateValueValidation


objectC validatorClass>>validateAspectA

  self aspect: #aspectA.
  self valueIsDefined
```

Note that in the third validator we restricted ourselves to the `notNil` test. This prevents the infinite recursion.

As circularities are model specific, it is usually simpler to make them explicit in the validators than trying to resolve them automatically. However, depending on your application, it may be worth it to implement a transitive closure checker.

4.6.6 Pervasive validation services

When implementing the user interface of an application, we frequently find places where it is necessary to validate the name of an object provided by the user. The most common validation rules include checks to make sure the name is not blank, and sometimes, that it does not contain invalid characters. Validators are natural providers of this kind of services.

There are several ways in which validators can help to maintain the user interface implementation simple and well factored. First, we can implement a default validation service in the top of the validation service hierarchy:

```
AbstractValidator>>valueIsAcceptableName: aString

  (aString isNil or: [aString trimBlanks isEmpty]) ifTrue:
    [self failValidationBecause: 'The name cannot be blank.'].
  aString trimBlanks ~= aString ifTrue:
    [
      self failValidationBecause:
        'The name cannot start or end with spaces.'
    ]
```

Furthermore, specific subclasses can refine the default validation by adding more checks.

```
FileDialogValidator>>valueIsAcceptableName: aString

  | hasInvalidCharacters |
  super valueIsAcceptableName: aString.
  hasInvalidCharacters := aString anySatisfy:
    [:any | #($: $" $* $/ $\ $? $[ $]) includes: any].
  hasInvalidCharacters ifFalse: [^self].
  self failValidationBecause:
    'The name cannot contain :, ", *, /, \, ?, [ or ].'
```

In this way, even a file dialog can make use of the validation services provided by `SUnit` based validation. More specialized name validation services can also take care of name collisions by comparing the candidate string with all other names used so far.

4.6.7 Validation performance

A system that makes heavy use of validations ends up validating everything. This is healthy and has many benefits. Not only validation tests are very easy to write — as they check aspects for the most part, their complexity is typically $O(n)$, where n is the number of instance names.

Even though individual validators are fast, care must be taken not to ruin the responsiveness of the application when thousands of objects have to be validated. Here are some tips on how to address performance, should it become an issue.

1. Use `beginsWith:` instead of `match:` to find the validation selectors that should be used by the validation suite.

2. In `AbstractValidator>>allTestSelectors`, avoid going up to `Object` looking for selectors that are never above the level of `AbstractValidator`.

3. Write `SUnit` tests to make sure no object is validated more than once.

4. Do not run large validations if nothing has changed.

5. When validating large amounts of objects, provide visual feedback to the user.

Items 1 and 2 above can significantly improve the performance of validations. Since validation suites are built dynamically for each object, it is critical to improve the scanning of selectors as much as possible. In a typical scenario we have the following class hierarchy.

```
Object
  TestCase
    AbstractValidator
      DomainObjectValidator
        SomeSpecificValidator
```

A quick check in VisualWorks yields an approximate answer of 350.

How many messages are implemented in `AbstractValidator` or in any of its superclasses? Certainly it is not just a few. Regardless of the amount, it does not make any sense to search for validator selectors at `AbstractValidator` or above, since we already know that actual validations start at `DomainObjectValidator`. This can be a huge win since the selector search is performed when each object has to be validated. Once again, there is no need for the run time to learn multiple times what is already known at design time.

An application can also find it useful to trigger an event each time an object is about to be validated. This feature is very easy to implement.

```
AbstractValidator>>performTest

  self object triggerEvent: #aboutToValidate.
  super performTest
```

With this feature, the `SUnit` test for item 3 above becomes trivial. The same mechanism can be used to accomplish item 5 above: each time an object is about to be validated, some progress indicator registered to the `#aboutToValidate` events can be updated.

4.7 Exercises

Exercise 4.1 [12] Implement `CharacterArray>>prettyPrint`.

Exercise 4.2 [24] Refactor `runCase:` and eliminate nested exception handlers.

Exercise 4.3 [09] Refactor `runCase:` further and eliminate any occurrence of `ifTrue:ifFalse:`.

Exercise 4.4 [18] Refactor `runCase:` again and remove the code that handles test case passes altogether.

Exercise 4.5 [25] Make sure that you can still debug tests after you solve the previous exercise. Also, refactor `runCase:` one more time to eliminate references to exception classes.

Exercise 4.6 [15] Use the unique reference technique and refactor `SUnit` so that classes are referenced once and only once. Then, use these improvements to let validation versions of `SUnit` classes change the inherited behavior by refining just the message with the unique class reference.

Exercise 4.7 [29] Take a look at the piece of code below.

```
SomeTestCase>>testExceptionHandling

  self halt
```

How do you explain the behavior of this test method in the test runner?

Exercise 4.8 [10] Refine `SUnit` so that it is possible to hide validators in the test runner.

Exercise 4.9 [07] Prevent the execution of any validation from the test runner.

Exercise 4.10 [39] Implement the `SUnit` validation framework and migrate all the existing validation code in your project.

Exercise 4.11 [09] Is it possible to implement `runCase:` in `ValidationResult` in terms of `super`?

Exercise 4.12 [16] Sometimes it is necessary to validate that a change is correct in the context of the original value. Extend validation to handle these cases.

Exercise 4.13 [10] The runtime packager will discard the validation messages because they have no explicit senders. Fix this.

Exercise 4.14 [15] Extend `SUnit`-based validation so that it is easy to validate values held inside value holders.

Exercise 4.15 [18] Refactor the validation helper methods to eliminate sends of `isKindOf:`.

Exercise 4.16 [24] Extend `SUnit` based validation so that it can be used to provide suggestions.

Exercise 4.17 [21] Extend `SUnit` based validation so that it can be used to provide warnings.

Exercise 4.18 [21] Extend `SUnit` based validation so that it can be used to ask for confirmations.

Exercise 4.19 [28] Refactor `SUnit` so that the execution of each test case is modeled with an individual result object.

Exercise 4.20 [18] While there may be situations in which adding the instance name `aspectForReporting` to validators makes sense, it was introduced to cope with a user interface deficiency. In other words, the category is being changed via descriptions, and the user interface itself seems unaware of this. Clearly, this is not a validation problem. Address this deficiency.

Exercise 4.21 [14] Sometimes, it may be convenient to perform validation on user interface objects. This arises in the case of dialogs which do not necessarily have a domain object behind them. Extend `SUnit` based validation to perform validation on user interfaces.

Exercise 4.22 [38] Redesign your application's UI interaction with validation so that it is not punitive.

Exercise 4.23 [24] Assume there are several different dialogs that need similar validation services. Unfortunately, since you used intention revealing names, the dialogs' instance names are different and thus a single validator cannot be reused. Extend `SUnit` based validation so that the result of executing a single validation message can be obtained.

Exercise 4.24 [49] Implement source code validators that pass or fail depending on whether they are able to detect the presence of a design pattern in source code.

Exercise 4.25 [31] If the message `validateInTheContextOf:` invokes several validation messages, it may not execute all of them. This occurs because the first failure will be caught by the `SUnit` exception handler in `runCase:`, and thus `validateInTheContextOf:` will be terminated before it has a chance to finish. Address this deficiency.

Exercise 4.26 [40] *(Hernán Wilkinson)* Instead of having a validator class per kind of object, implement a validator class per kind of validation in a hierarchy such as shown below.

```
AbstractTestCase
  AbstractValidator
    ValueIsDefinedValidator
    ValueIsNonEmptyStringValidator
    ValueIsTimestampValidator
    .
    .
    .
    CompositeValidator
```

Then, implement a class side method for the domain object that needs validation following the pattern shown below.

```
DomainObject class>>validator

  ^CompositeValidator new
    add: (ValueIsDefinedValidator forAspect: #someAspect);
    add: (ValueIsDefinedValidator forAspect: #someOtherAspect);
    .
    .
    .
    yourself
```

This arrangement would not cause a hierarchy of validators that mimick that of domain objects. But, is this arrangement better overall?

Exercise 4.27 [45] Explore adapting SmallLint and the Code Critic to `SUnit` based validation.

Exercise 4.28 [36] When measuring the performance of a particular piece of code, it is beneficial to stress the implementation with several different test cases.

This is typically done in a workspace. Unfortunately, when there are several competing implementations, the task of keeping track of all the times goes from "barely manageable" to "a complete mess".

Create an extension of `SUnit` designed to benchmark different approaches across multiple test scenarios.

Chapter 5

An efficient reference finder

5.1 Preliminaries

The previous chapter illustrates how much you can obtain from refactoring when your working hypotheses apply to every occurrence of a particular pattern. This is because it becomes possible to arrange things such that there are no special cases that get in the way of clear expression of intention. In other words, it is a matter of choosing your point of view carefully.

And indeed, your point of view determines what things you will see and what things you will ignore. Your point of view is implied by your intentions because it determines what distinctions you perceive.

We should exercise our skills to choose points of view artfully. And since we have discussed information spaces and traversals quite a bit, it is time to put a reflexive twist on them. Let's consider the traversal of the information space implied by the Smalltalk image by writing a reference finder.

5.2 The space to traverse

Since we will be considering the traversal of a space, the first thing we should consider is the shape of the space in question. The Smalltalk image can be seen in many ways, but as we are interested in references, we could take a look at it from a garbage collection point of view.

With these intentions, the image "starts" at some globally visible object that is considered to be not garbage by convention. Typically, this is the `Smalltalk` dictionary. Anything that is directly or indirectly referenced by this non garbage

context is said to be not garbage either.

And what is a reference? If we look at objects from a Smalltalk point of view, references occur via regular instance names. For example, associations reference the values of their `key` and `value` instance names. In addition, references also occur by means of index based storage. For instance, arrays reference whatever is stored in any of their integer named slots.

These are the only ways in which an object can reference another object. This is because things like class names and pool dictionaries are implemented in terms of instance names or integer named slots.

So it seems we could think of the Smalltalk image as a tree starting at `Smalltalk`. But since the leaves could easily reference their parents or even `Smalltalk` itself, it does not quite qualify as a tree. In reality, it is a graph where nodes are objects and references are represented by arrows connecting the nodes.

Interestingly, the node for `Smalltalk` is not special by itself. We just start traversing the graph from this node only because of our convention to never consider it garbage[1].

Also, note how the arrows only have one direction: no object has immediate knowledge of where it is referenced from. This is why tools like a reference finder become necessary so that we can derive this knowledge from the graph.

The truly interesting references to an object are the ones that occur closest to `Smalltalk`. Since these are the most natural reference paths, we should build a breadth first reference finder — one that works as some sort of onion peeler in reverse.

Unfortunately, the nature of the space and the implementation techniques available make it too easy to implement depth first reference finders instead. For example, we could let an object iterate over instance names and indexable slots. However, this forces the static context to remember which ones it chose to dive through, and things typically get quite complicated soon thereafter. As if managing that iteration was not difficult enough already!

We could also try to take advantage of an existing primitive that answers where a given object is directly referenced from. But, alas, the answer of this primitive leaves us with figuring out a breadth first path from `Smalltalk` to the object in question, while the information we are provided happens to be oriented in a direction opposite to the one we want. Making sense out of the result is typically left as an exercise for the developer. Sorting through the mess of false

[1]To be more precise, we are interest in finding out all the objects that are connected to `Smalltalk`. The fact that we are going to say they are not garbage does not matter as far as the connectedness check itself is concerned.

positives, even when done mechanically, takes a lot of time and effort.

To summarize, while these techniques make it very easy to implement a depth first reference finder, the problem is that depth first traversal has the irritating behavior of producing extremely long and convoluted reference paths. What we need is a breadth first reference finder, and these tools give us no easy way to implement one. It looks like this is going nowhere.

But there is another way to take a look at objects and their mechanisms for referencing other objects. What if we saw the reference graph in terms of forms and distinctions instead?

5.2.1 Distinctions, forms, and objects

We have said numerous times that objects model distinctions. However, we have not examined this issue rigurously until now. It is time we take care of the loose ends so we can rely on a solid foundation for our traversals.

For example, distinctions are drawn in a form. We have assumed we always have a form where to create our objects, but what is that form exactly? And why can we assume it is always there so lightly?

It turns out that the form is the object memory, and since Smalltalk provides automatic memory management, we do not have to worry about it. Stop for a moment and consider the implications. Drawing distinctions is what we do when we write Smalltalk programs. Drawing distinctions is what programs do for us. Automatic memory management means that the most important and elementary operations that occur in the process of reaching our objectives are tax free. In other words, Smalltalk not only keeps out of our way when we try to get things accomplished — it actually spends quite a bit of effort to let us achieve our goals without distractions[2].

Another issue that merits careful consideration is that when distinctions are drawn, the form on which they are inscribed is severed into two perfectly disjoint areas. These two are forms as well. That means that drawing a distinction creates a new form inside of it, and therefore objects contain their forms. What about

[2]Nobody should have to put up with mandatory hindrances when using our most critical abilities: to perceive differences in value, to draw distinctions, to name those distinctions, and to allow messages to cross distinctions. Any level of inefficiency, obscurity, or unnecessary complication regarding these matters comes with extremely expensive consequences. If we allow any of these issues to creep up on us, we will also give up to their stealthy yet powerful influences. There is no limit to how much they can pervasively limit us, while at the same time letting us believe we are accomplishing something valuable. This delusion is absolutely inexcusable and unacceptable, and must be fought back at all costs.

their contents? In the same way we draw distinctions on a form, in Smalltalk we reference other objects by means of instance names and indexed slots. Since the internal structure of objects is defined by their class, this means that the shape of object forms is both predetermined and fixed[3].

So what are objects, distinctions or forms? Well... actually, objects exhibit dual behavior. Seen from the outside, they behave like a distinction because they separate their insides from the outside. We typically mean this when we say that objects provide strong encapsulation for their "contents" — it is impossible to access the instance names of an object[4]. However, when seen from within, objects are forms. When the point of view resides inside of them, their contents are immediately accessible without the need to send a message. In this way, objects are both distinctions and forms at the same time. At any time, we can change the point of view by sending a message to an object and crossing the distinction boundary.

Something we have not discussed yet is the fact that the value of names can change in Smalltalk. As far as tracking down references to an object goes, however, we can assume that the values of names do not change. Therefore, we will leave the transmission of values across networks of distinctions for later.

5.2.2 Contexts of immediate name accessibility

Contrast this network of objects and messages crossing distinctions with the best explanations of how we think. As discussed earlier, we have this sort of scratch RAM that has 7 ± 2 available slots. Each time this storage space overflows, our ability to resolve distinctions is severely impaired. In order to function efficiently, we need to chunk used slots together to make space in the scratch RAM. What is going on?

I would like to humbly put forward the following idealization. Our scratch RAM is our context of immediate name accessibility. It is in this context that we can perceive differences in value, draw distinctions, and send messages to cross boundaries. When we need to make space in our scratch RAM, our technique is to carve separate contexts of immediate name accessibility by clustering names

[3]Even things like ordered collections are implemented in terms of fixed size objects — either arrays or themselves.

[4]We cannot even say that the contents of an object are accessed by sending messages. Since the implementation to the messages is private to the object, senders cannot assume that sending an accessor will actually answer the real contents of the object. While sometimes it is a very useful assumption to make, it is not guaranteed to be true 100% of the time.

into chunks. In Smalltalk, we describe these chunks in terms of objects.

Sending a message to an object changes the current context of immediate name accessibility. This represents our mind crossing into and out of chunks, swapping the contents of our scratch RAM in the process. Note how message arguments and answers allow distinctions to stay in the scratch RAM across multiple context crossings.

So, if we could train ourselves to perceive our thought processes in these terms with enough clarity, then it would not be hard to translate them directly into objects and messages. Mastering this technique would allow us to fluently use it to the service of communication. The hard part would be how to design the objects and the messages such that the result has all the desired emergent properties: intention revealing, no syntactic sugar, proper carving of contexts of immediate name accessibility, efficient due to lack of `ifTrue:ifFalse:`.

If we get to the point where translation is easy and design is hard, then it follows that we should refine the design until we cannot improve it further, and then simply translate it. Certainly, wasting our energies on translation will not get us anywhere: in order to get stuff done, we need an idea of what to implement in the first place! Therefore, we should spend most of our time and efforts in the hardest problem first — the design of our solution to the problem at hand. This is where our work can make the largest impact. When deadlines are considered to be a design limiting factor too, this approach can maximize the ratio of value over time quite aggressively.

5.3 The Smalltalk space

We have discussed forms and distinctions for quite a while, but we have not yet implemented them in Smalltalk. This is a good opportunity to do so, as it will allow us to represent the parts of the information space we want to traverse in our own terms.

5.3.1 Laws of Form objects

Let's start by creating the class `Form`. As we said earlier, it represents the piece of paper or canvas where we draw distinctions. A number of messages come to mind, such as `size` and `isEmpty`. But while the implementation of collection like behavior is quite straightforward, the question remains: what kind of collection does `Form` resemble the most? To answer that question, we need to examine its interaction with distinctions.

If we consider the circles on a piece of paper metaphor for distinctions, we could think of them in terms of these things that divide a space into two regions. We could also see them as these portals that have the capability of taking us from one space to another. As such, distinctions are the links connecting two different forms.

Just like in Croquet.

Because of this, distinctions could be implemented as some sort of association. In particular, distinctions can be named, and hence they should also remember their name.

The first axiom of the laws of form says that calling a name twice is the same as calling a name once. Therefore, a natural first approach would be to model forms in terms of a set of named distinctions.

Watch out for the desire to optimize prematurely at this point.

Now it is time to choose instance names for our objects. We will start with class `Form`. What could we ask `aForm` to obtain a collection of the distinctions it contains? Well, the message `distinctions` sounds quite straightforward, and as such we will use it.

The names for distinctions require much more care. Distinctions are drawn on a form, and this is their form — in other words, the form of a distinction is where it is drawn. Therefore, `form` should answer the form where the distinction lives. Since `form` is now taken, what other selector could we use to access the form inside the distinction?

In these situations, it is good to review the language that is used to talk about distinctions. Let's see... "distinctions separate a space into two disjoint pieces", "distinctions are the boundaries between such pieces", "distinction is perfect continence". It seems like there is a theme of enclosure, or even containment. The selector `contents` suggests itself. However, it is not clear that it is a good choice. This is because the contents of a distinction can only be accessed by crossing the distinction. As such, `contents` is intention obscuring because it seems to indicate it is fine to know about the contents of a distinction without crossing it — `contents` does not have strong encapsulation connotations. But not all is lost since we did mention the act of crossing distinctions. Thus, we will choose the verb instead of the noun, and send the message `cross` instead. If we let it be an accessor, then we will have an intention revealing instance name too.

To summarize, we have the following classes and instance names so far.

```
Object
  Form (distinctions)
  Distinction (form cross name)
```

Now we should design how these objects interact together by means of messages. Clearly, distinctions need a form where to live. This indicates that instances of `Form` will be created first, and that forms will create instances of `Distinction`. What would be a good process for these distinctions to be created and crossed into? What would *we* do if we wanted to distinguish something in a form?

At this point, very interesting questions appear. For us, intention distinguishes between different values, and the way in which we care about the differences determines the kind of distinctions we draw. And we just take this for granted. To illustrate the point, every so often I try to look at something as a blob, without further interpretation. In my experience, it is almost impossible to do with letters. The association between the glyph and its meaning happens too quickly for it to be stopped. Drawing distinctions is so automatic for us that it is almost invisible.

When we try to reproduce this act in Smalltalk, we run into further difficulty. For example, what if we told a form to distinguish something? Certainly, we would tell it what to distinguish by means of an argument to a message. But in Smalltalk everything is an object, and therefore distinguishing something in a form object requires that the very something we are trying to distinguish must exist beforehand, so we can pass it as an argument. What is going on?

The issues are that objects are distinctions that live in the image, and that the Smalltalk image is a form. In other words, form and distinction objects are redundant because they reproduce what Smalltalk provides for free. Then why should we try to copy what is already given? Because reflecting on these matters can be quite valuable. Redundancy can work on our favor here, as it will make explicit something we usually take for granted. And when this happens, we will be able to change what we take for granted and create new things.

So we have our form, and an already existing distinction (in some other unspecified form) that we want to distinguish in the new form. In Smalltalk, it would be natural to take the object and put it in the new form. This new distinction would then be identical to the original one, and thus we could think of this re-distinction as some sort of copy or act of reference. But then, as per Laws of Form, if we take a distinction existing in some form, and copy it into another form, then we can refer to the copy as the *name* for the original distinction. And this is where things become interesting[5].

What is being copied is the pointer to the object header, but fortunately we do not have to deal with that.

First of all, since the copy is a name, and because we are in the Smalltalk

[5]Interesting indeed: you could think of the form in your mind being the unspecified form, and the Smalltalk image being the form in which you copy distinctions already present in the unspecified form — oh.

image the copy is identical to the original, then each object we distinguish in the new form is both the object and its own name. In this way, Smalltalk objects have this very peculiar dual property: they are their own name.

And when you cannot name them, they are garbage.

But then, if we do not need to care about distinction names explicitly, we can simply distinguish objects in our new form and let them stand for their own names. Our original design for form, a set of named distinctions, now looks very promising. Therefore we implement the following message.

```
Distinction>>= aDistinction

    ^self name = aDistinction name
```

In this way, forms would be the equivalent of sets. Of course, if necessary, we could easily implement `IdentityForm` and `IdentityDistinction`, but we will leave those for when we actually need them.

So all this is great, but... hey, aren't we forgetting something? Well, yes — we implemented `=` but we did not implement `hash`!

Baaad programmer! Slap! Slap!

```
Distinction>>hash

    ^self name hash
```

We should not forget to implement `hash` when we refine `=`, as failure to do so can lead to issues that are quite hard to debug.

Baaad doggie! Don't chew on the power co@Σ?# +$∼!&Γ: — NO TERRIER*

Back to forms and distinctions now. At last, we can start implementing the messages that will allow us to give shape to our silly putty. Since we cannot do much with empty forms, let's start with the message below.

In case you are wondering, the DOS character set includes (among others) the greek letters shown above. In particular, the index of Σ is 228, and that of Γ is 226.

```
Form>>distinguish: aName

    ^(self distinctionNamed: aName) ifNil:
      [
        | distinction |
        distinction := Dictinction named: aName.
        distinction form: self.
        self distinctions add: distinction.
        distinction
      ]
```

We can enforce the *only one distinction per name* restriction because the first axiom of Laws of Form says that invoking the same name twice is the same as invoking it once. In other words, we are not altering the meaning of the form structure by doing so. To that effect, pay close attention at how `ifNil:` is not being used as a control mechanism, but rather as a means of replacement. Indeed, `ifNil:` should be read as meaning `butIfNil:` instead.

So how is `distinctionNamed:` implemented?

```
Form>>distinctionNamed: aName

  ^self distinctions
    detect: [:any | any name = aName]
    ifNone: [nil]
```

You may be tempted to use a dictionary here, but keep that temptation at bay for now — the exercises have the details.

Once we have these messages, we should be able to cross into another form via a distinction. To do that, we simply implement the following message.

```
Form>>cross: aName

  ^(self distinguish: aName) cross
```

Note how this can create a distinction the first time, and reuse it thereafter. This is consistent with the Laws of Form axiom that says that using a name twice is the same as using it once. Also, note how the distinction answers its contents form when answering the message `cross`. This emphasizes how important proper names can be.

Finally, there should be a way to undistinguish something. And what would a good selector for that be? Certainly, `undistinguish:` seems a bit coarse. The selector `forget:` is also misleading. We need to undo a distinction, and that is not the same as forgetting it.

Perhaps we could find a selector by looking at the state of things before the distinction occurred. Before the distinction was drawn, there were no differences in value.

Hmmm... perhaps we are getting somewhere. Let's continue finding alternate ways to say the same thing.

Both sides of the distinction had the same value to us. We could use them interchangeably. They were equivalent to us. We did not need to care about their

differences.

Well, all of these are positive assertions and we do not seem to be making much headway. How about some negative ones?

The technique is almost automatic, to the point of seeming to have very little merit. However, it relies heavily on how evolution has worked for so much time. If something does not quite work, find a peculiar characteristic of it and apply some change. In other words, purposefully introduce mutations in how the traversal of the space is done, and evaluate the outcome. Do the random walk.

We were looking at things with a coarser resolution. Both sides fell in the same bucket. We would not care if we picked one or the other. We would confuse one for the other.

Hold it right there! That is it!

```
Form>>confuse: aName

  | existingDistinction |
  existingDistinction := self distinctionNamed: aName.
  existingDistinction isNil ifTrue: [^self].
  self distinctions remove: existingDistinction
```

Now that we have our basic Laws of Form objects in place, let's see how this relates to distinctions existing in the Smalltalk form — I mean image.

5.3.2 Smalltalk objects

Now that we have this structure of forms connected to each other by two sided links we call distinctions, how does this relate to Smalltalk objects? Are objects more like a form, or more like a distinction? And in particular, what is the approach that best fits a reference finder?

On one hand, objects are distinctions in that they impose a boundary that forces us to send messages in order to interact with them. But, at the same time, the object contains its local names such as instance names. So, in as much as the object is a distinction, it also contains a form where all such local names exist.

Again, we find that objects exhibit dual behavior. However, we could try to model actual objects in terms of forms and/or distinctions. Which approach is best?

If we decided to model objects in terms of a distinction, then we would want to cross into it every time we needed to traverse its contents. In part, we take that for granted when we send messages, because Smalltalk is already providing for the automatic crossing of the object's boundary. Thinking of an object as a distinction, thus, does not seem very convincing for our purposes.

So perhaps we should think of an object as a form with an implicit distinction around it, something like a read only form. We do not need write capabilities

to travel from object to object anyway. Let's try this out, and make a subclass of `Form` called `ObjectForm`. Since it is not meant to allow object modification, messages like `distinguish:` and `confuse:` can easily be implemented in terms of `self shouldNotImplement`.

But what about messages like `cross:`? How are we going to allow access to the object's distinctions? What would the instance name `distinctions` hold in this case? What are the names of the distinctions inside an object's form, from this point of view?

Deep down, after the compiler does its work, objects store references to other objects in numbered slots. The ability to provide nice instance names is given to us as a convenience, however, they are really just fancy labels for each position of an array. Now, as far as traversing this structure of objects, we do not really need to know the nice names for these slots. All we need are distinct labels so we can distinctly reference each of the distinctions inside the object. Since these slots are numbered already, we will use integer labels.

In Smalltalk, objects have two kinds of these numbered slots. First we have the indexable slots, with which things like `Array` are implemented. These are accessed, at the lowest level, via `basicAt:`. The message `basicSize` tells us how many indexable slots the object has.

On the other hand, we have the named slots that are used to hold the values of instance names. These slots are accessed via `basicInstVarAt:`, and their amount can be obtained by sending the message `instSize` to the class of the object in question.

Therefore, we could arrange our object forms such that the distinctions inside `anObject` are numbered starting at 1, and going up to `anObject basicSize + anObject basicClass instSize`. And since it seems that `anObject` is going to be receiving things like `basicAt:` and `basicInstVarAt:`, we could reference `anObject` in the instance name `distinctions`.

Thus, it seems that the only thing we need to do is to create instances of `ObjectForm` on arbitrary objects, and have them figure out what are the named distinctions as part of their initialization process. In fact, to avoid having to ask the same questions over and over again, we could have them remember some of this information. Piece of cake.

```
Form (distinctions)
  ObjectForm (distinctionsBasicSize
    firstName lastName parentForm)
```

```
ObjectForm class>>on: anObject

  ^self new
    distinctions: anObject;
    yourself


ObjectForm>>distinctions: anObject

  distinctions := anObject.
  self distinctionsBasicSize: anObject basicSize.
  self lastName: self distinctionsBasicSize
    + anObject basicClass instSize.
  self firstName: (1 min: self lastName)


ObjectForm>>distinctionNamed: anInteger

  ^(anInteger between: self firstName and: self lastName)
    ifTrue: [self at: anInteger]
    ifFalse: [nil]
```

While the message `ObjectForm>>distinctionNamed:` is public because it refines the implementation in `Form`, the message `ObjectForm>>at:`, shown below, is private.

```
ObjectForm>>at: anInteger

  anInteger <= self distinctionsBasicSize ifTrue:
    [^self distinctions basicAt: anInteger].
  ^self distinctions basicInstVarAt:
    anInteger - self distinctionsBasicSize
```

With instances of `ObjectForm`, we can pretend that any object in the image lives in the world of Laws of Form.

There is perhaps one more detail to take care of. Since our reference finder will be traversing all distinctions of each object form, one would expect that it would send `distinctionNamed:` to the object form for each available name. This causes problems.

In order to send the message `distinctionNamed:` to the object form, the sender would have to know how object forms label their distinctions. What is more, senders would have to use the fact that object forms label distinctions with consecutive integers. But that data point is not in the public domain. Why would users of `ObjectForm` have to use knowledge about the mechanism by which `ObjectForm` can access all of its distinctions, especially when this information depends on encapsulated implementation details?

It looks like a case of insider messaging.

One could argue the scenario would be different if senders asked the object form for all its names first. But even so, if the enumeration has to travel through all of the distinctions anyway, why should users of `ObjectForm` even care about how many are there or what is the naming scheme?

Even from a pragmatic point of view, repeatedly sending `distinctionNamed:` will repeatedly range check the message argument. So now senders would have an incentive to send the private message `at:` to get around the problem, and therefore the implementation details of `ObjectForm` would be duplicated across all its users.

In short, users of `ObjectForm` should not rely on private implementation details to implement an efficient enumeration mechanism. Instead, they should delegate that behavior to `ObjectForm`, just like it is done with collections. In other words,

```
ObjectForm>>namesAndDistinctionsDo: aBlock

  1 to: self distinctionsBasicSize do:
    [:eachName |
      aBlock
        value: eachName
        value: (self distinctions basicAt: eachName)
    ].
  self distinctionsBasicSize + 1 to: self lastName do:
    [:eachName |
      aBlock
        value: eachName
        value:
          (self distinctions basicInstVarAt:
            eachName - self distinctionsBasicSize)
    ]
```

As far as efficiency goes, note that by being implemented in `ObjectForm`, the iteration is even more efficient because it does not need the checks present in `ObjectForm>>at:`. By keeping the implementation close to the context it runs on, we obtain several advantages: faster execution, no sprinkling implementation details across multiple objects, and a much more expressive implementation on the sender side.

We just need one more detail. Since we are modeling objects that behave both as forms and distinctions, we should provide a special implementation of `cross:` so that it answers an instance of `ObjectForm`. In fact, you may have noticed the instance name `parentForm` when we subclassed `Form`. We did not discuss it then, but it is useful because forms do not provide the bidirectional linking that distinctions have. Therefore, since instances of `ObjectForm` already know about their contents, all they need to know is their parent form. In other words,

```
ObjectForm>>cross: anInteger

  ^self class new
    distinctions: (self at: anInteger);
    parentForm: self;
    yourself
```

Now that we can model and traverse objects in terms of Laws of Form, it is time to write a reference finder for them.

5.4 Collaborations and emergent properties

Let's consider the requirements for our reference finder now. In particular, what is exactly its purpose? It turns out that it is connected to the garbage collection services provided by the virtual machine.

As we discussed before, objects that are referenced, directly or indirectly, from objects like `Smalltalk` are considered not garbage. This also means that objects that cannot be found after traversing the image starting at `Smalltalk` are considered garbage, and thus the space they take can be reclaimed by the garbage collector.

What does it mean, exactly, that objects cannot be found after traversing the image starting at `Smalltalk`? Perhaps the part about traversing the image is clear already: anything that is known by `Smalltalk` is not garbage, anything that is known by objects known by `Smalltalk` is not garbage, and so on.

But what are we referring to when we say that objects cannot be found by this traversal? From a pragmatic point of view, we may consider that since we cannot send messages to garbage objects, it is safe for them to be physically removed from the image. But there is a deeper, more subtle implication at play here.

What makes an object garbage is that if any objects know it by name, then it is only because the referrers are also garbage. In other words, objects are garbage when they are not stored in any instance name (indexed or not) of objects known to be not garbage[6].

In practice, garbage detection works by making `Smalltalk` not garbage by definition, any then stating that instance name references from objects that are not garbage make the referenced objects inherit the property of being not garbage through the reference.

Fortunately, all of this is done by the virtual machine for us, so we do not have to deal with it explicitly. In fact, sometimes we may even rely on the garbage collector to make up space by getting rid of stuff our objects do not reference any more.

All is well and everybody is happy until, for some reason, the virtual machine refuses to let go of some object. This means that there is a reference to it, that there must be at least one instance name pointing to it from some object that is known to be not garbage.

Sometimes we remember where the reference is. But that does not always happen, especially in object systems composed of a large amount of parts. So, where is the reference to our suspicious object, and which is the object that knows our suspect by name?

It turns out that we can have Smalltalk figure this out by itself. Our objective is then to find all references to an arbitrary object starting from `Smalltalk`. In order to find an answer to that question, we will replicate the work of the virtual machine — to a point. Instead of looking at objects in terms of a Smalltalk space, we will look at them as forms and distinctions.

5.4.1 Intentions and distinctions

By now, you might be thinking that the solution to this problem eventually depends on messages like `namesAndDistinctionsDo:` — and you are right. As

[6]These would be so called *strong references*. However, there are objects that can refer to others by means of *weak references*. Weak references are not followed by the virtual machine as it traverses the image looking for garbage.

important as some messages are, however, it is even more important to send them in a proper context. There is no hurry, we will write those message sends at some point. The more pressing issues are to determine what parts should our reference finder be composed of, and how should these parts interact so that the emergent properties of the whole let us satisfy our goals.

If you have never done so, this would be a good moment to inspect `Smalltalk` *in a live image.*

The image can be seen as a directed graph starting at the object `Smalltalk`. This graph, in turn, looks like a map as well. So, in a way, if we thought of `Smalltalk` as Rome, we could look at this problem in the context of the phrase *all roads go to Rome*. But what would our reference finder look like from this point of view? Well, since we are already talking about maps, streets, and perhaps some sight seeing, we could think of it in terms of finding a shop by driving past all street blocks, starting at the geographical center of Rome[7].

I am not up to do all that driving, are you? Just finding what we are looking for is enough work already. Because in addition to the driving, we would have to remember all the places we have traversed. Moreover, we would need enough time to look at each block with attention, and while we do that we would not be able to devote ourselves to things like the steering wheel and the pedals. This is asking to get lost at best, and an accident waiting to happen at worst.

In other words, there is too much responsibility for just one person. Let's boldly hire a taxi instead. We will let the driver take us through all the street blocks of Rome in whatever fashion he desires, with the arrangement that, at each corner, he will ask us if we are ok with the direction he wants to go next. Our task will be to remember all the street blocks we have already visited, and pay attention to the street blocks. And, if the driver asks us if we want to go through a block we have already seen, we just say no and he will then choose another block of our liking.

So there seem to be two objects: the driver, and the passenger. How can we characterize the relationship between the two? To the passenger, the driver provides a stream of street blocks, separated by requests to make a decision. To the driver, the passenger is just someone to feed a stream of blocks to, in exchange for decisions about the traversal. The passenger does not need to know details about how the traversal is being performed as long as it visits all the street blocks. The driver does not need to know details about how the passenger makes decisions either, he is just being paid to drive.

[7] This, in turn, is closely related to the issue of visiting all halls in a labyrinth. The solution to this problem requires significantly more work than just being able to enter a maze knowing one can also exit. See for example "Recreative Problems and Experiments", by Yakov Isidorovich Perelman.

Neither party, alone, can solve our problem. But their interaction certainly will. Thus, we will adopt this design for our reference finder.

5.4.2 The passenger

Since the behavior of the passenger can be described independently of how the traversal is performed, this allows us to write an implementation for it that will work with *any* traversal we can think of. Therefore, let's create our first reference finder class: `PathTracer`.

Since we have the class `PathTracer` as the first brush stroke on our canvas, we can easily think of the following class message:

```
PathTracer class>>pathsTo: anObject using: aStreamClass

  ^self new
    pathsTo: anObject
    using: aStreamClass
```

Now let's consider how we could implement the instance side message sent above. Since it seems that the newly created instance will have to remember some objects by name, we must give some consideration to what instance names it will need to do so. As a passenger, the path tracer should know what shop it is looking for, which we will refer to as its `target` object. And since we are talking about things the passenger needs in order to perform its duties, it should have some way to remember all the places it has visited, which we will refer to as its `storage`. So, at first, we could provide the following implementation for `pathsTo:using:`.

```
PathTracer>>pathsTo: anObject using: aStreamClass

  self target: anObject.
  self initializeStorage.
  self traverseWith: aStreamClass
```

This is great, but since our passenger will also take note of the ways in which the target can be found, it also needs a way to answer results. Let's consider this carefully. To begin with, all paths to our target would start at `Smalltalk`. In particular, it may be that some references have a common street block other than `Smalltalk`. When that is the case, it would be nice to merge these paths so that their common roots are listed only once in our results.

How could we do this with ease? Well, one could use a dictionary plus the `at:ifAbsentPut:` message. This would be a quite straightforward use of the standard Smalltalk class library. But... we have already seen how much better we can do and how much more expressive we can be. It may be invisible at first, but it is so obvious once we think of it. We could simply let the results be an instance of `Form`.

Of course! The axiom regarding the repeated use of a name fits this perfectly. To me, at least, it is quite beautiful to first see the Smalltalk image in terms of forms and distictions, and then provide answers in the same frame of mind. But, besides the fact that it is indeed pretty, there are more advantages to this approach. In particular, it will save us from writing code to map one point of view into the other. We can have the cake and eat it too. Therefore,

```
PathTracer>>pathsTo: anObject using: aStreamClass

  self target: anObject.
  self form: Form new.
  self initializeStorage.
  self traverseWith: aStreamClass.
  ^self form
```

Let's continue painting. The next thing to do is to initialize the object storage. What could be difficult about that?

Many things. Here we find a technical issue we must handle carefully. Ideally, since we will remember visited objects in terms of their identity, we would just write something along the lines of

```
PathTracer>>initializeStorage

  self storage: IdentitySet new
```

Some of you may be familiar with this already, but instances of `IdentitySet` rely on sending the message `identityHash` to access and manage their contents efficiently. And the speed with which they can do this relies heavily on the values of `identityHash` being pretty much unique across the objects they contain.

In VisualWorks, there are 14 bits dedicated to the values of `identityHash`. This means that we will see, at most, $16,384$ different `identityHash` values. But clearly, there are way many more objects than that in a Smalltalk image, and thus instances of `IdentitySet` will become inefficient very quickly. While

refusing to go over the details at this time, our handy Smalltalk mentor tells us that the performance impact can be quite significant. Let's assume that it is so, at least for the time being. What to do?

This turns out to be a rather thorny problem. But perhaps we are lucky. Browsing the hierarchy of `IdentitySet` reveals that it has a subclass called `ObjectRegistry`. Reviewing its implementation, one can get the impression that it refines `IdentitySet` with speed in mind. We could use it for now, but let's keep a note to ourselves: we really need to review exactly what is this business of hashing objects.

Therefore, feeling a bit of discomfort from doing something we are not sure about, we write:

```
PathTracer>>initializeStorage

  self storage: ObjectRegistry new
```

Since this object registry will hopefully behave like any other identity set, at least we seem to have a temporary excuse to pretend there is no performance issue while we implement the rest of the path tracer. If nothing else, we have quite a bit to do yet, and execution speed will only be a problem once we get this thing working.

When there are plenty of issues, more often than not the best thing to do is to tackle them one at a time while forgetting about the rest. This helps keeping our stress level down and our best intellectual abilities intact[8].

So... after staring at the implementation of `initializeStorage` for a while, perhaps we start wondering if it is correct to begin with. The easiest check is to pretend the implementation is done, and then to make up some mock example to see if it does what we think it should. Let's say we start our traversal with an empty storage, which will fill up as we visit objects. That seems fine until we consider the special cases. What if our traversal took us back to `Smalltalk`? Then we would not recognize it as an already seen object and waste our time wondering why is this taxi driver taking us to all these places we already know. So we should add `Smalltalk` to the storage to begin with, and then we would save ourselves a bunch of trouble. As we will make reference to `Smalltalk` from other methods, let's give it a name in the form of a selector.

[8]It is a well-documented fact that fear progressively and literally blocks our intellectual abilities. When in panic, people frequently cause their situation to become worse as a result of irrational decisions.

```
PathTracer>>scanRoot

  ^Smalltalk
```

With this convenient message, now we can refine `initializeStorage` to send the message `scanRoot` instead of referencing `Smalltalk` directly.

```
PathTracer>>initializeStorage

  self storage: ObjectRegistry new.
  self storage add: self scanRoot
```

There is one more detail. What if our traversal takes us to ourselves? Then the taxi driver would drive us through street blocks we already know we are not interested in. In particular, it will drive us through our own storage. What a waste of time!

The issue is that we are supposed to be separate from the image, and the act of traversing ourselves breaks this illusion. But in order for something to observe itself, first it must divide itself into two[9]. Therefore, we should refine `initializeStorage` once more.

```
PathTracer>>initializeStorage

  self storage: ObjectRegistry new.
  self storage add: self scanRoot.
  self storage add: self
```

Now that the storage is taken care of, the next message we need to implement is `traverseWith:`. It turns out to be much simpler than what it may seem at first glance.

```
PathTracer>>traverseWith: aStreamClass

  self traversal: (aStreamClass on: self newScanForm).
  self traversal drive: self
```

[9]Think of the implications, and how this makes things like Laws of Form look even more attractive.

This just follows the driver and passenger approach we discussed before. For the sake of completeness, let's check the implementation of the message `newScanForm` as well.

```
PathTracer>>newScanForm

  ^ObjectForm on: self scanRoot
```

Note how this time the form is an instance of `ObjectForm`, while the results held by the passenger are stored in instances of `Form` instead. This reflects the deliciously subtle and meaningful change in point of view between the passenger and the driver. In fact, it is time to cross that boundary ourselves. The next message we need to paint is `drive:`.

5.4.3 Abstract traversals

From the point of view of the taxi driver, what does this process look like now? Other than the fact that there is a passenger riding in the back seat, what is he aware of?

The driver knows of its particular traversal technique because it knows its own implementation. This knowledge allows it to generate new movement vectors in the information space it is traversing. Of course, in order to traverse its space properly, it also needs to know about its current position in the space. This is the static context that is fed to the movement vector generator.

Yes: this is just another implementation of `match:`!

Therefore, instances of `FormTraversal` will have two instance names. One will be called `passenger`, and in practice it will hold the path tracer. The other will be called `form`, and it will hold the current location the driver is at. Let's paint now, as you may be guessing how `on:` is implemented on the class side:

```
FormTraversal class>>on: aForm

  ^self new
    form: aForm;
    yourself
```

The implementation of the message `drive:` is now quite self explanatory — maybe a bit obvious.

```
FormTraversal>>drive: anObject

  self passenger: anObject.
  self takePassengerOnTraversal
```

Before implementing the message `takePassengerOnTraversal`, let's design the terms in which the conversation between the traversal and the passenger will occur. As we mentioned before, the passenger needs to tell the traversal whether to explore a particular street or not. It is a bit different with objects, but the analogy still works.

By the time the traversal reaches a new object, it can assume the passenger wanted to visit it. Clearly, if the passenger had not wanted to visit it, then the traversal would not have crossed into it. Nevertheless, at this point the passenger still has to see this new destination at which the traversal arrived. Therefore, the traversal can tell the passenger that it has to visit the new object. At this point, the passenger has two courses of action, depending on whether the latest location is the desired target or not.

But before the passenger can make any decision, the action request from the traversal must be received first. Keeping in mind that the traversal's current location is kept by the instance name called `form`, the discussion above implies the following:

```
FormTraversal>>ifDistinctionsAreInteresting: aBlock

  self passenger
    visit: self form distinctions
    ifInteresting: aBlock
```

Note how the distinctions of the form are not a set. They are an arbitrary object, because the form of a traversal is an instance of `ObjectForm`.

Now let's paint further on the passenger side. If the object visited is the object the passenger was looking for, then it will need to record its position in its results form. But it cannot do that by itself because only the traversal knows how it got there. Therefore, it will have to ask the traversal to write down the current traversal position on its results form. On the other hand, if the object visited is not the target, then the object visited is interesting in the sense that further exploration is needed. Therefore,

```
PathTracer>>visit: anObject ifInteresting: aBlock

  self target == anObject
    ifTrue: [self traversal crossPositionOn: self form]
    ifFalse: [aBlock value]
```

Back on the traversal side, what should we do to paint `crossPositionOn:`? The current location of the traversal is its form... but it is not just a regular form, it is an instance of `ObjectForm`, and it turns out that these know their parent forms. With a bit of care, we can let them walk up the parent chain, and start crossing themselves on the results form.

```
FormTraversal>>crossPositionOn: aForm

  self form crossOn: aForm


ObjectForm>>crossOn: aForm

  (self parentForm crossOn: aForm)
    cross: self distinctions
```

To avoid `nil` checks for when this process reaches the scan root, we also add the following message.

```
UndefinedObject>>crossOn: aForm

  ^aForm
```

How do undefined objects cross themselves on a form? By not doing anything. Of course, because they are undefined forms. Fantastic!

This is great, but how is the passenger allowed to have a say on whether or not an object is worth exploring?

Looking at things from the traversal side, if the passenger thinks an object is interesting, then it will need to traverse all the objects reachable from the current location. If the passenger thinks the current location is not interesting, then the traversal needs to find a new place to go according to its traversal strategy.

From the passenger side, in contrast, distinguishing an object as interesting would be like reaching a new street corner and deciding that all streets starting

from the corner must be examined. Of course, any interest would depend on what the passenger decides after consulting its object storage. If the object had already been visited, no further exploration is necessary.

In either case, a question from the traversal to the passenger is in order. We could start painting this bit of the conversation by implementing the following message on the traversal side.

```
FormTraversal>>shouldCrossInto: anObject

  ^self passenger shouldCrossInto: anObject
```

And then, we could implement the following on the passenger side.

```
PathTracer>>shouldCrossInto: anObject

  self target == anObject ifTrue: [^true].
  (self storage includes: anObject) ifTrue: [^false].
  self storage add: anObject.
  ^true
```

Note that the passenger will always want to visit the target, regardless of whether it has been visited before. This is because it is only when the passenger actually visits the target that the traversal will be aware of the target's position, and thus it is only at this time that the traversal will be able to record the target's location on the passenger's results form.

Now that our conversation between the traversal and the passenger is ready, we will attempt two different implementations of `takePassengerOnTraversal`.

5.4.4 Depth first traversal

First, we will try a depth first traversal. While we have already seen that it is not very useful, its simplicity will serve to illustrate the mechanism by which traversals can change position in their information space. Writing a breadth first traversal will be easier once we become familiar with how to move between object forms.

When the taxi driver moves about in a depth first fashion, as soon as the passenger finds an interesting object, its contents are scheduled for immediate examination. Given that the traversal's current position is stored in its `form` instance name, changing position means either crossing into one of the named

distinctions inside this form, or uncrossing from the form into its parent form. The process ends when the traversal of the scan root form is done.

Because of the particular behavior of this kind of traversal, it seems that a recursive implementation would capture its characteristics best. So how would the recursion work? Let's see... if the current distinctions are interesting then for each of them: cross into each, repeat the procedure currently being described for each, then uncross from each. In other words,

```
FormDepthFirstTraversal>>takePassengerOnTraversal

  self ifDistinctionsAreInteresting:
    [
      self namesToCrossDo:
        [:eachName |
          self crossInto: eachName.
          self takePassengerOnTraversal.
          self uncross
        ]
    ]
```

This method merits careful study, because it is all that is needed to perform a depth first traversal. Let's review the implementation of the messages sent by the method above.

```
FormTraversal>>namesToCrossDo: aBlock

  self form namesAndDistinctionsDo:
    [:eachName :eachDistinction |
      (self shouldCrossInto: eachDistinction)
        ifTrue: [aBlock value: eachName]
    ]
```

Aha! So this is both where the passenger is asked where to go, and the carefully crafted context from where the message `namesAndDistinctionsDo:` is sent! Not only this is intention revealing, it is also quite efficient: by inlining the question into the iteration, there is no need for intermediate collections of names.

Finally, let's examine the navigation messages. The ones below are used to dive into new objects.

```
FormTraversal>>crossInto: aName

  self form: (self cross: aName)

FormTraversal>>cross: aName

  self form cross: aName
```

The message below lets the traversal uncross from the current form into the form's parent form, and is the counterpart to the message `crossInto:` shown above.

```
FormTraversal>>uncross

  self form: self form parentForm
```

This is almost too easy — and there is barely any code to begin with! Let's try something a bit more difficult now. Can the breadth first traversal be written as elegantly?

5.4.5 Breadth first traversal

Something that distinguishes a breadth first traversal from a depth first traversal, is that the traversal of the neighbors of a particular location does not occur at a moment adjacent to the traversal of the location itself. Each onion layer is traversed whole before starting with the next one, and the layers as a whole are queued one after the other.

But how can we express this in Smalltalk? How do we begin to think of an implementation? Sometimes it is a good idea to pick something particular about a problem, perhaps something simple even, seeing what pattern it has to satisfy, and then building off from there. In other words, let's choose a constraint and see where it leads us to paint.

For example, there is that queue characteristic of the breadth first traversal. Once the traversal learns about new things to visit, it assigns them a turn and puts them to wait at the end of the queue. The idea is that eventually it will get to them and, perhaps, queue their interiors as well.

In this way, it seems we could have the traversal process queue things in an ordered collection. It would feed itself from the *left*, and it would add things

to the queue from the *right*. We could further arrange things so that the queue starts holding the scan root, and so that the process ends when the queue is exhausted. That looks promising, and so we write the following.

Stack the dominos close to each other. Push the first one. Watch.

```
FormBreadthFirstTraversal>>takePassengerOnTraversal

  self formsToVisit: (OrderedCollection with: self form).
  self depleteFormsToVisit
```

Now that the recursion is set, we should provide an implementation for the actual traversal behavior. Let's keep in mind that we should visit the current location, but then schedule any reachable locations for later.

```
FormBreadthFirstTraversal>>depleteFormsToVisit

  [self formsToVisit isEmpty] whileFalse:
    [
      self form: self formsToVisit first.
      self ifDistinctionsAreInteresting:
        [
          self namesToCrossDo:
            [:eachName |
              self formsToVisit add:
                (self cross: eachName)
            ]
        ].
      self formsToVisit removeFirst
    ]
```

That is it, really. No further implementation artifacts are necessary. What we wrote is intention revealing, efficient, concise, powerful, artful, beautiful.

In the method above, you can see the other sender of `cross:`. Note also how carefully is the list of forms to visit treated. It would not be possible to use `do:`, but `isEmpty` and `whileFalse:` do the job with ease.

With care, but not with fear.

One could question why isn't the message `first` replaced with `removeFirst`, which would allow to eliminate the last sentence of the loop. The answer is that, at first, it had been written that way. But when it was necessary to debug the traversal, removing first and then processing would change the queue. Therefore, restarting the inner block would lead to an execution scenario different than the

one the block had been running before. This kind of things can be the source of much irritation when trying to figure how things are or are not working.

But what really put the last nail on the coffin was that, ironically, a set of profiler runs showed that this apparent optimization resulted in no tangible benefits[10]. Therefore, the debugger friendlier version survived.

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." —Brian W. Kernighan

5.5 Final remarks

As interesting as this design is, there is one thing we have not done yet. We have not compared it to any existing reference finders in VisualWorks.

For example, the inspectors have a menu item called [`Inspect Reference Paths`]. Performing that action from an inspector looking at the class `Form` takes about 10 seconds in the image available to the author. After doing its work, it opens a series of arrays that, however, do not seem to track down the full reference path to the object in question. There are duplicates, some references are missing, and the paths do not start at a unique scan root.

Our reference finder, on the other hand, takes approximately the same amount of time, finds more references, and produces no duplicates. It seems we can claim our journey through the Smalltalk space is a success.

5.6 Exercises

Exercise 5.1 [20] It would seem natural for distinctions to know their names. Should it be so?

Exercise 5.2 [12] The class `Form` is a subclass of `Object`. Would it be a good idea to subclass from `Set` instead?

Exercise 5.3 [28] Implement `Form` in terms of a `Dictionary`.

Exercise 5.4 [36] Implement a counterpart of `PathTracer` that counts the size of all the objects reachable from a particular root.

[10] Interestingly, the time profiler shows that about 50% of the execution time is spent sending the message `==` to `nil` in `PathTracer>>isEmpty:` — just what the unoptimized version of `match:` would do.

Exercise 5.5 [17] The method below is inefficient because, in essence, it asks the storage the same question twice.

```
PathTracer>>shouldCrossInto: anObject

  self target == anObject ifTrue: [^true].
  (self storage includes: anObject) ifTrue: [^false].
  self storage add: anObject.
  ^true
```

Address this inefficiency.

Exercise 5.6 [24] The reference finder, as shown in the chapter, is inefficient because it will traverse objects that have no instance names as well as integers, byte arrays, and such. Refine the message `shouldCrossInto:` to filter out such objects.

Exercise 5.7 [18] Create a subclass of `Object` called `Example`. Add an instance name called `contents` to it, together with its accessors. Then, create this artificial instance.

```
example := Example new.
example contents: example
```

Why does the expression below fail to cause an infinite loop in the reference finder?

```
PathTracer breadthPathsTo: example
```

How about this alternate example instead?

```
example := Example new.
example contents: (Array with: example)
```

Exercise 5.8 [24] Implement a simple tree user interface for forms.

Exercise 5.9 [29] Enhance the reference finder to ignore references starting at instances of trippy inspectors upon request. This can be helpful when requesting references to an object being inspected, as the reference from the inspector itself is rarely what we are looking for.

Exercise 5.10 [16] Improve the reference finder so it can be configured to ignore references originating from debuggers as well.

Exercise 5.11 [10] Change the reference finder one more time so it can be told to skip references originating from the process monitor, as invoked via `ctrl`+\.

Exercise 5.12 [18] Replace the implementation behind the inspector menu item [`Inspect Reference Paths`] with our reference finder.

Exercise 5.13 [34] Sometimes, it is useful to see how two objects differ from each other. Write a different kind of path tracer that, looking at an object as its guide, examines another object and reports how both differ from each other.

Chapter 6

A pattern of perception

6.1 Motivation

At times, we will encounter complex problems that are structured such that a lot of detailed planning and careful work has to be done to tune what we think is the best solution, followed by experiments to determine if our guesses were correct or not. One such problem that comes up frequently is that of tuning a garbage collector.

In VisualWorks, most of the garbage collection behavior is delegated to the image. In particular, there is a class called `MemoryPolicy` that implements a particular garbage collection and memory management policy[1].

Typically, when performance adjustments are needed, what occurs is that the developers in question think of a particular use case, then tweak this or that aspect of garbage collection, and a decision is made on whether the change was good or not. This approach to problem solving usually works well, but it has a fundamental deficiency which is best illustrated by means of an example.

For instance, after going through all the garbage collection knobs that can be adjusted, it might seem natural to think of a strategy that adjusts to a particular memory usage scenario. However, after we look at the strengths and shortcomings of the current VisualWorks approach in order to gauge how to change them to suit our specific needs, we will see that any attempt to replace the fixed behavior implemented by the active memory policy with another fixed behavior

[1]This chapter originally described how VisualWorks' garbage collector works, among other things. However, because of the scope of the material, it has been moved into a separate book. In addition, this book used to have a seventh chapter about hashing, which has become its own book as well.

implemented by a new memory policy will also be flawed in that the new fixed behavior will not be able to adapt to previously unseen circumstances any more than the previous one. In other words, once we start down the path of manual intervention, we also go down the path of *repetitive* manual intervention.

In addition, if we just rely on manual work, we will soon find out that garbage collection tuning takes a lot of time and effort. Because of the variability involved, it is comparatively expensive to collect good information on which to make tuning decisions that will be correct over a reasonable amount of time. And, in most cases, there will be the often hard to disprove feeling that the execution time saved is less than the developer time spent tuning garbage collection to begin with, or that the time developers spent tuning garbage collection is worth more than buying a faster computer in the first place — regardless of whether this assertion is true or not.

Right or wrong, in practice this is the typical impression managers get.

Now, of course garbage collection is just an example. It could be any number of things, such as how to write a program that adapts to playing a game more successfully, or how to write a program that changes its behavior to optimize the use of available resources with different goals in mind, or how to write a program that optimizes itself.

Therefore, perhaps we would do better if we automated the process that we go through when we attack such problems. In other words, it would be more productive to have a program that could decide which observations to make and what to do as a result, as opposed to either a program that had a fixed behavior that was good for a limited set of situations only, or no program at all forcing us to rely on the energy and resources of the development team to obtain improved results each time.

But how are we going to write such a program to begin with? An in depth answer to this question is in order because automating a process we carry out typically involves much more than just solving the problem at hand. We should also concentrate on mimicking our behavior at the more fundamental level where matters such as "which of these observations am I going to take a look at?" are decided.

So if our goal is to try to come up with a way to copy our behavior when we deal with an issue, this will mean addressing two distinct concerns: how to copy our generic problem solving behavior into some sort of framework, and how to solve the actual problem at hand as an application of the framework. Let's concentrate on the first issue now. Hopefully, if we come up with a good abstraction and a flexible implementation, we will be able to use it to address the second issue more easily.

6.1.1 The first distinction

There is something we seemingly cannot escape doing when we tackle a problem. We typically take it for granted, but it is of the utmost importance. Pretty much the first thing we will do is to separate ourselves from the problem in order to observe it.

Here is where Laws of Form hits with all its might. Be sure to have the previous discussions about them in mind.

Think of how many times we have silently done that in the previous chapters. How familiar are we with this act? How explicitly can we become aware of its occurrence? How much can we use the process we are trying to describe to observe ourselves applying the process itself, so that we can describe what we do with enough detail so that we can describe it inside a computer?[2]

Perhaps this task seems daunting. However, the fact remains: we go through these motions all the time. We learned how to accomplish this somehow, and thus it follows that there is a way in which this knowledge was encoded in our minds for the first time. All we need to do now is to copy the signal processing structure we depend on to live our lives, by using the signal processing structure itself[3]. So, to begin with, if we remove ourselves from this problem as well and look at it from the point of view of Laws of Form, what distinctions appear before us? What differences in value are we looking at *exactly*?

In other words, take partial derivatives on each dimension of the observed space so a gradient can be used both for prediction purposes and to detect coordinates with small enough values which tell us the dimensions we can ignore according to the resolution we choose to perceive with.

A possible way to interpret what we do when we try to solve a problem (and in particular when we do object oriented work), is that one of the most important differences in value we pay attention to is whether the presence or absence of a distinction significantly affects the result of the evolution of the system at hand. In other words, after separating ourselves from the problem, the first thing we seem to do is to sort things into the following two buckets.

- Things significantly related to our problem.
- Things not significantly related to our problem.

The threshold between significant and insignificant is dictated by the required quality that our solution to the problem in question must have.

[2]Good programming depends on powerful abstractions, a clear mind where to observe the reflection of the abstraction, and clear expression of intent to reflect the abstraction into the computer. Note how this observation does not require our first person presence in the process, thus suggesting we should write programs thinking in the third person instead on the grounds that it is less involved and thus requires less effort.

[3]Note how doing so is an excellent opportunity to set up a tight feedback loop by which the abilities of our signal processing mechanism sharpen themselves...

This is typically why it is so hard to think about problems in which we are a part of the system that computes the solutions by means of the emergent properties that arise from the chaotic interaction of the system's parts. Our very presence affects the results, much in the same way that the uncertainty principle seems to ruin our attempts to obtain ever more precise observations. In order to think about our problems in a more powerful way, we need to create a context of observation, to carve out a portion of ourselves which is as perpendicular as possible to the issue at hand — because it is from this point of view that we will sort things into the "related" and "not related" buckets.

This is our first distinction, the boundary that separates ourselves from our environments and distinguishes us as observers.

6.1.2 Crossing the first distinction

In order to take a look at our problem from the first distinction, we have to cross the boundary the first distinction represents in a controlled manner. In the same way that we have the means to perceive and cause perceivable change in our environment, we will need to model a connection channel that allows the crossing of the boundary between our first distinction and our surroundings.

In Smalltalk, boundary crossing is achieved by sending messages. Note how object encapsulation provides a distinction that separates the contents of the object from its environment. There is no way to directly interact with the interior of an object's boundary. Instead, messages must be sent to the object with the tacit understanding that answers will be reasonable[4].

When we interact with our environment, we depend on the fact that we are sensitive to some of the messages that reach us. Our senses are passive receptors of this information: photons excite our retinas, the heat from the environment lets our skin know about the temperature around us, changes in air pressure are

[4]Life, in a way, works that way too. Cells have boundaries that separate them from their environments, and it is only through messages that multicellular organisms organize themselves. If a cell does not provide reasonable answers, its relationship with the rest of the cells becomes unstable. Outcomes typically range from eviction, to the whole cell colony shutting down entirely. One way or the other, improper interaction is strongly selected against.

The fact that in Smalltalk it is very easy to do damaging things, such as `true become: false` or `Object become: nil`, is sometimes used to argue that the lack of restrictions is actually a defect. This argument is most counterproductive. In fact, it is the lack of pessimism what makes Smalltalk powerful, as its scarce constraints let the cost of change stay low. Besides, were safeguards put in place against doing damage, they would immediately become dead code because they would never run after the program becomes stable. Thus, it is not worth to spend time or energy in them.

interpreted as sound, speech, or music. We measure the interference caused by the environment in us, and we then say we perceive things.

We also correlate things we do with the reaction observed in the environment we live in. The way we walk maintaining our balance on various types of ground depends on how the pressure felt on our feet is correlated with our estimated speed and acceleration in 3 axes. Many things, some trivial such as how we pick up the phone and dial a number, some fundamental such as the way in which we have learned to speak, or to whistle, or to sing, depend on the feedback we obtain from our environment. The precision with which this is done makes our everyday life a huge circus act[5].

"Although the whole of this life were said to be nothing but a dream and the physical world nothing but a phantasm, I should call this dream or phatasm real enough, if, using reason well, we were never deceived by it."
—Gottfried Wilhelm von Leibniz

These activities, which we take so much for granted, occur across the first distinction. We receive messages from a blob which itself is left uninterpreted — who could care, much less prove, whether the laptop I think I am writing this book on really exists or not! We should not worry about the actual implementation details, in the sense that as long as we can have a stable interaction with our environment, further inquiry is unnecessary. It is only when we perceive a fault in our understanding that we first pinpoint an unexplained difference in value and then embark ourselves into a new journey of investigation and discovery. Therefore, what we should pay attention to is by what mechanisms we interact with our environment across the boundary that distinguishes us as observers, what is the nature of our interactions, and what are the attractors implied by these activities.

Thus, let's assume that, as happens with cells and other things which lay inside a distinction, there is an *interface* attached to our boundary which allows us to be aware of some messages going in and some other messages going out. This is our connection to our environment. Assuming we can trust our surroundings truly exist in as much as they will provide feedback, on one hand the interface provides us uninterpreted perceptions of the blob around us, and on the other hand it allows us to cause changes in the blob.

The ethernet cable is connected. This provides the basic scaffolding setup on which an observer entity can evolve. Now, what we can say about the traffic coming and going through the wire?

6.1.3 Making sense of raw perceptions

Let's examine the incoming part of our interface first. In fact, let's pay very close attention to the subtle details, as this process is so automatic for us that it has

[5]See "This Side Up", episode 2 of the BBC series "The Real Thing", by James Burke.

become effectively invisible.

One level where some of the work is done without us even noticing is where the raw perceptions are massaged before they are even interpreted. For instance, the optic nerves of our eyes cause blind spots to appear well within our field of view. But unless we try to do something like looking at stars with that particular region of our retinas, we will never notice the insensitive spot because our brains fill them up for us.

Another level is where interpretation work is performed. For example, take a look at this page. It is unavoidable to recognize letters, is it not? It might even be impossible to look at a letter "e" without immediately recognizing it as a letter "e", instead of it being seen as some arbitrary scribble on a piece of paper.

So how could we describe the process by which we go from "some differences in perceived color", to "scribble on a piece of paper", to "this particular sentence and what it means"?

Assuming one has raw data coming from our interface, what we typically do is to scan it for differences in value. In the case of this piece of paper, the difference in value could be described in terms of "darker here", or "lighter over there". By grouping similar values together, we draw distinctions such as "the inside of this figure is darker than its surroundings". If these distinctions follow the same path of recognition as other distinctions we have previously learned to draw, we then *recognize* the new distinction as being equal to the previous one. This is pervasively performed for everything we do. No wonder brains are good at pattern matching.

Note how the way in which distinctions are drawn could be arranged to be their own description as well. In fact, it would be very convenient to have the way in which we recognize a letter "e" to be the lookup key attached to a hash bucket labelled "letter e", for example.

`a = b ⇒`
`a hash = b hash`

Pay special attention to the implications: whether the hash bucket is labelled "letter e" or not is absolutely inconsequential. The only thing that matters in our recognition game is that similar enough distinctions end up in similar enough hash buckets[6].

So what if we had hash buckets of limited resolution? Then, eventually, we would have label collisions and things would get messy. This is where intentions come in. Depending on what we mean to do, we may end up looking at only one of the possible interpretations for the hash bucket, thus disambiguating the meaning of the observed label. Or perhaps our purposes will call for looking at things with more resolution so that previously colliding observations will end up

[6]You may want to become familiar with the Heckbert quantization algorithm at this point.

in a different hash bucket entirely. One way or the other, collisions happen and we deal with them either by refining the precision of our point of view, or by switching our point of view entirely.

In general, `a hash` ~= `a identityHash`*, depending on the intentions at play for values of* `a hash`*.*

To summarize, based on the raw data coming from the interface, we create distinctions inside our first distinction according to the intentions with which we care about perceived differences in value. We will refer to the entity that executes this mapping process as our *eyes*.

6.1.4 Mapping stimuli to reactions

Assuming our eyes are working properly, we will have our raw perceptions sorted into nice looking hash buckets. But what to do with these now?

First, we should consider how come the hash buckets exist in the first place. This is so that we explicitly describe what gives them a reason to come into being, because the buckets are the distinctions drawn as a result of having some intention.

Some intentions are about using the incoming connection of our interfaces, such as "distinguish all the pieces of equipment in my home office". Some others, however, require using the outgoing portion of our interfaces, for example: "look up a friend's phone number and call".

Intention induces relationships between the hash buckets corresponding to the things we perceive, and the things we know how to do. The map is built so that, after a period of training, we can quickly select the actions that we predict will satisfy our requirements with high probability. Of course, the things to do themselves are sorted into hash buckets as well.

Thus, in the same way that the $3x + 1$ problem implied a behavior map in the integers, the space of distinctions observed as hash buckets according to our intentions needs to be mapped into the space of hash buckets corresponding to things to do. The problem is now how to choose the map wisely so that the feedback cycle of this mapping application converges to an attractor that has the emergent properties that satisfy our intentions.

This can be difficult for two reasons. If we see ourselves, together with the two hash bucket spaces, embedded into a space that has time as one of its dimensions, our first problem is that it is frequently impossible to have full knowledge of our position in this space because our presence affects it too much. But even when we achieve a reasonable degree of clarity of observation, our hash buckets typically have limited resolution — and even then, they can only resolve a subset of the information space we live in. More concretely, there are only so many colors

we can see, so many sounds we can hear, and in general there is only so much resolution with which we can perceive.

Because of this lack of precision, the things we decide to do are (most of the time) approximations made in the hope that the actual results will be close enough to what we predicted. It is truly amazing that we can do so much with so little. At the same time, this means we are bound to run into cases when our approximations will not be as good as we need them to be. What are we going to do about that?

Let's be bold and take for granted that we are always going to have prediction error. Then, predicting too far out into the future will cause the prediction error to accumulate and grow quickly[7]. The better our hash buckets model the actual process at hand, the smaller the prediction error will be. However, no matter how good our understanding of something is, there is always the risk that something we are not aware of is at play[8].

It follows that, more than being able to make extremely good predictions about the future, what matters is the ability to make corrections and adjust the predictions on the fly as soon as prediction error is detected, together with moving in the information space in such a way that always leaves room to make such corrections. In this way, given some general direction in which we know we want to move towards in our information space, we should spend time making sure that no matter which path we are going down on, the prediction error is not leading us astray and the required precision of movement is not beyond our reach. In short, what we need to ensure is that our lack of precision does not end up moving us so far away from the attractor we want to reach that we end up moving into the influence zone (or *basin*) of an unwanted attractor instead[9]. Progress is defined in terms of the contraction of the possible solution space.

---

[7]Note how this relates so well to numerical integration of differential equations!

[8]Hence the huge market for risk management of all sorts.

[9]If we allow our environment to be represented by a system of linear equations up to some degree of accuracy, then its attractors will be given by the eigenvalues and eigenvectors of the approximating matrix. Thus, all we need to do to let it converge is to make sure our prediction error is less powerful than the eigenvalues. If all we are looking for is our particular attractor, it does not even matter if the prediction error is comparatively large, because as long as it is bounded quick convergence to the eigenvectors is guaranteed.

Moreover, note how we can draw a distinction between programming languages that are early bound and those that are late bound. Early binding assumes that developer predictions into the future are of an extraordinary, almost clairvoyant quality. This is especially so as soon as one accepts that things change all the time. As such, the cost of the associated correction that becomes necessary when it is found that the attractor is too far away rises accordingly — so much so that success typically lies beyond reach.

Let's review a concrete example that illustrates how these mechanisms work. Basically, there are two ways in which one can write programs that play chess. On one hand, we have the brute force methodology which examines every single possible move up to some practical limit. On the other hand, we have algorithms which try to determine whether certain play combinations are worth examining further. In some cases, they may also decide whether to wait until the next turn and stop making decisions until new information is known. Today's chess programs are somewhere between these two extremes.

Note then how the problem of prediction error manifests itself immediately. If a program is brute forcing the search for good moves, then the vast majority of its work will become useless very quickly. And in almost every turn, the vast majority of the computation must be redone as well — unless the opponent's play matches what was seen as the best move by the previous round of calculations. The likelihood of large prediction error associated with brittle predictions that go too far out into the future results in high correction costs that in this case can be measured in terms of the computing resources spent for each move.

Pruning parts of the search that are deemed worthless tends to lower the prediction error even if the predictions span over a relatively long time. High quality predictors that concentrate on the most interesting evolution paths of the system thus have less prediction error to deal with, and as such incur in lower correction costs when prediction error is found.

Modern chess algorithms rely more and more on managing the risk associated with prediction error instead of focusing on optimizing the brute force search of best moves. And in fact, the modern approach works better as routinely shown in computer chess tournaments. But besides what could seem to be anecdotal evidence, there is a stronger reason why this way of designing computer programs is desirable. There is plenty of evidence that this approach works very well in nature. If billions of years of evolution selected this arrangement of interaction, it is my humble opinion that we should do the same when writing software. This is a wheel that does not need to be reinvented.

Going from chess back to our more abstract problem, how can we summarize what we learned from the discussion above? To put it succintly, the design of the map between hash buckets corresponding to perceived distinctions and hash buckets corresponding to actions to take depends heavily on the estimated quality of our predictions.

If we think we can predict well over long periods of time, then naturally we should increase the resolution of the hash buckets to take advantage of this. However, we should keep in mind that high quality predictions have a tendency to

take significant time to calculate. In some cases, the time lag between prediction computation and action can introduce enough prediction error to the point that the results of some or all of the effort spent deciding what to do become useless or counterproductive.

If, on the other hand, we can only predict reasonably over short periods of time, we might as well lower the resolution of the hash buckets we are using. What is the point of spending a lot of energy coming up with a prediction that has less uncertainty than the known minimum prediction error?[10] In these cases, it is more important to correct any deviations as quickly as possible, instead

[10]On page 161 of his book "Recreative Algebra" (Spanish edition, written no later than ca. 1940), Yakov Isidorovich Perelman notes the following.

> Not long ago, our schools used logarithm tables having 5 significant decimal digits. Currently, we use tables with 4 significant decimal digits because they satisfy the needs of technical calculations. However, for the majority of practical applications, tables with 3 significant decimal digits are more than enough because common measurements are hardly ever taken with more than 3 significant decimal digits.
>
> The use of few significant decimal digits is a recent development. I remember the times when schools mandated the use of voluminous logarithm tables containing 7 significant decimal digits, which were replaced with tables having 5 significant decimal digits only after much arguing. It appears that when 7 significant decimal digit tables appeared in 1794, they were rejected as an inadmissible novelty. The first decimal logarithm tables ever, made by the British mathematician Henry Briggs in 1624, had 14 significant decimal digits. A few years later, the Dutch mathematician Adriaan Viacq reduced them to 10 significant decimal digits.
>
> As we can see, the evolution of commonplace logarithm tables has been in a restrictive sense, going from more to less significant decimal digits. This process has not yet finished nowadays, as there are still those who do not understand that the precision of calculations cannot exceed the precision of measurements.
>
> The reduction of mantissas has two important practical consequences. First, the significant reduction of the space required to store the tables; and second, the corresponding simplification of their use which in turn speeds up calculations performed using them.
>
> Printing logarithm tables with 7 significant decimal digits takes 200 large pages. Making a hard copy of the 5 significant decimal digit tables takes 30 pages half the size of the previous ones. The tables with 4 significant decimal digits take ten times less space, and thus can be printed in just 2 large pages. Tables with 3 significant decimal digits fit in one page.
>
> As to the speed of the calculations, employing tables with 5 significant decimal digits takes one third of the time required by using tables with 7 significant decimal digits.

of trying to determine the best possible course of action. In other words, when predictions are known to be of limited quality, what we should do is to use a good enough prediction calculated efficiently together with a consistent error correction mechanism.

Another case in which it can be beneficial to lower the hash bucket resolution is when we can show that coarse, comparatively cheap short term predictions produce results just as good as more expensive, longer term considerations. As a valuable side effect, this may also have the advantage of lowering the feedback cycle time lag without compromising the quality of the overall convergence of the behavior to its attractor, thus making it easier for it to react quickly to unexpected circumstances.

6.1.5 Choosing which benefit function to maximize

So how are we going to strike a proper balance between time spent predicting the behavior of the environment being observed and the ability to react quickly to unexpected changes? While in general that depends on the particular problem we are trying to solve, the process by which the actions to perform are chosen will be referred to as a *strategy*.

At first it may seem that a strategy should produce a series of concrete operations that are to be queued for execution. This however is not necessarily so. To see why, let's review a real life example in detail.

Consider any sport and you will see that competing parties try different strategies according to their perceived strengths, with the idea that the one with stronger eigenvalues wins (the eigenvectors are fixed because the game rules themselves are fixed)[11].

Playing any game well requires choosing a strategy in advance. Strategies can be compared before the game is played by measuring how much effort is

[11]Roughly speaking, a square matrix A is said to have eigenvectors v_i and eigenvalues α_i when $Av_i = \alpha v_i$. In some sense, the eigenvectors represent the directions in which the matrix A compresses or enlarges the vector space in which it operates.

An attractor corresponds to $|\alpha_i| < 1$, and a repeller to $|\alpha_i| > 1$. This is because, by definition and by the properties of matrix operations, we have $A^n v_i = \alpha_i^n v_i$.

Let's consider two players, the strategies of which can be thought of as matrices A and B. The idea of winning is to see which matrix has stronger attractors with respect to the game. In other words, it is a competition to see which of the matrices has eigenvalues closest to zero and eigenvectors most aligned with the eigenvectors that describe a winning situation in the game. If both players can play simultaneously, a game can be interpreted as the "continuous" evaluation of $(A + B)v$, where v describes the state of the game. If the game has turns, it can be seen in terms of recursively evaluating $Av = w$, then $v = Bw$.

required to find "good" eigenvectors and eigenvalues. In addition, being proficient in competition sometimes requires dynamically adjusting how much effort to spend while the game is being played. The effort can be measured in terms of the degree of accuracy to which actual predictions and actions are meant to mimick the eigenvectors and eigenvalues chosen.

So let's say we are the coach of a sports team and that we get to decide the overall strategy that is to be followed by the players. Do we get to say exactly what the players will do? Not really. At best, the resolution with which we specify the encompassing goals the players should try to accomplish is many times removed from the level of detail players will have to deal with.

Consider soccer. As coaching staff, we get to decide the way players stand in the field, what kind of plays should be preferred over which others, and so on. But when it comes time to make a decision as simple as making a pass, we have no control whatsoever.

Or consider basketball. Sure, the strategy could be as simple as "just give the ball to Michael Jordan", but as coaches we will not have direct control over what Michael Jordan does.

This, of course, says nothing about what the opposing team players will do, and how our players will have to adapt to new information as it becomes available. No amount of micromanagement can bridge this separation between us and our players. The most we can hope for is that our players keep our guiding principles in mind when they decide how to move in the information field they perceive. We do so with the idea that if they are good players, they will figure out ways to move that support the overall strategy we as coaches decided upon.

If we model what we think happens in this way, we can also see that it covers sports in which there is no obvious coach. For example, let's review how this model for strategies works with golf. In this situation, the coach and the player are one and the same, and the person playing the game must switch contexts back and forth. The coach selecting the strategy may make decisions such as executing this or that kind of shot, how to hit the ball to give it a particular type of spin, and so on. Once all these decisions are made, what the coach produces is a specification of the ideal thing to do under the given circumstances and according to his or her intentions. Note that, as in other sports, the work product of the coach are *objectives*. These are given to the player to carry out, and it is only at this point that minute details such as the precise height of the ball above the grass blades comes into consideration.

To summarize the work of the strategy, then, once it chooses the general kind of direction it wants to move towards through the information space, the player

will be able to finetune how much to move in the direction given. Thus, we really have objectives that, like functions that we want to maximize, can tell us which is the direction of movement that produces the maximum function value.

6.1.6 Going after maximum local benefit

Now, if we are a point where the strategy has produced objectives, how is it that the player plays exactly? In particular, what is the least amount of activity necessary to meet objectives?

If we look at what we just described in the previous section from a different point of view, what is going on is that the strategy is choosing an attractor. And, more subtly, it does not specify the attractor explicitly but rather implicitly, by providing a way to measure how much each possible action contributes to getting closer to the attractor. The actual work of trying to achieve the objective, the search for the best possible move is left undone until the player actually plays.

Remember our travelling behavior functions that had a static context? The objective is what aims the behavior process' movements in its environment, but it does not spell out what the attractor is directly. The idea is to cause the attractor to appear by specifying a map in the information space that has the desired attractors.

Did you notice what we just said above? The objectives *aim* the player and its behavior process in its travels through the information field, right? But if that is so, what is really in control is the *objective*!

Note how, once again, the idea is to take advantage of how just the mere expression of ideas into words cause the rest of the details to appear out of thin air.

Ok, so what does the objective control then? What the objectives control is the direction in which the player will move. And how do players move? Well, since we had the body part metaphor going when we said there was a part of the process which we would refer to as the eyes, then we will push it further by stating that the objectives control the *hands*.

The setup of a having strategy that chooses objectives that control the hands has a number of advantages. First, from the point of view of the strategy, decisions can be made with enough resolution to be helpful, but not so much resolution as to make the whole process too expensive. Second, from the point of view of the player, if it has been told how it should move in its information space, then all it needs to do is to follow the given path as closely as possible concentrating only on dealing with the unavoidable prediction error. In other words, the division of responsibility is done such that each task is rather simple *because the context in which it has to work is relatively small*[12]. Third, this approach allows efficient

[12]If this holds, it should be easy to implement it in understandable code as well, right? We

use of computation resources because it allocates said resources to perform only what is most important and absolutely necessary. For example, if there are no objectives then clearly it is useless to play so the strategy gets prioritized. If there are objectives and the player reports progress, then it is not necessary to figure out new objectives and therefore all efforts can be put into minimizing the deviation from the path deemed best.

From a neurological point of view, objectives chosen so that following them is comparatively cheap can allow the strategy to run concurrently with the player itself. A perfect illustration of the strategies, objectives and players at work is the process by which we drive a car from point A to point B following directions. The strategy wants to go from A to B, and produces objectives such as turn here or there as appropriate. The player takes these objectives, adjusts for traffic, signals and a multitude of other factors, and moves accordingly. Should the player report missing an objective, then the strategy can adjust the objective queue as necessary. The worst case is getting lost, in which the objective could be either "drive at random until we are not lost"[13], or "ask somebody". It seems reasonable to conclude that cheap driving objectives allow a driver to meet them while running the strategy in the background, and that for the most part we seem to be able to do this without much trouble.

A request to "pay attention to what you are doing" should have acquired a new meaning now.

6.1.7 Achieving our goals

Finally, we get to the point where the objectives control the hands with the idea that some actions will cross the first distinction, this time going from the side of player into the side of environment. If our hands are well calibrated, then the prediction error corresponding to the intended effect will be small, and as such a new observation will not be surprising. This leads to a stable evolution of the player's trajectory in the information space it is embedded in, towards the attractor implied by the player's behavior.

To summarize, let's go over a typical iteration of a player. First, the perception part of the interface experiences its environment, and the raw information is passed over to the eyes, which according to the intentions of the player separate the raw perceptions into distinctions by looking for interesting differences in

will examine this further, but at least in principle, simple models that work are interesting because they are easy to deal with.

[13]The idea is that the random choices will be biased, and that hopefully their bias will resonate with a situation in which we know not to be lost, thus implicitly moving towards that attractor without describing it explicitly.

value. The resulting pattern of distinguished objects is passed to the strategy, which selects a number of objectives to meet as it sees fit. These objectives are passed to the hands, which in turn let themselves be controlled by the objectives by means of double dispatching. The actions fired by the objectives through the hands are queued at the action part of the interface, which at some point will hopefully cause a change in the environment. The action part of the interface perceives its surroundings again, and the cycle begins anew.

There are a number of fine details that we should consider. First, our own experience suggests that each of the parts of the perception pattern, namely the perception and action parts of the interface, the eyes, the strategy and the hands are independent of each other and run in their own "thread", so to speak. Second, when objectives control the hands it may be useful to synch up with the environment before having the hands perform the intended action. Thus, objectives may peek through the eyes in realtime waiting for the appropriate trigger before moving the hands as intended.

Neither of these is a major problem for our model. In fact, the perception pattern seems robust enough to take on real problems with ease. To make sure this is so before implementing it, let's take a look at it from a different point of view.

6.1.8 A more mathematical interpretation

In previous sections and chapters, we have been talking about information spaces, dimensions, maps between spaces, and even behaviors seen as functions that imply attractors. Now, it is time to take the metaphor a bit further in an attempt to formalize it.

Let's assume that the environment in which we live can be represented as some sort of space that looks like a vector space[14]. It is a big assumption to

[14]If you are not familiar with linear algebra, this would be a good time to go through some definitions. A *vector* is an ordered collection of elements. Vectors of size n are typically written as $(x_1, x_2, \ldots, x_n)$. A vector space V of dimension n over a field of numbers K is a set of vectors of size n whose elements are in K, such that these two conditions are met:

1. If $v, w \in V$, then $v + w \in V$, where $v + w = (v_1 + w_1, \ldots, v_n + w_n)$.
2. If $v \in V$ and $\alpha \in K$, then $\alpha v \in V$, where $\alpha v = (\alpha v_1, \ldots, \alpha v_n)$.

Note that from the definition above, the null vector of which all its elements are the null element of the field, must be in the vector space.

A linear function F from a vector space V to a vector space W is a function that takes elements $v \in V$ and maps them to elements $w \in W$, and that complies with the following two

make, but since we do not experience our environment directly, we do not have access to the true nature of the universe and as such the assumption cannot be disproven right away. As such, we will keep working under this premise as long as it helps us get to a more precise model of what we perceive. For naming purposes, we will refer to the vector space representation of the universe with the letter U.

As observers of U, we are inscribed in U. At the same time, we are also separated from U by our first distinction. To emphasize our most humble of positions, we will refer to our existence as an observer with the letter μ.

When μ experiences U, what μ actually perceives is a subset of U. In fact, it sees a *projection* of U. This corresponds to the perception process of the interface in the pattern of perception we discussed. In linear algebra and the world of vector spaces, this projection can be done with a linear function. Since this function is the one that allows μ to perceive, we will refer to the projection of U made by μ as $p_\mu(U) = P_\mu$. In particular, note that since by definition P_μ contains everything that μ can observe in the universe, it must also contain μ in as much as μ can experience itself. If you were μ, then P_μ would contain your thoughts, your memories, and your idea of who you are.

Once we have P_μ, we interpret these raw perceptions and produce distinctions. Note that since these distinctions can be experienced, they also live in P_μ, maybe in a particular subset of it. In any case, again there is a function that goes from

---

conditions:

1. For any $x, y \in V$, $F(x + y) = F(x) + F(y) \in W$.
2. For any $\alpha \in K$ and $x \in V$, $F(\alpha x) = \alpha F(x) \in W$.

Note that if x_0 is the null vector of the space in question, the conditions above imply that $F(v_0) = w_0$. From here it is not hard to see that the image of a linear function is also a vector space.

The operation of linear functions over vectors x can be seen in terms of the multiplication of a matrix the rows of which are vectors in V by the vector x. The composition of linear functions corresponds to the multiplication of matrices.

Two vectors x, y are called *linearly independent* if there is no $\alpha \in K$ such that $\alpha x = y$. A set of vectors $T = t_1, \dots t_k$ is linearly independent to x if there are no $\alpha_i \in K$ such that $x = \sum_i \alpha_i t_i$. For each vector space V of dimension n, there are sets of linearly independent vectors B of size n called the *bases* of V. Any vector in V can be written as a linear combination of the vectors of any basis of V.

The most natural basis of a vector space V is the set of canonical vectors e_i, which are equal to the null vector except at the position i in which they have the multiplicative null element of K. If K were the real numbers, then the multiplicative null element would be the number 1. The basis of canonical vectors is called the canonical basis. As you can see, a canonical basis makes finding the linear combination of any vector a trivial matter.

P_μ into the space of distinguished objects. We will call this function d, and the resulting space is thus $d_\mu(P_\mu) = D_\mu$.

At this point in the pattern of perception we thought of, the strategy takes objects in D_μ and produces objectives. Objectives, again, are in P_μ because they can be perceived. Therefore, once more, there is a function that produces objectives which we can express as follows: $o_\mu(D_\mu) = O_\mu$.

The hands take objectives and queue actions. These actions (which are not to be confused with their results) can be perceived, and so they are in P_μ. Since the objectives may look through the eyes to coordinate their activities, there is a function a that produces actions in the following manner: $a_\mu(D_\mu, O_\mu) = A_\mu$.

Finally, the action process of the interface takes the actions in A_μ and executes them, thus affecting U in as much as there is overlap between U and P_μ. We will call this last function e, and its operation is described best as $e_\mu(A_\mu) \to U$. The arrow represents the act of injecting modifications into U by crossing the first distinction of μ. Thus, we could write that p_μ does this: $p_\mu(U) \leftarrow U$.

Let's introduce the following refinement to our notation: we will use $\to$ to denote mappings between spaces inside the player, and $\mapsto$ to denote mappings that cross the boundary of the player. Thus, we can describe the workings of the pattern of perception we thought of in a single line.

$$U \mapsto P_\mu \to D_\mu \to O_\mu, \quad D_\mu + O_\mu \to A_\mu \mapsto U$$

There are a few key observations to make. One of them is that if each function can be expressed as a matrix, then one could simply consider the product of all these matrices, call the resulting matrix M, and then $M(U) \mapsto U$. It might even seem like a way to achieve some sort of efficiency. However, it also confuses things as now we cannot readily distinguish the matrices that make up M. Since we cannot distinguish parts, it will be harder to induce predictable change at low cost. Here is where again we see how our guiding principle that tells us to create a network of parts such that their relationship induces a desirable attractor provides tangible benefits.

Another observation to make is that this arrangement is meant to mimick our own thought processes. When will this run most efficiently and require the least calculation effort? When the basis of each space is close to being the canonical one!

This leads to the final and most critically important observation. Some basis canonicalization could occur because we observe things better, or because we find an easier way to do things. But there will be some times where, no matter how hard we try, we cannot make the bases close to being canonical. When

this happens, the symptoms include the presence of numerous special cases or exceptions, and a multitude of situations that are somewhat similar yet different enough so that it requires treating them as separate entities. Nothing is clear cut, nothing is easy to do. It may seem that we are extremely busy, but despite this our accomplishments are not as valuable as we would expect.

How do we address these situations? By untangling the linear combinations of our non canonical bases. And when that cannot be done in the space we are working in, then what we do is to *increase* the dimensions of the space in question so that each canonical vector of the basis corresponding to this larger space can represent a particular messy linear combination in the smaller space. In a way, we make the space larger to make enough room so that we can express ourselves more clearly, in the same way that using a richer vocabulary to communicate leads to using specialized words that stand in front of much longer expressions. We usually refer to this in terms such as *learning*.

What does this procedure of using the canonical vectors of a larger space to facilitate more powerful thought processes represent? The acts of naming, of drawing more refined distinctions, of chunking. It also leads to the extremely important conclusion stating that to make things easier, *sometimes what must be done is to add dimensions to the information space until the answer becomes obvious*.

An observer becomes aware of itself by adding a dimension to its information space, which can then be used to distinguish what is itself and what is not. Some computer languages such as Smalltalk are reflexive precisely because of this.

This may sound like magic at first, but we have already seen some examples of how it works. When we created one class per character, we increased the dimensions of our execution context expression space so that the run time answers required from them became obvious. When we designed an efficient reference finder, we again increased the dimensions of our expression space so that, by separating the responsibilities involved, the answer to each of the required bits of local behavior was obvious. Therefore, there is plenty of evidence that shows the design principle works, that actual programs designed with it in mind exist, and that its use has valuable consequences such as clarity, maintainability, flexibility, and execution efficiency of the resulting software.

Note how many observer reflections are at play here.

So, now we know precisely what we are talking about when we make reference to a pattern of perception. And, at this point, there is only one thing left to do. To ensure that our idea of how to design complex object systems actually works and that it is more than just wishful thinking, we will implement the pattern of perception we have been discussing itself in Smalltalk.

6.2 A perception pattern in Smalltalk

Once again, here we are facing the often dreaded situation. We know enough of what we want to tell if an implementation suits our requirements, but we have not written a single line of code yet. What do we do first? At this critical point, it will be most valuable to assess what could be the benefit of each of the actions available to us.

We are choosing between objectives, we are taking a look at the development strategy at work!

There is a school of thought that says the first thing one should do is to write code and, um, *see what happens*. It seems quite optimistic, does it not? You just pound some code into existence, perhaps *play with it* a bit (whatever that means), and by magic you arrive at the desired result. Well, it may be just fine to proceed as such, but I contend that this approach is best only when our predictions are awful because our hands are not calibrated yet. In other words, this is most helpful when our information space is so removed from reality that regardless of what we think will be the theoretical effect of us doing something, in practice we cannot match the prediction of our theories to actual observed phenomena. In these situations, we can use this technique to get our bearings right quickly. Indeed, we have heavily used a similar approach to find good selectors in previous chapters. However, in this situation, we already know our problem to a fine enough degree of detail. As such, we choose to *not write a single line of code*.

Well, you say, that surely cannot last for long. True enough. But what is important is what we will do *before* writing a single line of code, and why we will deliberately wait before typing in definitions for classes, messages and methods.

Our activity at this point will be to collect the objects we have distinguished, and to compare them looking for similarities at different levels of resolution. If possible, we should find an abstraction under which some of them are more or less the same. This is because doing so will make it easier to organize the implementation of our program. In addition to that, we will pretend the objects exist and that we debug a typical use case in our minds. The idea here is to see how these objects talk to each other. Conversations should be short and concise, without unnecessary chatter or roundabout communication traffic. We have seen good examples of a clean messaging patterns in this book already — for example, the reference finder we implemented is extremely quick not because it is hacked together, but rather because the objects involved do not talk more than they need to.

So that is what we will do before we write code. The reasons for this are so important that it is impossible to overstate how critical they are. What would happen if we wrote code without having a good estimate of what the effect will

be? Then, of course, we would have to change it. And here we must stop and take a hard look at what is going on. First, we do not know what we want to do because we cannot predict the effect of our actions, but we know enough of our problem that we should be able to predict with enough quality. In other words, we *did not spend time evaluating the consequences of our actions when doing so could have provided value.*

That, on its own, sounds less than optimum already. But there is more. Let's say that we go through a feedback cycle and change the code so it is better. And how do we actually accomplish the change? Well, because we are observers living in our environment, we do that *by crossing our first distinction*, right? What we need to look at here is the huge cost of crossing our first distinction too frequently. If we use our hands to type, then we are not thinking about our problem — we are just emptying our outbound queue of actions to execute in our environment. And it so happens to be that typing is not something we do extraordinarily fast. Try this experiment: think about a problem, but type every single thought process you go through. Do this for ten minutes. How painful is that? If you talk instead of writing then it may not be as bad, but it is still very slow compared to the real thing, is it not?

Therefore, to profusely type in order to describe frequent changes to computer programs is clearly tantamount to having an extremely low overall performance as developers. This is because the capabilities of our outbound interface with the environment are not those of an ethernet connection. They are more like those of a 300 baud modem without built in error correction. Thus, we should send data through this interface sparingly, reserving our precious CPU cycles to come up with a design that not only solves our problem but also minimizes the need to put traffic through such a constrained communication channel. What is more, we should further tweak our design so that it allows *anybody else* to make efficient use of their outbound connections as well.

Therefore, the productivity of a developer should never be measured in terms of how much code is typed.

As you can see, the problem of how to implement programs efficiently in a computer language is very similar to a register allocation exercise in which we run the fastest when most of the information stays close to our 7 ± 2 scratch RAM slots. In this context, typing a lot is like thrashing the page file. We must learn to avoid this at all costs. If we do not, were somebody to run a time profiler on our development process would find that we spend 50% or more of the time in IO wait. Is that the mark of efficiency now?

So, given these constraints, how do we become really good at efficiently using our capillary wide outbound communication pipe? First, by training ourselves to withhold the emission of code until enough design iterations in our mind find code

that remains stable over time. Then, and only then, we should implement the pieces that have become really close to their attractor. The more we practice this skill, the more benefit we will obtain from it. We can also improve our efficiency as developers by choosing a computer language that requires us to type a minimum of syntax sugar and comply with the least possible process rituals. Smalltalk ranks very well in these regards, and this is why it is such a valuable vehicle for human expression.

Given these considerations, let's go through a round of training by looking at all the objects we have identified in the pattern of perception we have been talking about, and see if we can find similarities.

To make our discussion easier, let's adopt the following convention first. As you have seen, we have been referring to the pattern of perception as the pattern we have been discussing, or the pattern we thought about, and so on. This is because we should be keenly aware that, as much as it may seem like a correct approach, the product of our work is just our interpretation and as such it is extremely likely to be correct only up to a certain point. However, to qualify the pattern we have been describing all the time causes us to incur in needless repetition. Thus, from now on, we will not explicitly mention the fact that we keep in mind that the pattern of perception discussed so far is just an arbitrary way to describe something we perceived.

6.2.1 Characterization of the parts

The pattern of perception consists of a player that has an interface with an incoming and outbound pipe, a set of eyes, a strategy which produces objectives, and a set of hands. Apparently, there is not a lot to abstract here because each of these objects has a distinct, well defined responsibility.

Or is there something under the surface? Because let's see here, the eyes for example take information from one space and produce information in another space. The strategy takes the output of the eyes and produces information in the space containing the objectives. The hands do something similar. Both pipes of the interface do the same. So actually, these objects have a lot in common. And what is more, we have already made an abstraction that applies to all of them, remember?...

Every single object we have been talking about is a behavior process!

Ah, but then what we should do now is to model behavior processes. Once we have them stabilized in our minds and then in code, implementing the rest of

This onion is full of layers!

the parts should be more straightforward[15].

As we saw before, a behavior process is a function embedded in an information space. The function takes a position vector from the space it is embedded in, chooses a direction in which to move, and produces a vector which represents how much it wants to move.

A behavior is a vector field over a vector space.

Should we explicitly model an information space and position vectors? For now, the answer is no. Doing so could cause us to translate the language of our domain into an abstraction capable of holding anything we throw at it, and as a consequence we run the risk of losing some of the ability to concretely describe our problem. Thus, while we will not create classes such as `InformationSpace` at this time, we will however model everything else to be a good reflection of them.

Do you remember the discussion, a few chapters earlier in the book, about whether behavior processes travelling in an information space end their existence because they finally evaluate or because of an unhandled exception? That is one of the details we have to consider now. Fortunately, it is not as thorny as it seems. In Smalltalk, an unhandled exception is a problem, and as such it should not happen. Because the context in which we have to make this design decision, we have no choice other than to assume a model in which standalone behavior processes *do evaluate*.

See how easy that was? Scary, huh?

Also, did you notice the distinction we just made? We said that *standalone* behavior processes will evaluate to something. But what would happen if the behavior process is not standalone, meaning that it does not correspond to a Smalltalk process? What does that mean to begin with? Well, one interpretation could be that the player queries its behavior processes because its design goals ask for a specific composite behavior which is best induced by controlling the sequence in which each of the behavior processes behave more tightly. This means that our design has to accomodate multithreaded and singlethreaded players.

At first this may seem like an overblown requirement, but I am sure you will be able to think of instances in which you behave one way or the other. As the point of the pattern of perception is to allow us to model our own behavior, then this level of flexibility is truly a desirable feature to have.

Nevertheless, it may feel as if it will cause lots of code to appear. And yet, that just does not need to be the case. A forked Smalltalk process running in the behavior process could simply reuse the code that allows the player itself to query the behavior process in the first place, so this should not be hard to do.

What are we going to do with the information space in which the behavior

[15]Note here how the principle of not writing code right away is at work.

process moves? We said we do not want to explicitly show that we are thinking about information spaces because that would allow too much of an arbitrary abstraction to shine through. So now we need to hide that a bit without lying too much. How are we going to tackle this? Well, we could talk in terms we can expect others to be very familiar with. For example, what Smalltalk construct is similar to what we want to do? Where in Smalltalk does it happen that there is a map between some input objects and some output objects? The point of a behavior process is to implement behavior, right? So what is the Smalltalk construct that allows behavior to occur?...

Of course! In Smalltalk, *the only thing we can do is to send messages!*

Therefore, we will use the word `sender` to implicitly talk about the source of position vectors, and the word `receiver` to talk about whoever receives the displacement vector from the behavior process[16].

What other names will behavior processes have to know? Well, in order to make housekeeping easier, there should be an instance name for any embedded process running inside a behavior process. Borrowing from the circuit analogy we discussed earlier in the book, we will call this process the `signal`. To be able to stop a signal gracefully, we will also let behavior processes know whether their `signalShouldEvaluate`.

We have been discussing behavior processes for quite a while now, and we have finally reached a point in which it looks as if our description of them is stable enough. There does not seem to be anything obviously missing from our previous descriptions, and while it is not as direct as it could be, it seems rich enough to capture anything we throw at it — albeit perhaps doing so indirectly. Thus, let's create our first class:

```
PerceptionBehaviorProcess
  (sender receiver signal signalShouldEvaluate)
```

All the instance names will have accessors for them, but only the getters for the objects named `sender` and `receiver` should be public. We should also make sure that behavior processes start up initialized already, and so we will go through the `super new initialize` motions and provide an `initialize` instance method such as the one below.

[16]This looks almost too easy. But consider the many pages and the amount of effort we had to go through to reach this place! It just seems simple now because it is becoming obvious that the amount of code we will have to write is very small — which is *exactly* the point of not writing code right away.

```
PerceptionBehaviorProcess>>initialize

  self signalShouldEvaluate: false
```

What messages should behavior processes understand? This one suggests itself:

```
PerceptionBehaviorProcess>>behave

  ^self
```

Note that we did not do `self subclassResponsibility`. This is because we are assuming that a missing implementor of `behave` will be extremely easy to detect because neither the behavior process nor its embedding composite will behave as intended.

Let's teach behavior processes how to spawn an embedded signal now.

```
PerceptionBehaviorProcess>>spawnSignal

  ^[
    [self signalShouldEvaluate] whileFalse:
      [
        self behave.
        self wait
      ]
  ] fork
```

Notice the message `wait` being sent after `behave`. If we did not actually wait or at least yield to the next process, such a block would hog execution resources. Thus, the default implementation of `wait` is as follows.

```
PerceptionBehaviorProcess>>wait

  Processor yield
```

Note that refined implementations may choose to do something else such as waiting on a delay, i.e.: `(Delay forMilliseconds: 100) wait`. This decision will be left to the particular application using the pattern of perception.

To avoid the creation of multiple embedded signals, we will mark the message `spawnSignal` as private and use a protected mechanism to fork the behavior

process on its own Smalltalk process. Now, what do behavior process objects do from a Smalltalk point of view? Well, they observe their position and decide to move accordingly. Therefore, the following message suggests itself.

```
PerceptionBehaviorProcess>>observeIndependently

  self signal notNil ifTrue:
    [self error: 'Process is already observing'].
  self signal: self spawnSignal
```

But now, what will happen if we spawn a process, and then tell the behavior process that its signal should evaluate? Then `signal` will refer to a dead process, not `nil`. If we decide to tell the behavior process to start observing again later, it will complain on the grounds that the corresponding instance name does not point to `nil`. Thus, we amend `spawnSignal` as follows.

```
PerceptionBehaviorProcess>>spawnSignal

  ^[
    [self signalShouldEvaluate] whileFalse:
      [
        self behave.
        self wait
      ].
    self signal: nil
  ] fork
```

You will remember that the accessors for `signalShouldEvaluate` are private. Thus, how will we tell behavior processes that their signal should evaluate? By using messages to rename private names into proper public names according to the metaphor we want to expose. We do this for two reasons. First, to suggest to those interested in the reasons behind the implementation that there are deeper design principles at play. And second, to offer a public interface that is consistent when viewed from outside. Thus, we do as follows:

```
PerceptionBehaviorProcess>>unwind

  self signalShouldEvaluate: true
```

Note that, in this way, we also avoid keyword messages for free. Finally, how are we going to tell a behavior process what are its `sender` and `receiver` supposed to be? Since we already have `observeIndependently`, we could take that a step furter and implement something like this:

```
PerceptionBehaviorProcess>>
  observeAccordingTo: anIntention
  andDistinguishFor: aBehavior

    self sender: anIntention.
    self receiver: aBehavior
```

Note how once more we rename things to hint at what is really going on. While in this particular case some could argue that it may be a bit too much, I find it hard to resist the beauty of telling a behavior process to *observe according to an intention, and distinguish objects for another behavior*.

Well, it may seem puzzling, but the few messages we have just gone through are the complete implementation of `PerceptionBehaviorProcess`. Now we can look at other behavior processes, such as the eyes and the hands, in terms of our more general abstraction. But to model them properly, first we will have to see how they interact with each other.

6.2.2 Relationship between the parts

The first part we distinguished is the player itself, because it was created by drawing the first distinction. Inside this region, we find the behavior processes we mentioned: the eyes, the strategy, the hands, the in and out pipes of the interface, and the objectives.

How do all these objects interact together in a reasonable manner? First, there has to be some setup that allows each of the behavior processes to know which objects are its senders and receivers. Since the player has a context in which all of these parts are visible, then it is natural for this arrangement to be the responsibility of the player.

And what arrangement should this be? This is a critical section, and some of the answers we will reach may surprise you.

First, the interface's perception process can be characterized as being passive in the same way our retinas are passive. We can choose whether to pay attention to the values coming from them, even with our eyes open. Therefore, the interface does not observe according to any intention, and does not distinguish objects for

anything else. This leads us to conclude that although the perception and action processes may be behavior processes, the interface itself is not.

By clustering the perception and action processes inside an interface object, we get the benefit of creating a single place to hold potentially repetitive code in. Because this execution context exists, we can just move all the boundary crossing code to the interface while having the perception and action processes reuse that functionality which will now be implemented only once. In addition, this allows similar interfaces to share code thus avoiding further code duplication. It also follows that the perception and action processes must know their interface by instance name.

The eyes come next. Clearly, their sender is the interface because they observe the raw perceptions looking for differences in value according to their intention. Who they provide the found distinctions is not so clear. At first it would seem this object should be the strategy. But let's say that our player will have to play a number of different variations of the same game. How can it know the strategy to use before the eyes distinguish the game being played?

So in reality, the eyes distinguish objects for the player, not the strategy. This allows the eyes to tell the player which strategy to use according to the situation they perceive. Another observation that supports this design decision is that sometimes we will close our eyes when thinking hard about about which objectives we should choose, with the understanding that further observation is unnecessary. Thus, the strategy will pull observations from the eyes only as they are needed.

The strategy marks the boundary between the pull and push approaches used in this pattern, because it will pull objects from its sender and push objectives to its receiver. Obviously, the senders of the strategy are the eyes, and the receivers are the hands.

As we discussed, the hands let the objectives take control and push commands to execute. Thus, it is straighforward to let the receiver of the hands be the interface. The sender is not the strategy, however — the senders of the hands are the eyes. This is because at times objectives will pull data from the eyes until some condition is met before issuing commands.

A perfect example of this is how you start your car: you turn the ignition key and leave it turned until you hear the engine turning fast enough. The objective is to start the car, the command is to stop turning the key, and the trigger the objective is looking for is a particular perceived sound.

This completes our relationship overview. The senders and receivers for the eyes, strategy and hands are summarized in the table below.

Behavior process	Sender	Receiver
Eyes	Interface	Player
Strategy	Eyes	Hands
Hands	Eyes	Interface

Now, it is time to write code.

6.2.3 Implementation

Perception interface, perception process

Let's begin with the interface processes. Since these two behavior processes will know the interface by instance name, we can create an abstract superclass for them so we can put shared code in only one place. Thus,

```
PerceptionBehaviorProcess
  PerceptionInterfaceProcess (interface)
```

The accessors for `interface` will be private. Now, let's consider the perception process as a subclass of the interface process class[17],

```
PerceptionBehaviorProcess
  PerceptionInterfaceProcess (interface)
    PerceptionPerceptionProcess
```

From our discussion earlier in the chapter, we are working under a model in which raw perceptions are interpreted on demand once every so often. Thus, we have two concerns to address. First, that the perception process should not perceive all the time because otherwise it would take over computing resources that might be better spent somewhere else. Second, if we make the perception process perceive once in a while, how will the eyes know if a new perception is available?

We can address the first of these two issues for the case in which the perception process is running on its own thread by implementing the following messages in `PerceptionInterfaceProcess`.

[17]The class name prefix does not help — check out the exercises.

```
PerceptionInterfaceProcess>>delay

  ^Delay forMilliseconds: (1000 / self frequency) ceiling


PerceptionInterfaceProcess>>frequency

  ^100


PerceptionInterfaceProcess>>waitUntilNextCycle

  self delay wait
```

With these simple messages, if particular subclasses decide that `wait` should do more than just `Processor yield`, they can easily refine `wait` to do `self waitUntilNextCycle`.

Note that this approach allows us to avoid sending `ifTrue:ifFalse:` in `wait`, such as shown below:

```
PerceptionInterfaceProcess>>wait

  self shouldWaitBeforePerceivingAgain
    ifTrue: [self delay wait]
    ifFalse: [Processor yield]
```

This is not as good as it could be for a number of reasons. First, of course, we have an unnecessary `ifTrue:ifFalse:`, and the consequence is that specific behavior processes will be implemented in a way that will not take full advantage of their own identity. We have seen what comes out of an excess of `ifTrue:ifFalse:` before, so avoiding that situation is desirable. The second reason is more subtle, but has a significant impact nonetheless. What would be the implementation of the message `shouldWaitBeforePerceivingAgain`? To answer an instance name, the existence of which adds to the complexity of our objects? To answer a boolean, causing `self subclassResponsibility` in the abstract superclass? Or perhaps to implement it as `self signal notNil` in which case knowledge about the implementation details are spread farther than need be?

None of these is better than simply writing the obvious code for the particular context. Thus, we will provide `waitUntilNextCycle` and let concrete subclasses

spell out explicitly what is their behavior supposed to be.

We addressed the first of our two issues, how to avoid perceiving all the time, for the case in which the perception process is running on its own thread. If the perception process is not running on its own thread, however, then our model dictates that the frequency of perception is the responsibility of the player. As such, no further code is needed in this context.

Note that this frequency approach is also available for the action process. When needed, this functionality can be used freely while imposing a minimum cost for the change in behavior.

Now we can over the second issue, how to let the eyes know that a new perception is available. We could let the eyes remember the previous perception and compare it with the one the perception process has. However, what if the new perception is the same as the old one? The comparison would not be able to distinguish that particular bit of information. To make it worse, in some cases the comparison could be expensive.

Let's consider other solutions. Perhaps we could have the perception process watermark or timestamp the perceptions somehow, and then we could have the eyes look at this extra bit of information. However, modifying the raw perception violates the design principle that states that at this point perceptions are not interpreted. Perhaps we could have the perception process trigger an event, and have the eyes record the ocurrence of the event in an instance name, but multiple threads would cause semaphores and rather elaborate code to appear.

Overall, these solutions are too complicated. What is more, they are not solutions to the problem at hand. If we ponder this situation for a while, we will see that the bit of information we need to distinguish is *when the perception process perceives again*, instead of *the current raw perception is new or old*. This is also consistent with leaving the raw perceptions uninterpreted.

So how do we signal when the perception process has perceived again? If we use a boolean, then we will need to synchronize the access to the variable in case the eyes look at it at the wrong time. If we go down that path, we will find the same solutions we just discarded a couple paragraphs above. We need to think differently.

What did we say above, when we described what we need to distinguish? That we need to indicate *when the perception process perceives again*. Perhaps we can massage this enough so that it will yield a good solution. First, that *again* at the end is superfluous: if we perceive again, we could perceive the first time, so we can perceive once[18]. Let's take it out and try again.

[18]Remember the `continue:...` selectors in the chapter about `match:`?

Now we have to indicate *when the perception process perceives* once, twice, many times. Once, twice, oh... perhaps the amount of times is the key. Clearly, the perception process will perceive one time after another. Thus, we make the obvious observation that states there is a sequence of perception events. What we want is for each of those events in the sequence to be distinguishable from one another. But if we distinguish then we can name. Therefore, and inevitably, each event must have a distinct name[19].

What we have to decide now is what naming scheme we will use to name perception events. We can reuse the fact we have been talking about the sequence of perception events, and use sequenceable names such as the integers. How proper to use integers this way!

Now that our ideas about the perception process seem to be stable, we can implement the behavior of the `PerceptionPerceptionProcess` class. First, we will add an instance name `perceptionCount`, with a public getter and a private setter, and a suitable `initialize` method such as the one below.

```
PerceptionPerceptionProcess>>initialize

  super initialize.
  self perceptionCount: 0
```

What is the behavior of a perception process? To update its old raw perception with a new one. Therefore, and leaving the door open for perception processes that behave in ways we cannot imagine right now,

```
PerceptionPerceptionProcess>>behave

  self updatePerception
```

To avoid code duplication, we can let `updatePerception` do nothing other than updating the perception counter.

```
PerceptionPerceptionProcess>>updatePerception

  self perceptionCount: self perceptionCount + 1
```

By careful use of `super` in refined implementations of `updatePerception`, we

[19]Note that naming a perception event is *not* the same as timestamping the resulting raw perception.

also get proper semaphoring for the eyes free of charge.

This rather tiny amount of code completes the implementation of perception processes. Next part, please!

Eyes

The basic job of the eyes is to provide the strategy with an information field so that it can decide which objectives to go after. Thus, the implementation of the eyes revolves around the management of the information field distinguished from the raw perceptions provided by the perception process in the interface.

What instance names will the eyes need? Since we just discussed the name `perceptionCount` for the perception process, it is easy to see that we will need an instance name in the eyes so they know what is the latest perception event they interpreted. Thus, we will reuse the same name and add an instance name called `perceptionCount` to the eyes. We will also initialize it in this way:

```
PerceptionPlayerEyes>>initialize

  super initialize.
  self perceptionCount: nil
```

Technically speaking, we do not need to initialize an instance name to `nil`, but being explicit here has some value. Imagine looking at the method that checks if the value of `self perceptionCount` is the same as the value of `self sender perceptionCount`, without first seeing that the value in the eyes was initialized to `nil`. Would not it be tempting to initialize it to zero to avoid a seemingly unintended comparison with `nil`? To avoid this, we choose to write down our intention to let the initial value of `perceptionCount` to be undefined. We really mean that in this situation, and thus we explicitly express it so.

The eyes inherit the instance names `sender` and `receiver` from the abstract class of behavior processes, but to make things easier for us we will rename these names via local accessors. This renaming is done to customize the meaning of general names to the context in which they will appear. Thus, we write:

```
PerceptionPlayerEyes>>interface

  ^self sender
```

```
PerceptionPlayerEyes>>player

  ^self receiver
```

Note that this technique, which can be used to make it easy for code to be written with different degrees of resolution in the same class hierarchy, works particularly well in this case because we do not need customized setters. And regardless of whether we decided to implement specialized setter messages or not, it is yet another perfect example of how message selectors can be used to name (or rename) things.

Now, the strategy consumes information fields from the eyes. However, does it need to care about whether the perception is current or not? The answer is no, because it is preferable to let the eyes deal with their own bit of management, rather than to expose other objects to unnecessary detail. Besides, all that the strategy needs is the information field. Thus, we will let the strategy ask for that by having it send the message `field`, with the understanding that whatever is given to the strategy can be assumed to be current.

The message `field` matches the instance name behind it, so in this case the accessor will have to provide further functionality behind the scenes. This provides us with a convenient bite size first domino from where to build upon our implementation of the eyes. Let's write the method for this message and see where that takes us.

```
PerceptionPlayerEyes>>field

  ^self perceptionCritical:
    [
      self privateEnsurePerceptionIsCurrent.
      self privateField copy
    ]
```

What are the reasons behind this implementation? First, there is the need for a mutual exclusion semaphore because both the strategy and the hands have the eyes as their senders. What the mutex semaphore ensures is that, in the case when multiple processes are observing independently sending the message `field` on their own, only one thread will update the field at the same time. To provide this functionality, we simply provide an instance name which we can conveniently call `perceptionSemaphore`, and initialize it in this way:

```
PerceptionPlayerEyes>>initialize

  super initialize.
  self perceptionCount: nil.
  self perceptionSemaphore: Semaphore forMutualExclusion
```

Let's go back to the implementation of `field` now. Why is it necessary to use the message `perceptionCritical:`? Well, the thing is that if we did not have this message, then we would have to write `self perceptionSemaphore critical:`, and as it turns out we would have to write it many times. Instead of doing so, we simply provide a shortcut message that does that for us. An indirect benefit is that it also gives us a quite convenient spot where to put a single breakpoint if we need to debug a problem.

If we pause for a moment, here we can see that creating messages like the one above not only helps us by naming or renaming things — it also *creates* locations in the network through which the behavior signal represented by a Smalltalk thread travels. By carefully crafting the smallest of execution contexts, we can make our code easier to read and easier to debug. Please note how these smallest of changes, at practically negligible costs to the developers, can have such a profound impact later on.

So, inside the critical block in the implementation of `field`, we ask the eyes to actually ensure the perception is current, and then we answer a copy of the perceived field by using a private getter for `field`. But why the copy? Because we must allow other Smalltalk processes to look at a field snapshot which is left undisturbed by the activity of the process embedded in the eyes. When the eyes update the field, they could do just that: perform a potentially destructive update on an existing object. Since this is not thread safe, then we make the eyes answer copies of the field. In this way we mimick the situation we described in our earlier modelling discussions, the one in which our minds take a snapshot of what we perceive and pretend that nothing changes for a while even though we may physically perceive otherwise.

Therefore, and along these lines, the implementation of `privateField` is straightforward:

```
PerceptionPlayerEyes>>privateField

  ^field
```

The implementation of the message `privateEnsurePerceptionIsCurrent` is quite simple as well:

```
PerceptionPlayerEyes>>privateEnsurePerceptionIsCurrent

  self perceptionCount = self interface perceptionCount
    ifTrue: [^self].
  self newPerceptionAvailable
```

Note how clearly stating that `perceptionCount` has an undefined initial value in `initialize` makes it easier to look at the method above from a more refined point of view. Also, the use of the renaming message `interface` makes the code to be more clear because then we do not have to remember who the `sender` is in the current context. In fact, the senders of `sender` and `receiver` will be the renaming messages, and so the information in the table at the end of the last section is explicitly present in the code and available upon request. Next, let's take a look at `newPerceptionAvailable`.

```
PerceptionPlayerEyes>>newPerceptionAvailable

  self perceptionCount: self interface perceptionCount.
  self interface couldNotPerceive ifTrue: [^self].
  self updateFieldFrom: self interface
```

The first line of the method above is self explanatory. But why the second one, where did that come from? Well, sometimes, interfaces will be able to tell if the connection with the environment is broken. If this were to happen, then it would be useless to update the field. We behave similarly when we purposefully close our eyes and ignore the transient shapes and colors. Finally, since the actual interpretation behavior will be provided by more concrete subclasses, the default implementation of `updateFieldFrom:` is just to answer `self`.

```
PerceptionPlayerEyes>>updateFieldFrom: anInterface

  ^self
```

Now, all this is great, but we are missing a very important detail. What is the implementation of the message `behave` supposed to be for the eyes? Since the strategy is pulling updated fields, at first it seems as if the eyes have nothing

to do on their own. But that is not so. Two of their responsibilities are yet to be addressed at this point. The first one is to tell the player which strategy to use. The second one is to tell when the player should stop playing because e.g.: the game is over.

Let's assume for a moment that a player does not need to change strategies, and that determining the strategy once is enough[20]. If the player is controlling how behavior processes interact together, then it is the responsibility of the player to tell the eyes to provide information about which strategy should be used at the right time. To support that, we provide the following message.

```
PerceptionPlayerEyes>>setPlayerStrategy

  ^self
```

When each behavior process is observing independently on its own Smalltalk process, however, it is the responsibility of the eyes to tell the player which strategy to use. Therefore, we refine `observeIndependently` as follows.

```
PerceptionPlayerEyes>>observeIndependently

  self interface ensurePerceptionAvailable.
  self setPlayerStrategy.
  super observeIndependently
```

That leaves just the implementation of `behave` to go. Since in this context the goal is detect when the game is over, roughly speaking, then we could do as follows.

```
PerceptionPlayerEyes>>behave

  self shouldUnwindPlayer ifTrue: [self player unwind]
```

We can use the default implementation of `shouldUnwindPlayer` below to get things going, with the understanding that concrete subclasses will be able to

[20]If this is not the case, then we can let the eyes distinguish which game is being played, and then we could let the strategy be a composite of strategies which are selected by observing the relevant bits of the perceived field. In other words, there would be different strategies depending on the level of resolution implied by the point of view from which the field is being looked at. In any case, note that even in this scenario the eyes still need to let the player know that the strategy should be the composite one.

refine it as necessary.

```
PerceptionPlayerEyes>>shouldUnwindPlayer

  ^self signalShouldEvaluate
    or: [self interface couldNotPerceive]
```

Finally, we will review the implementation of `unwind` later, when we implement the player object.

The implementation of the eyes is now done. For the sake of completeness, here is a brief description of the class shape.

```
PerceptionBehaviorProcess
  PerceptionPlayerEyes
    (field perceptionCount perceptionSemaphore)
```

Strategy

Let's tackle the strategy behavior process now. In the same way we did for the eyes, we will provide renaming messages for the `sender` and the `receiver`, like this:

```
PerceptionStrategy>>eyes

  ^self sender


PerceptionStrategy>>hands

  ^self receiver
```

If the strategy is running on its own Smalltalk thread, then the amount it needs to wait for a new perception to become available is exactly the amount of time the eyes need to wait for a new perception to become available. Thus,

```
PerceptionStrategy>>wait

  self eyes perceptionDelay wait
```

Finally, we will provide an additional message so subclasses can refine it at will.

```
PerceptionStrategy>>observeField

  ^self
```

We will not provide any functionality here because we do not know how strategies will deal with their fields a priori.

Well, the implementation of the strategy was simple enough. It looks like we are going downhill now. Or is it just the nice feeling of being at the summit? Let's find out — on to the next part!

Hands, objectives

Since the strategy pushes objectives into the hands, the hands will need to have some sort of collection where to keep the objectives until they are actually needed. To provide thread safe access to this collection, we will need a mutex semaphore. Thus,

```
PerceptionBehaviorProcess
  PerceptionPlayerHands
    (objectives objectiveSemaphore)
```

Only the getter for `objectives` will be public. The remaining three accessors will be labelled as private.

Now, regarding the collection to store objectives, it looks like an ordered collection is a good choice to provide the objective queuing functionality required. Therefore,

```
PerceptionPlayerHands>>initialize

  super initialize.
  self objectives: OrderedCollection new.
  self objectiveSemaphore: Semaphore forMutualExclusion
```

This situation is very similar to the one we encountered with the eyes. We will need the semaphore critical shortcut message, and we will also have to rename the abstract instance names to give them a meaning closer to the concrete context in which they will be referenced. Therefore, we simply follow the same principles we used before.

```
PerceptionPlayerHands>>objectiveCritical: aBlock

  self objectiveSemaphore critical: aBlock


PerceptionPlayerHands>>eyes

  ^self sender


PerceptionPlayerHands>>interface

  ^self receiver
```

Now we should provide thread safe messages via which the strategy can push objectives into the hands. Typically, it will be the case that each strategy run will produce objectives from scratch, and as such it will be convenient to have the hands clean up their objective queue and simply accept new ones. As such, we do as follows.

```
PerceptionPlayerHands>>newObjective: anObjective

  self objectiveCritical:
    [
      self objectives removeAll.
      self privateAddObjective: anObjective
    ]


PerceptionPlayerHands>>newObjectives: aCollection

  self objectiveCritical:
    [
      self objectives removeAll.
      aCollection do: [:each | self privateAddObjective: each]
    ]
```

While this is very straightforward, why is it that objective addition is handled with such care? Well, since the objectives will have to control the hands, they will need to know the hands object at some point. We could pass `self` as an

argument to them at the appropriate time, but depending on how much the objectives have to do, this might end up looking like this:

```
PerceptionObjective>>traverse: aField andControl: someHands

  "statements making reference to someHands"
```

Now, if we refactor the imaginary code above, we see that we will either have to pass `someHands` around with at least some of the messages we send, or that we will have to remember it in an instance name so that the other bits of the objective implementation can make reference to them. One way or the other, the name `someHands` just gets in the way when we pass it as an argument. The exact reason is that, when passed this way, it explicitly increases the size of the context needed to understand the code. Why take such a convoluted approach? Instead, we will let the objectives know the hands via instance name, and we will let them know the identity of the hands they will control upon addition to the objective queue. Therefore,

```
PerceptionPlayerHands>>privateAddObjective: anObjective

  anObjective control: self.
  self objectives add: anObjective
```

In this way, when we implement concrete objectives, we will be able to make reference to the hands object via a getter which will only appear where it is truly necessary.

Let's implement the behavior of the hands now. The idea is that the hands iterate through the collection of objectives and let each of them control the hands in turn. But... should all objectives get a chance to control the hands at once? Some problems may need the first objective to be executed first, while some others may need all the objectives to run in one go. Since we cannot make a decision on that at this time because we do not know what future requirements will be, we will implement `behave` so that it is easy for different implementations of the hands behavior process to do as they wish.

```
PerceptionPlayerHands>>behave

  self objectiveCritical: [self criticalBehavior]
```

For the sake of completeness, we can provide a sample implementation of the message `criticalBehavior` such as the one below.

```
PerceptionPlayerHands>>criticalBehavior

  self objectives isEmpty ifTrue: [^self].
  self objectives first traverse: self traversalField.
  self objectives removeFirst
```

Note the care taken not to remove the first objective from the objective queue until *after* the traversal is done. This is the same debugger friendly approach we used when implementing the reference finder. Once again, this implementation technique allows easier debugging of the movement through the traversal field because by carefully designing the implementation of `criticalBehavior` we can more easily restart the message in the debugger without losing the first objective.

The implementation of `criticalBehavior` makes reference to the message `traversalField`. At this point there is not enough information to provide a concrete enough implementation, and thus we will just make it answer `self`.

```
PerceptionPlayerHands>>traversalField

  ^self
```

Finally, we should refine the implementation of `wait`. At first it may seem that we have to guess how much the hands should wait, but we can delegate the specification in the same way the eyes deferred to the interface. Thus,

```
PerceptionPlayerHands>>wait

  self actionDelay wait


PerceptionPlayerHands>>actionDelay

  ^self interface actionDelay
```

There is one message missing in our abstract implementation of the hands. We will need to let the interface know that it should queue a command coming from the objectives in some way. To avoid having each objective having to talk to the interface by themselves, thus polluting their implementation with unnecessary

details and making them more complex than necessary, we will let hands provide a default way to queue commands.

```
PerceptionPlayerHands>>queueCommand: anObject

  self interface queueCommand: anObject
```

We will review the implementation of this and other interface messages in the next section.

This completes the implementation of the behavior process corresponding to the hands. Since while writing the behavior of the hands we ended up sending messages to objective objects, we have actually specified some of the behavior objectives should have. Thus, this is a good time to complete the implementation of objective objects.

Since objectives do not inherit from `PerceptionBehaviorProcess`, we will make them a subclass of `Object`. We know that they will have an instance name for the `hands`, and it may also be convenient to add an instance name for the `traversalField` for those objectives whose behavior implementation is complex enough that it may be a good idea to avoid passing the field around as an argument. Therefore,

```
Object
  PerceptionObjective (hands traversalField)
```

We will let the getters of the instance names above to be public, while we will make the setters private.

Since there is nothing to initialize, at first it would seem that going through the `super new initialize` motions is not worth it. However, specific objectives may have different intentions, and being able to send `super initialize` without having to remember that `Object` does not provide a default implementation seems like leaving homework behind. Therefore,

```
PerceptionObjective>>initialize

  ^self
```

Going back to the instance names though, hmmm... we made reference to the message `control:`, not `hands:`, when we implemented the hands object. That suggests that there is some instance name renaming going on.

```
PerceptionObjective>>control: someHands

    self hands: someHands
```

Indeed, and again note how the availability of renaming accessors makes their usage more clear in the context they appear. Sending `hands:` from the hands object would not reveal the full intention of letting the objectives take control, and certainly having an instance name called `control` in the objective objects would be confusing at best. Also note that `control:` is a public message.

Finally, since we do not have enough information to tell how objectives will choose to traverse their traversal field, we will provide a default implementation of `traverse:` with the intention that concrete subclasses will refine it as they see fit.

```
PerceptionObjective>>traverse: aField

    ^self
```

Note that, in this way, we can send `super` from the subclasses without causing side effects, and that we also leave ourselves a single place to put a breakpoint if we want to debug the implementation of multiple objectives.

This completes the implementation of objectives. Now, let's implement the remainder of the pattern of perception.

Perception interface, action process

Whether it has an embedded Smalltalk process or not, the action process will receive commands and queue them up for execution. But we have seen the pattern before...

```
PerceptionInterfaceProcess
  PerceptionActionProcess (commands commandSemaphore)
```

Yes, there will be another ordered collection to serve as a queue, another mutex semaphore to allow thread safe operation, and protected messages to get access to the commands queue. Note that in this case, no accessor will be public. Since we have seen this a number of times already, I am sure you can almost predict the methods below by now.

```
PerceptionActionProcess>>initialize

  super initialize.
  self commands: OrderedCollection new.
  self commandSemaphore: Semaphore forMutualExclusion


PerceptionActionProcess>>commandCritical: aBlock

  self commandSemaphore critical: aBlock
```

What may not be as obvious is the implementation of the message `behave`.

```
PerceptionActionProcess>>behave

  | nextCommand |
  [
    nextCommand := self nextCommand.
    nextCommand notNil
  ] whileTrue: [self sendCommand: nextCommand]
```

Why is there no critical protection in the method? Because if there was a large backlog of commands to execute, and if each command took a long time, then the hands' embedded process would not be able to add commands until the queue was emptied. Thus, the critical sections are protected differently in this case.

```
PerceptionActionProcess>>nextCommand

  ^self commandCritical:
    [
      self commands isEmpty
        ifTrue: [nil]
        ifFalse: [self commands removeFirst]
    ]
```

Note how the criticality arrangement makes it harder to write this code in a more debugger friendly manner.

The message that allows hands to queue commands also needs to be thread safe, thus:

```
PerceptionActionProcess>>queueCommand: anObject

  self commandCritical: [self commands add: anObject]
```

The message above and `behave` are the only ones which need to be public.

In the implementation for `behave`, we sent the message `sendCommand:`, the method for which is below.

```
PerceptionActionProcess>>sendCommand: anObject

  self interface sendCommand: anObject
```

This completes the implementation of the action behavior processes. You will have noted, however, that both the perception and action processes act as buffers or queues which then delegate their actual boundary crossing behavior to the interface object. This tells us that the time to go over the implementation of the environment interfaces has come.

As we discussed before, while the interface object has a perception and an action behavior process, it is not a behavior process by itself. Therefore,

```
Object
  PerceptionEnvironmentInterface
    (perceptionProcess actionProcess)
```

Most of the interface's behavior is about delegating to the right behavior process. The exception to this is the boundary crossing implementation, which we can leave defaulted to do nothing. Thus, let's write the following methods.

```
PerceptionEnvironmentInterface>>responseToPerceptionRequest

  ^nil


PerceptionEnvironmentInterface>>sendCommand: anObject

  ^nil
```

With these out of the way, we can concentrate on the more minute details of behavior delegation.

First, the behavior process accessors will be private. We should initialize them to something, so we will provide the following `initialize` message.

```
PerceptionEnvironmentInterface>>initialize

    self perceptionProcess:
      (PerceptionPerceptionProcess withInterface: self).
    self actionProcess:
      (PerceptionActionProcess withInterface: self)
```

But we can clearly tell something is not right here. First, we have parentheses and some code pattern repetition. It would be nice if we did not have to use parentheses. And yet that is not the worst of our problems. What typically happens with `initialize` is that subclasses refine it according to their needs.

So what are we going to have to do with this message when we subclass the interface object? We will not be able to send `super` because of the explicit class references in this method, and if we reimplement it then we will have to copy & paste the implementation above, change the class names, and then add whatever else we may need. And of course, the resulting refinement of `initialize` will have the same deficiencies as the original one. This is utterly unacceptable.

Let's do something different instead. Since what is causing all of these issues is that there are explicit class references in a place where changing them is costly, then what we have to do suggests itself. We just have to move those references somewhere else... for example, to their own method! Thus,

```
PerceptionEnvironmentInterface>>initialize

    self perceptionProcess: self newPerceptionProcess.
    self actionProcess: self newActionProcess
```

Ah, this is much nicer now. Even the parentheses are gone. All there is left to do is to implement the messages that answer new processes. Note how the methods below make it much easier for subclasses to customize the default initialization behavior.

```
PerceptionEnvironmentInterface>>newPerceptionProcess

    ^PerceptionPerceptionProcess withInterface: self
```

```
PerceptionEnvironmentInterface>>newActionProcess

  ^PerceptionActionProcess withInterface: self
```

Now that we have our behavior processes set up, we can start delegating messages to them. First, we have to allow for the player to invoke their behavior in the case where the pattern is running in a single thread. That is easy to do.

```
PerceptionEnvironmentInterface>>perceive

  self perceptionProcess behave


PerceptionEnvironmentInterface>>execute

  self actionProcess behave
```

Piece of cake. Let's see... then we have the messages sent by the eyes, which are now implemented below.

```
PerceptionEnvironmentInterface>>perceptionCount

  ^self perceptionProcess perceptionCount


PerceptionEnvironmentInterface>>latestPerception

  ^self perceptionProcess latestPerception


PerceptionEnvironmentInterface>>couldNotPerceive

  ^self latestPerception isNil


PerceptionEnvironmentInterface>>ensurePerceptionAvailable

  [self couldNotPerceive] whileTrue:
    [self perceptionDelay wait]
```

```
PerceptionEnvironmentInterface>>perceptionDelay

  ^self perceptionProcess delay


PerceptionEnvironmentInterface>>actionDelay

  ^self actionProcess delay
```

This is almost too easy — good, that is precisely the point! Then there is the queuing message sent by the hands...

```
PerceptionEnvironmentInterface>>queueCommand: anObject

  self actionProcess queueCommand: anObject
```

And finally, since the interface object is acting as a composite made up of behavior processes, it would be nice if there was a bit of polymorphism available. Therefore, we add these three messages.

```
PerceptionEnvironmentInterface>>observeIndependently

  self connectToEnvironment.
  self perceptionProcess observeIndependently.
  self actionProcess observeIndependently


PerceptionEnvironmentInterface>>connectToEnvironment

  ^self


PerceptionEnvironmentInterface>>unwind

  self perceptionProcess unwind.
  self actionProcess unwind
```

There are a couple of observations to make here. First, note the existence of `connectToEnvironment`. We leave that around on purpose just in case some interace needs to do something special before spawning the embedded signals in

the perception and action process. Second, we can see our use of `unwind` paid off here. If we had been sending `signalShouldEvaluate: true` instead, we would find ourselves with no instance name to change in our object, and thus pushing the magic boolean to each of the behavior processes. By using `unwind` instead, we avoided writing a bunch of unnecessary code.

This completes the implementation of the action process and the environment interface. There is just one more entity to implement: the player. Now that we have all the parts ready, let's tie them together.

Player

As we said before, the player knows four objects by name. Since the player is not a behavior process itself, then we can subclass object and simply provide the required instance names as shown below.

```
Object
  PerceptionPlayer (interface eyes strategy hands)
```

The four getters will be public, while the four setters will be private. How about `initialize`? From our recent experience, we already know how we do not have to write it, and thus we implement it right the first time.

```
PerceptionPlayer>>initialize

  self eyes: self newEyes.
  self hands: self newHands.
  self interface: self newInterface
```

Since the strategy will be chosen by the eyes, we should leave it undefined at this time. The messages referenced above are quite straightforward.

```
PerceptionPlayer>>newEyes

  ^PerceptionPlayerEyes new


PerceptionPlayer>>newHands

  ^PerceptionPlayerHands new
```

```
PerceptionPlayer>>newInterface

    ^PerceptionEnvironmentInterface new
```

Finally, we come to the two most important messages: `play` and `unwind`. Let's tackle `unwind` first, since it is the easiest one.

```
PerceptionPlayer>>unwind

    self strategy notNil ifTrue: [self strategy unwind].
    self hands unwind.
    self eyes unwind.
    self interface unwind
```

Note the care taken to specify in which order the behavior processes should unwind — although if anything, it is just a matter of aesthetics since the state of each spawned behavior process signal is unknown here.

Then, we have the message `play`. At this point, we have to decide whether we will play with independent observers or in iterations. Thus, we write the following method.

```
PerceptionPlayer>>play

    self shouldPlayInIterations
      ifTrue: [self playInIterations]
      ifFalse: [self playWithIndependentObservers]
```

We will provide a default implementation for `shouldPlayInIterations`, and leave concrete subclasses to refine it as needed.

```
PerceptionPlayer>>shouldPlayInIterations

    ^true
```

Now we have to provide an implementation for the two branches referenced in `play`. Since we defaulted `shouldPlayInIterations` to `true`, let's examine that case first.

The process of playing in iterations has two stages. First, we need to set up the behavior processes by telling them things such as who are their senders and

receivers. After we do this once, we can then invoke their behavior in order, once per iteration. The iterations end when the eyes detect the game is over. Therefore, the method below seems quite reasonable.

```
PerceptionPlayer>>playInIterations

  self prepareToPlayInIterations.
  [self eyes shouldUnwindPlayer]
    whileFalse: [self playOneIteration]
```

The implementation of the message `prepareToPlayInIterations` now suggests itself.

```
PerceptionPlayer>>prepareToPlayInIterations

  self interface connectToEnvironment.
  self interface perceive.
  self eyes
    observeAccordingTo: self interface
    andDistinguishFor: self.
  self eyes setPlayerStrategy.
  self hands
    observeAccordingTo: self eyes
    andDistinguishFor: self interface.
  self strategy
    observeAccordingTo: self eyes
    andDistinguishFor: self hands
```

This implementation merits careful study. First the player tells the interface to connect to its environment. While the default implementation is to do nothing, this allows for things such as logins to occur. Then, the eyes are told that they will observe through the interface, and that they will be distinguishing objects for the player. But why is that done first? Because the very next thing that happens is that the player asks the eyes to set the player's strategy. Without the eyes telling the player which game is being played, there is no need to continue setting up the behavior processes. Once we have a strategy, we should not set it up just yet because it may need the hands to be set up correctly, and that has not happened yet. Therefore, we configure the hands before configuring the strategy.

One last detail: why the need to tell the interface to perceive? Because if it does not do it at that time, the eyes will not be able to tell if the player should unwind or not properly in `playInIterations`. If there is no perception, then `shouldUnwindPlayer` will not see any perception and conclude that one could not be obtained. Thus, we have to explicitly ask for a perception at least once.

This may make things sound like they are way, way too easy. It could even feel as if there must be some piece of messy code somewhere, right? Let's try a challenge question then. So, how is an iteration played? The answer is even simpler than the implementation of the previous message!

```
PerceptionPlayer>>playOneIteration

  self interface perceive.
  self eyes behave.
  self strategy behave.
  self hands behave.
  self interface execute
```

It is really hard to see how this could be any simpler. Let's try the multithreaded version instead, surely the mess must be hiding there.

```
PerceptionPlayer>>playWithIndependentObservers

  [
    self interface observeIndependently.
    self eyes
      observeAccordingTo: self interface
      andDistinguishFor: self.
    self eyes observeIndependently.
    self hands
      observeAccordingTo: self eyes
      andDistinguishFor: self interface.
    self hands observeIndependently.
    self strategy
      observeAccordingTo: self eyes
      andDistinguishFor: self hands.
    self strategy observeIndependently
  ] ifCurtailed: [self unwind]
```

And yet, the mess is nowhere to be found. And the implementation above is very simple, even though its invocation spawns five Smalltalk threads! Let's go over the fine details. First, notice that we do not ask the eyes to set up the player's strategy. So how can we be sure that the strategy will be there before we try to set it up? Because the eye's implementation of `observeIndependently` specifically sets up the player's strategy before answering! Since it is done in the same Smalltalk thread this method is being executed on, there can be no thread safety issues.

The rest of the inner block is straightforward... but why did we add an `ifCurtailed:` around the whole thing? Because, should it fail because of a bug, we might have live processes running around in our image. Thus, should there be a problem that causes the inner block to be unwound abnormally, we make sure to unwind any live processes that may have had a chance to get spawned. Doing so makes it easier for us as developers to recover from bugs that may still be in our to do list.

Believe it or not, this completes the implementation of the perception player object. That was not a whole lot of code, was it? Now that we have what we think is a working perception pattern implementation, it is time to go through the exercises so it can be seen in action.

6.3 Exercises

Exercise 6.1 [18] Find a better prefix for the classes of the pattern of perception so that one can avoid situations like the one below.

```
PerceptionBehaviorProcess
  PerceptionInterfaceProcess
    PerceptionPerceptionProcess
```

In particular, `PerceptionPerceptionProcess` is not very nice.

Exercise 6.2 [39] Use the perception pattern to solve Smalltalk Solutions 2006's Coding Contest. Bonus points if you can take care of the qualifier round within 63 hours (this includes time needed to sleep, eat and other things), and if you can then solve the final round within 2 hours.

Exercise 6.3 [37] Use the perception pattern to solve Smalltalk Solutions 2007's Coding Contest. Take no more than 9 calendar days to solve the qualifier round, and no more than 2 hours to solve the final round.

Exercise 6.4 [42] Use the player pattern to distinguish *interesting* differences in value in the output of `TimeProfiler profile: aBlock`, so that a player can give suggestions as to what to go after first.

Appendix A

Design distinctions behind Smalltalk

A.1 Regarding labels

I believe G. Spencer-Brown's ideas had at least some influence in the development of Smalltalk. Laws of Form was first published in 1969. Spencer-Brown was at Xerox PARC in 1979. In particular, David Robson implemented infinite-precision arithmetic on Alto computers and let Spencer-Brown run calculations at night. Later, special hardware was manufactured to run Laws of Form.

In his book, G. Spencer-Brown mentions working on forms and distinctions with his brother George. However, Alan Kay informs us that G. and George are one and the same.

While this may seem like circumstancial evidence of Spencer-Brown having some degree of influence in the development of Smalltalk, the original Design Principles Behind Smalltalk article contains the following quote.

> *The mind observes a vast universe of experience, both immediate and recorded. One can derive a sense of oneness with the universe simply by letting this experience be, just as it is. However, if one wishes to participate, literally to take a part, in the universe, one must draw distinctions. In so doing one identifies an object in the universe, and simultaneously all the rest becomes not-that-object. Distinction by itself is a start, but the process of distinguishing does not get any easier. Every time you want to talk about "that chair over there", you must repeat the entire processes of distinguishing that chair. This is where the act of reference comes in: we can associate a unique identifier with an object, and, from that time on, only the mention of that identifier is necessary to refer to the original object.*
>
> *—Dan Ingalls, Design Principles Behind Smalltalk, 1981*

Dan Ingalls confirms that this should be the case.

It sounds very familiar, doesn't it? Thus, G. Spencer-Brown should be credited for finding some of the ideas that shape Smalltalk in terms of distinctions and signals, as shown in Laws of Form.

> *The theme of this book is that a universe comes into being when a space is severed or taken apart. The skin of a living organism cuts off an outside from an inside. So does the circumference of a circle in a plane. By tracing the way we represent such a severance, we can begin to reconstruct, with an accuracy and coverage that appear almost uncanny, the basic forms underlying linguistic, mathematical, physical, and biological science, and can begin to see how the familiar laws of our own experience follow inexorably from the original act of severance [...] Distinction is perfect continence. There can be no distinction without motive, and there can be no motive unless contents are seen to differ in value. If a content is of value, a name can be taken to indicate this value. Thus the calling of the name can be identified with the value of the contents.*
>
> *—G. Spencer-Brown, Laws of Form, 1969*

A.2 An experiment

Well, but if it is truly about labels and distinctions, how does it actually happen for us? At this point, it may be good to consider the following interpretation.

There is a soup of labels. The labels are in a space which contains all the things we can experience. Each point of this space has a corresponding label, which can be thought of its position indicator in the space (note the dual behavior of labels).

Perceiving things in this label space corresponds to the act of distinguishing, out of the soup, a few individual labels and putting them inside a distinction. The idea behind doing this is that of clustering, as we saw earlier in the book. Then, according to your intentions, give a distinct label to each of the labels inside a distinction. Give a label to the distinction itself as well.

At this point, we just recognize things out of the soup of labels, and these things have no behavior associated to them. So now we need to model behavior.

Behavior, as we saw when discussing the optimization of `match:`, is a map between two label spaces, which in our case happen to be the same universal space. So a behavior is a label map living inside our soup of labels.

As we discussed, behavior may have a static context. But what if it refers to names that do not live in the static context? These would be the "arguments", so to speak. Let's write behavior assuming these names will be provided — note that this is what we do when we write methods and blocks. Then, invoking or

even mounting behavior on top of a distinction requires us to provide a name map between the names the behavior uses, and the names inside the distinction. In Smalltalk, this renaming occurs automatically when we send messages or evaluate blocks.

Note how Smalltalk classes would be more or less permanent mountings of particular behavior on top of distinctions which contain a fixed amount of labeled distinctions.

Now, behavior is a name map. The name map that connects behaviors and distinctions is evidently a name map. And our distinction is a label map as well. Should we write our programs in these terms instead?

After writing a prototype, I would have to say the answer to that question is no. In my experience, the code was much more verbose than Smalltalk. At the same time, Smalltalk code could be seen as a way to express the ideas in this section. The lesson, therefore, is that *it is not necessary to express the whole thought in the computer, just the result of the thought process is enough as long its expression is intention revealing.*

"Well, that is what [Smalltalk] is designed to do."
—Dan Ingalls

The reasons for this are fundamentally and critically important. As we know, we can only hold 7 ± 2 things in our minds. Consequently, as long as we carry most of our thoughts with things allocated to our local scratch ram, our speed of thought will be very fast. But if we now allocate some of the things we need to think into the computer, referencing such things will cause human interface traffic. In the same way that swap space thrashing brings any computer to its knees, excess of communication between us and the computer will considerably reduce our performance. Moreover, stressing our paging system will invariably lead to a higher probability of error.

Language design is an exercise of scratch ram register allocation. And as such, we can think of soups of labels, or even typed lambda calculus if we so desire — as long as the artifacts representing the results of such thought are extremely concise to minimize the impact of human interface chatter on our performance, yet intention revealing enough so that we can quickly reconstruct the thought processes that led to the particular expression of our solution.

Appendix B

A digression

Imagine everything we perceive was structured and represented as a sequence of bits which specify on which side of a distinction the particular value in question lies. Note how, in this arrangement, two perceived objects can be said to be equal in the strictest sense possible if and only if their bit representations are exactly equal. Clearly, if no bits are different, no distinction was drawn which shows the perceived objects to be different. This is because both objects fell on the same side of each boundary which was drawn in the space in which they were inscribed.

Integers, for example, naturally lend themselves to such arrangements. This is because, in their binary representation, the nth bit determines if their binary expansion as a sum of powers of two requires the presence of 2^{n-1} as a summand.

Moreover, assume that the distinctions drawn at perception time are chosen such that each additional bit increases the resolution with which the object is perceived. In other words, it would be like some game of twenty questions. This would allow to look at perceived objects up to an arbitrary level of resolution (or fuzziness), by simply masking out the unwanted bits. In turn, this makes it possible to see how similar objects are, and to identify in what regard is that they look like each other.

What operation is the one that tells us which are the bits that differ between two streams of bits? Why, of course... `bitXor:`. In addition, how could we find a gamut of boundaries that separate values perceived to be different? By using `bitXor:` and masking out an arbitrary number of bits in the answer.

In other words, we place our raw perceptions in hash buckets labeled by the very same perceived differences in value, then we map the occurrence of values in those hash buckets to hash buckets in our objective space by means of a function which in the context of the pattern of perception would be called strategy, and

the objectives decided upon hash action predictions which are then mapped again to hash buckets in our action space. And all of this is done up to an arbitrary level of resolution at each stage.

Finally, of course, if we add extra bits so that the perceived values can specify whether they represent a perception of something outside or inside the boundary of the player, we can let the player become aware of itself.

This just raises very important questions. Who drew the boundary of the player in the first place? What were the intentions behind that distinction being drawn? Regardless of how these questions are answered, note how the idea the player gets about its own existence is just an unintended consequence of how perception is performed. There is no need for the objects distinguished by the boundaries we draw when we look at differences in value to *actually* exist.

Appendix C

Causing brain change

John Sarkela has been a Smalltalk developer, consultant and instructor for many years. What follows is the organized content of many a discussion with him regarding Smalltalk. As you will see, this book is heavily influenced by him and Rebecca Wirfs-Brock, his mentor.

C.1 Teaching Smalltalk

C.1.1 Education as opposed to training

First there is the matter of what is education as opposed to training. Training implies rote and repetition. Education, however, is meant to be a process by which the frame of reference in which we represent our experience of the world is expanded.

Since opening our eyes to new ways of seeing things often causes a feeling similar to what we feel when we are hurt, education is to be carried out in a controlled fashion such that it is not too painful.

Most of the instruction work done by the instructor is usually about stretching the frame of reference of students according to the students' tolerance to the associated intellectual exertion, followed by a relaxation period which is used to reinforce what has been learned.

C.1.2 Efficient process for Smalltalk education

When showing Smalltalk to people who are not yet familiar with it, it is best to re-represent the actual problems at hand in a form suitable for the audience.

On several ocassions, John gave Smalltalk classes to COBOL programmers. Because the students did not have much of an object oriented background, he found that it was extremely effective to describe object oriented programming concepts by means of diagrams showing bubbles and arrows, in which bubbles stand for objects and arrows represent messages. Once you have bubbles and arrows, you can also label the arrows with a sequence of consecutive integers, and now you also have a description of a *conversation* between different objects.

Typically, for the first three days of a five day course, no Smalltalk code was written by the students. In fact, writing Smalltalk code was explicitly forbidden. However, after the bubble and arrow diagrams are well understood, it becomes easy to describe how to literally translate them into Smalltalk code.

For example, bubbles translate directly into classes. Also, if a bubble needs to keep the result of evaluating an arrow, then it means that a message send must be stored somewhere such as by means of an instance name or a temporary name.

These diagrams are very good tools that help organize which objects will take part in a conversation, which objects will be responsible for what, what is the nature of the relationship between the objects based on how they communicate to each other, how wide are the interfaces that connect them, and so on.

Note that their value as teaching tools should not be seen to imply that bubble and arrow diagrams are *not* useful to those already familiar with languages like Smalltalk. On the contrary, they are very valuable to understand object oriented programming issues in general.

C.1.3 Modalities of expression

There are three very important modalities of expression to keep in mind: textual, geometric, and neuromotor/proprioceptor. As you can imagine, each of them has their own advantages and disadvantages when used to teach Smalltalk.

Deep change needs to go through the neuromotor/proprioceptor level first. And example of this would be using body gestures to mimick the interaction between objects in terms of messages. This is important because it helps aligning the meaning of `self` in connection to concepts such as the sender or the receiver of a message. It also helps students to put themselves in the context of `self` and observe the conversation from the point of view of each object taking part in it.

Once this is internalized, it becomes much easier to skip the gestures through a geometric representation of the conversation, such as with the bubble and arrow diagrams. And once these are thoroughly learned, it becomes easy to translate all

this understanding into a textual representation such as Smalltalk source code.

Different people have different responses to each modality of representation, depending on how they evolved their distinction system. For the most part, however, the neuromotor/proprioceptor level is usually the best place to start.

Therefore, the basic organization of a module of instruction is such that it first makes heavy emphasis on a neuromotor expression of what is to be learnt, followed by a geometric representation of the same concepts, capped by learning how to express the acquired knowledge into effective source code.

C.1.4 Organization of Smalltalk courses

It should be an important goal of the training session design to bring the students into a context where it is safe to be wrong and vulnerable in front of others. If one cannot clearly observe and express what needs to be honed or perfected, then clearly it is extremely difficult for progress to happen.

A way in which the instructor can foster this type of environment is to pay attention to every single question, down to the most ill conceived ones, and rephrase them in terms the students cannot *yet* articulate. In other words, the work should be to help others to structure thought regarding still unfamiliar objects.

Assisting the class in this way also prevents the typical interaction between students and the instructor, in which students ask questions that are invariably rewarded with a correct, authoritative answer. This is to be avoided, because otherwise it becomes impossible to put more emphasis into focusing on *how to get to the right answer*, and *where could we look for the answer to the next problem*[1].

It should be a goal of education to train students in the process of educating themselves. When this does not happen, the instructor may find him or herself in a situation in which a question is asked to the students, and the reaction obtained is intense staring waiting for the punch line. Please pause for a moment and consider how useful it is to stare at someone that asks you a question if the idea is to produce a coherent answer.

These are the core modalities of teaching that can be used to design the overall arch for the course.

[1]For more details on the questions that may be useful to solve problems, see How to Solve It by G. Pólyà.

C.2 Programming in practice

C.2.1 Responsibilities

As discussed before, it is important to not let people write code until they can show bubble and arrow diagrams of what they are doing. Once they can do this, they can be asked how they would test the functionality of the program implied by the diagram. Note that this fosters thinking about writing tests before writing code, which naturally leads to well known productive software practices.

Responsibility driven design, as exemplified before by the bubble and arrow diagrams, makes the XP practice of writing tests before writing code a rather obvious consequence to developers who are taught about responsibility before learning how to write code. It should also be noted that it is utterly useless and counterproductive to force dogmatic postures on authoritative grounds, when the value of a practice can be derived so easily from a simple diagram that helps assign different work to different objects while coordinating their activities. Thus, our work as developers is about extremely *responsible* programming.

But what do we mean by responsible? Let's take a look at the following two definitions behind the distinction *responsibility*.

1. Conformance of action to shared values. In other words, *do the right thing*, according to what the context dictates the right thing is.

2. The ability of a system to respond. This refers to the capability of some object to behave well.

Clearly, we want to build programs that satisfy both. Therefore, when we consider collaboration in the context of it being the evolution of a signal in an information space, the goal should be to design the parts such that they conform to the definitions of responsibility given above.

This does not necessarily apply to Smalltalk only, but just in object oriented teaching in general. Moreover, what we can say about the requirements for a program can be said for the requirements for the development team.

C.2.2 Designing relationships

Sometimes, strong proponents of XP practices will argue that XP means no upfront design should be made because the goal is to make the simplest thing that could possibly work.

Claims such as these are thoroughly bogus. The original admonition against *excessive* time and energy spent in design was meant as a warning against analysis paralysis. Doing design upfront does not imply that no prototypes will be made as work progresses. Besides, writing tests first certainly is an exercise in design, because the tests carry with them implicit choices regarding the design of objects' external interface.

To do no upfront design implies unresponsible programming, because the responsibilities of the objects involved cannot be clearly defined when no thought is spent on this matter. It would have been much better to use the term *scoped up front design*.

This observation is best illustrated by means of an example. In Smalltalk training courses, sometimes students were asked to design a class hierarchy for kitchen utensils of all sorts, including silverware, flatware, cookware, cups and glasses of many kinds.

Note that there was no indication of what the intention behind the required distinctions was. As such, students would come up with one of their own and produce a diversity of elaborate class hierarchies. No two designs would have the same structure or guiding principles, and in some cases even multiple inheritance would be called into action.

After allowing for a healthy dose of discussion to determine which design was better, the original purpose was revealed: the customer was a recycling company and stuff had to be sorted according to whether the material to recycle was wood, metal, glass, or otherwise. All of a sudden, using multiple inheritance to model the idea of e.g.: *cupness* ceases to be relevant.

This shows that no upfront design is just as bad as work done in the absence of enough information to write meaningful tests.

C.2.3 Delegate and contract

When assigning responsibilities between objects, what typically happens is that whenever an object delegates a task to another object, the delegation is done such that the possible outcomes for the delegated task are implicitly constrained by the context from which the delegation occurs.

Smalltalk's mixed arithmetic, implemented in terms of double dispatching, is an example of the *delegate and contract* pattern that we can see explicitly in source code. Further evolution of the signal traversing the message send graph progressively contracts the space of potential answers until only one is possible, thus becoming obvious.

This occurs in other areas as well. In design, this manifests as different levels of objects in which responsibility is expressed, from the topmost architecture overview down to contexts in which there are actual objects.

The design itself is an attractor of the behavior process implied by our design activities. If we know our design behavior is a contractive process, in as much as each step further reduces the possible outcomes, then we know further design work will result in a better approximation to the attractor which represents an ideal design. Therefore, even if we do not where we are going, we will eventually get as close as necessary because only one end point is possible.

C.3 Success and failure

C.3.1 Shared context

Before we engage in any activity, there must be an intention according to which we want to act. The problem is that, for the vast majority of cases, the ideal intention is hardly ever known to the last detail, much less explicitly represented with the same degree of precision, before the project is done.

In order to go about fulfilling our objectives, first one has to know the lay of the field in which one stands. Distinguishing different values will cause each of us to move in different directions according to our own behavior traits. Once this is known, we should always keep an eye for opportunities to clarify or rectify our intentions as more information becomes available.

This context in which we are supposed to develop a piece of software, however, is particularly more complex when the behavior we are considering is that of a team composed of different individuals.

Even when each of the team members has a clear intention, if the team members have no shared values, then there is no way in which a team project can succeed because different developers will pull the behavior of the team in different directions. Therefore, a development team has to have a strong set of shared values in the context of which everything else is to be interpreted and looked at.

C.3.2 Consequences

The feasibility assessment of a project is strongly and invariably related to the particular characteristics of the development team's behavior.

The first order approximation to the answer to whether a project is likely to succeed is essentially tantamount to the existence of a context in which the development team has a set of collective shared values. For example, does the team know what the deliverable is? If the team members cannot agree on what is they need to build to succeed, then failure is obviously certain.

Only after the development team has a common point of view on what their work product should be, does the matter of technical feasibilily become an issue worth considering.

Further measurements, such as the ability of individual developers, the ability of members to work together as a team and so on, are pretty much insignificant compared to the first two.

If you have aligned intent and technical feasibility, pretty much any group of people following a contractive process should arrive at a fixed point such as a working deliverable.

Without a collaborative atmosphere, each developer will pull the environment in different directions. Because there is no need for these to be coherent, failure becomes extremely likely.

C.3.3 Complexity

This illustrates a very strong principle of successful complexity management. Instead of dealing with a combinatorial explosion of possibilities which force us to deal with anything that could have potentially gone right or wrong at each particular point of the traversal, the idea is to set up a combinatorial implosion so that each step of the process eliminates a sizable fraction of possible outcomes. In short, we should use complexity to drive the traversal to our fixed point.

In this context, the role of a manager is to coordinate the behavior attractors of each team member and each team activity, so that their overall behavior converges to an attractor implied by their arrangement which is compatible with the production of the desired deliverable.

C.3.4 Conflict resolution

Engineers and marketing people usually have different points of view on what should be done. This is simply because they value different objectives, in turn this is because they distinguish differently, and this is just because have different intentions. However, the context in which both engineers and marketing members of a team share values consistent with their goal should be the dominant one.

This immediately and *naturally* determines how conflict should be resolved. Simply put, actions incompatible with each other need to be evaluated in terms of the shared value context, and the one that best represents the team's shared values should be the one chosen.

Appendix D

Answers to exercises

D.1 It's all in a name

Answer to exercise 1.1. You would be able to think of no distinctions, since there can be no distinction without intention.

Answer to exercise 1.2. The law of calling would be broken because using the name again would not have the same value as using the name before the assignment is performed. If the change of value propagated to the sender as well, the law of crossing would broken as well.

Answer to exercise 1.3. Messages would have to provide answers to an object other than the sender.

Answer to exercise 1.4. There are no useful string references to `':='` handy, therefore it takes some investigation. Fortunately there is a class called `Parser`. That is a very suspicious name to have. And indeed, from `Parser>>expression`, we can see the parser scans over `:=` manually by first recognizing `$:`, and then `$=`. Therefore, adding an `or:` check for `$>` allows it to recognize `:>` as the assignment token.

Answer to exercise 1.5. In this scenario, the message is sent to its own sender. The argument of the message, in this particular case `self` itself, is already known in the context of the sender. Hence, when the message is sent, the argument crosses the boundary of the sender just to cross it again a moment later. By the law of crossing, this is equivalent to doing nothing.

If the argument of the message is always `self`, the implementation can be greatly improved by sending a unary message instead.

Answer to exercise 1.6. Implement `decimalPrintString` in both `Fraction` and `Integer`, then refine the implementation in `Number`.

```
Number>>decimalPrintString

  ^self printString


Fraction>>decimalPrintString

  ^self asFloat printString


Integer>>decimalPrintString

  ^self printString, '.0'
```

Answer to exercise 1.7. The issue is that 0 and 1 do not know their identity, and as such you will have to do more comparisons to figure out what to do when an integer receives the message `ifTrue:ifFalse:`. The hierarchy starting at `Boolean` addresses this issue by having `true` and `false` know who they are. Therefore, the implementation of the messages they respond to can use the fact that the code will be executed in the context of a particular receiver. This is something that cannot be done with integers.

Answer to exercise 1.8. I would be really interested to see a solution. Please let me know what you find!

Answer to exercise 1.9. The expression *method name* is not enough because there can be temporary names inside blocks. Thus, *closure name* is probably better.

Answer to exercise 1.10. Referencing local names directly instead of using accessors increases the complexity of the whole method. On the other hand, using accessor messages limits the scope in which the complexity increases to just the sentence in which they are sent. Therefore, always use accessors.

Answer to exercise 1.11. The expression is a direct translation of Laws of Form's first axiom.

Answer to exercise 1.12. Including a return in a sort block will break the implementation of `SortedCollection` because the sort block will return from the method controlling the sorting.

Answer to exercise 1.13. Since `SmallInteger class` does not implement the message `class`, `super class` is the same as `self class`. Evidently, `self class` is a metaclass, while `self superclass` is a regular class. Since they are different, the answer is `false`.

Answer to exercise 1.14. Since `SmallInteger` does not implement `class`, the `self` and `super` sends are equivalent. Hence, the answer is `true`.

Answer to exercise 1.15. Although the superclass of `Object` is undefined (in other words, `nil`), the superclass of `Object class` is `Class` — not a metaclass! Therefore, the expression will evaluate to `true`. Even classes are objects.

Answer to exercise 1.16. The answer is yes, because every negated fraction is a fraction.

Answer to exercise 1.17. Yes, because the reciprocal of a fraction cannot be an integer.

Answer to exercise 1.18. No, for example the floating point number below

```
(2.0 raisedTo: -127) / 128.0
```

has no reciprocal.

Answer to exercise 1.19. For the most part, the answer to the question is yes. The only exception is `SmallInteger minVal`.

Answer to exercise 1.20. As in the previous exercise, the assertion holds with one exception. In this case, the edge case is `SmallInteger maxVal + 1`.

Answer to exercise 1.21. No, the result can overflow into the large integers. For instance, `SmallInteger maxVal + SmallInteger maxVal`.

Answer to exercise 1.22. No, the result can be a small integer. For example, `(SmallInteger maxVal + 5) + (SmallInteger minVal - 4) = 0`.

Answer to exercise 1.23. In the same way that `should:raise:` makes sense in an environment where exceptions are *exceptional events*, its negative form makes sense in an environment where exceptions are *common*. As such, it would seem that `shouldnt:raise:` is intention obscuring.

Answer to exercise 1.24. Implement the message `AbstractProxy>>class` in a manner equivalent to the sample below. Note that this approach also leaves an explicit reference to where the behavior is implemented.

```
AbstractProxy>>class

  self class methodDictionary
    at: #class
    put: (Object methodDictionary at: #class).
  ^self class
```

There may be no source code and no decompilation of the method in question, but the compiled method is still accessible no matter how magic it might be.

Answer to exercise 1.25. The first expression can take advantage of double precision floating point denormalized mantissas, while the second one cannot since the required exponent for the operation does not fit in the bits allocated for it.

Answer to exercise 1.26. Obviously, something must be wrong because base two floating point numbers are of the form $a2^k$, with a being an integer. If the equality were to hold, then we would have

$$a2^k = \frac{2}{5}$$

and therefore,

$$5a = 2^{1-k}$$

which is impossible by the Fundamental Theorem of Arithmetic (which states that integers can be factored into prime factors in exactly one way, with the exception of a permutation of the factors).

Note that this derivation holds even with denormalized floating point numbers because the notation used is able to represent the exact value of all floating point numbers that use base 2.

Further inspection clearly shows that the message `asRational` does not do what its name implies.

Answer to exercise 1.27. Use the fact that `become:` is commutative, and that the primitive can be invoked directly by sending `primBecome:`.

```
String new primBecome: t
```

Answer to exercise 1.28. The issue is that as soon as the binary search enters the region where the objects sort as being equal, we can find two associations `x`, `y` such that the following conditions hold at the same time.

```
self binarySearchIs: x lessThan: y
self binarySearchIs: y lessThan: x
(x = y) not
```

Since the excluded middle principle breaks down as shown above, the binary search implementation shown does not work properly. At first, it seems that we should check against these cases explicitly and fall back to linear search when needed. However, our observation about the excluded middle also points at what needs to be done. In other words, if the excluded middle cannot be expected to work, then do not use it to avoid what might be seen as unnecessary work.

Premature optimization strikes again.

```
SequenceableCollection>>binaryIndexOf: anObject

  ^self
    binaryIndexOf: anObject
    ifAbsent: [0]


SequenceableCollection>>
  binaryIndexOf: anObject
  ifAbsent: aBlock

    | interimAnswer |
    interimAnswer := self
      binaryIndexOf: anObject
      from: 1
      to: self size.
    interimAnswer = 0 ifTrue: [^aBlock value].
    ^interimAnswer
```

```
SequenceableCollection>>
  binaryIndexOf: anObject
  from: start
  to: stop

    | pivotIndex pivotObject |
    start > stop ifTrue: [^0].
    pivotIndex := start + stop // 2.
    pivotObject := self at: pivotIndex.
    pivotObject = anObject ifTrue: [^pivotIndex].
    (self binarySearchIs: anObject lessThan: pivotObject)
      ifTrue:
        [
          | interimAnswer |
          interimAnswer := self
            binaryIndexOf: anObject
            from: start
            to: pivotIndex - 1.
          interimAnswer > 0 ifTrue: [^interimAnswer]
        ].
    (self binarySearchIs: pivotObject lessThan: anObject)
      ifFalse:
        [
          | interimAnswer |
          interimAnswer := self
            binaryIndexOf: anObject
            from: pivotIndex + 1
            to: end.
          interimAnswer > 0 ifTrue: [^interimanswer]
        ].
    ^0
```

D.2 Complex conditions

Answer to exercise 2.1. Three levels of block nesting and a pair of parentheses become unnecessary in this version.

```
EligibilityValidator>>validateSomethingHasChanged

  [model determination = model currentEligibility],
  [model effectiveDate = model currentBeginDate],
  [model determinationDate = model currentReviewDate],
  [model reason = model currentOpenReason]
    ifAllTrue: [self complain]
```

Answer to exercise 2.2. The message `xor:` will answer whether the receiver and the argument are different. Therefore, nested `xor:` sends are equivalent to chained `~=` sends.

```
self conditionA
  ~= self conditionB
  ~= self conditionC
  ~= self conditionD
    ifTrue: [self parityIsOdd]
    ifFalse: [self parityIsEven]
```

Answer to exercise 2.3. When the action taken by the first two sentences is the same, complex conditions are not even necessary.

```
^aHasDot = bHasDot
  ifTrue: [a < b]
  ifFalse: [bHasDot]
```

Answer to exercise 2.4. The logical implication $p \Rightarrow q$ is equivalent to the statement $\neg p \vee q$. Therefore,

```
True>>implies: aBlock

  ^aBlock value


False>>implies: aBlock

  ^true
```

Answer to exercise 2.5. A possible selector is shown below.

```
implies:ifTrue:ifFalse:
```

Answer to exercise 2.6. This type of extension produces shorter code and messages with less keywords. You could use the selector `=>`, for example.

Answer to exercise 2.7. Please let me know what you find.

Answer to exercise 2.8. Combine `allTrue` and `whileTrue:` as shown below.

```
[self conditionA],
[self conditionB],
[self conditionC]
  whileAllTrue: [self action]
```

This can be extended further by combining `whileTrue:` and `whileFalse:` with other members of the unary selector family. Interestingly, combining selectors in this fashion also yields `whileAllDefined:`, `whileAllUndefined:`, etc.

Answer to exercise 2.9. We can let the answer be defined by the context of the receiver as follows.

```
True>>xor: aBoolean

  ^aBoolean not


False>>xor: aBoolean

  ^aBoolean
```

Incidentally, this allows replacing the conditional logic in `Boolean>>xor:` with `self subclassResponsibility`.

Answer to exercise 2.10. Note how the new argument makes every context it visits harder to understand.

Answer to exercise 2.11. While writing the book, I found the much more beautiful implementation shown below.

```
ComplexConditionTestCase>>defined

  ^[self]
```

Answer to exercise 2.12. Let me know what you find.

Answer to exercise 2.13. Since `notNil` is not a global name, a third message becomes necessary to support three argument messages. A good selector choice would be `noneEvaluatesTo:`.

Answer to exercise 2.14. You could rewrite it along the lines of the sample solution below.

```
addChannelUsing: anInteger
bitsPerSymbolWithCodingStrategy: aCodingStrategy
andDeltaStrategy: aDeltaStrategy
```

Answer to exercise 2.15. Make a new hierarchy of classes similar to that of `Boolean`. Create two sibling classes called `Enabled` and `TurnedOn`. Create a unique instance of each and make them singletons. Reference them in the global namespace with names such as `enabled` and `on`. Implement `printOn:` methods accordingly. Let the singletons understand messages such as `ifTurnedOn:` and `ifEnabled:`. Do something similar for `Disabled` and `TurnedOff`. Then, you will be able to write code as shown below.

```
self letFeatureBe: enabled.
self turnFeature: on.
anObject feature ifTurnedOn: aBlock.
```

Answer to exercise 2.16. Howard Oh found this exercise while using Dolphin Smalltalk. He solved it by implementing the following message.

```
Object>>--> aBlock

  ^aBlock value: self
```

Once the keyword messages and parentheses disappear, the original expression boils down to the one shown below.

```
2 --> [:x | x * x] --> [:x | x + 5] --> [:x | x sqrt]
```

Interestingly, this also allows 2 to travel from left to right in the statement, as opposed to what happened in the original expression.

Answer to exercise 2.17. The fact that the first two statements were similar was not used. Consequently, it is possible to simplify further as follows.

```
aHasDot ~= bHasDot ifTrue: [^bHasDot].
^aHasDot ifTrue: [a < b] ifFalse: [a > b]
```

However, this kind of reduction may not be apparent at first sight, and the complex condition equivalent expression is more clear.

Answer to exercise 2.18. Use the keyword selector combination technique.

```
aCollection atIndexOf: oldObject1 put: newObject1.
aCollection atIndexOf: oldObject2 put: newObject2.
aCollection atIndexOf: oldObject3 put: newObject3.
aCollection atIndexOf: oldObject4 put: newObject4
```

Some Smalltalks may also implement `replace:with:`.

Answer to exercise 2.19. If the sender does not care whether it actually gets a new instance or an already existing instance from a class, it could just send the message `some`.

Comments are often used as deodorant.

Answer to exercise 2.21. The notes for each of the code snippets are below. You may be inclined to think the examples are fabricated — they are not!

`BankAccount>>setBalance`. The comment just restates what the code is doing, taking considerably more characters to do this than the size of the code being commented itself. Such comments are useless.

`SomeClass class>>new`. The message `new` is the de facto standard for instance creation purposes, so commenting on the fact is redundant.

`SomeCollection>>size`. The name of the message is enough intention revealing already, and yet both the selector and the implementation are verbosely commented regardless.

`"Check to see if the string ends with A or E"`. Although the selectors used have been carefully crafted, it appears that such clarity is not enough to prevent a restatement as a comment. Such explanations are completely unnecessary.

`WebClient class>>server: aServer port: aPort`. If descriptive argument names were used, such as `serverAddressString` and `portNumber`, then the comments would become superfluous.

`Integer>>factorial`. The comment talks about what the method does, not what the answer is.

`Magnitude>>between:and:`. The message selector is right above the comment, why is it necessary to repeat it?

As a side note, Roger Whitney, computer programming instructor at San Diego State University, notes that universities have a tendency to train students into writing comments for even the simplest methods because instructors do not read the students' work due to lack of sufficient time to properly evaluate homework assignments. Therefore, comments are perceived as a feature by the instructors who grade work by skimming, reinforcing the notion that even `x := x + 1` needs a comment.

"Comments are easier to write poorly than well, and commenting can be more damaging than helpful [...] The main contributor to code level documentation is not comments, but good programming style."
—Steve McConnell

D.3 On `CharacterArray>>match:`

Answer to exercise 3.1. Each step of the pattern reduction can be shown to be a necessary condition for a match to occur. In addition, the reduction cannot go on indefinitely, and hence it finishes.

Answer to exercise 3.2. Note how it becomes necessary to check whether the pattern ends with a star token.

Answer to exercise 3.4. VisualAge 6's version of `match:` is a perfect example of numerous things that should be avoided at all costs. Hopefully, you will be able to contribute a better implementation for the next release of VisualAge if the original code has not been taken care of already.

By the way, *octothorpe* is a name for the character `$#`.

Answer to exercise 3.5. Since `match:` does not support this feature, write the tests in `StringMatchesTest`. Make `StringInlinedMatchesTests` a subclass of `StringMatchesTests`. Then, create a subclass of `PoundToken` and change the parser accordingly. Note how the increased context size makes inlining more painful.

Unfortunately, `$#` is already taken. A solution would be to change `matches:` to use `$?` instead, thus making `$#` available for digits. However, this may cause more confusion than it eliminates.

Answer to exercise 3.6. Interestingly enough, tokens themselves do not need to be changed.

Answer to exercise 3.7. One solution is to combine the message `match:` with `ifTrue:ifFalse:`. Another solution would be to find a good binary selector to represent `match:`, but unfortunately there are not many ASCII glyphs to go around.

Answer to exercise 3.9. Make two subclasses, one for case sensitive tests and the other for case insensitive tests. Then, refine `pattern:matches:` as shown below.

```
CaseInsensitiveMatchTests>>pattern: aPattern matches: aString

  (self
    basicPattern: aPattern asLowercase
    matches: aString asLowercase)
      ifFalse: [^false].
  (self
    basicPattern: aPattern asLowercase
    matches: aString asUppercase)
      ifFalse: [^false].
  (self
    basicPattern: aPattern asUppercase
    matches: aString asLowercase)
      ifFalse: [^false].
  (self
    basicPattern: aPattern asUppercase
    matches: aString asUppercase)
      ifFalse: [^false].
  ^true
```

Delegate the actual match message send to `basicPattern:matches:`. Rearrange the class hierarchy as appropriate, for example:

```
StringInlinedMatchesTest>>
  basicPattern: aPattern
  matches: aString

    ^aPattern inlinedMatches: aString
```

In this way, no changes to the actual test messages are needed.

Answer to exercise 3.10. One possibility would be to use the prefixes `Leading` and `Trailing`.

Answer to exercise 3.12. Show that the relation is reflexive, symmetric and transitive. Equivalence, as you can see, is viral as well.

Answer to exercise 3.13. Show that the relation is areflexive (no software can be a derivative work of itself), antisymmetric (no piece of software can be a derivative work of a derivative work of its own), and transitive.

Answer to exercise 3.14. The implementation is trying to decide whether to create `aWord`. Hence, it is a modification of sorts to the instance creation behavior of classes. Thus, such modification belongs to the class. The original piece can be replaced with the following expression.

```
^Word nonEmptyWordOn: wordStream contents
```

Note that because of the context in which the decision to create a word is made, temporary names are no longer needed.

```
Word class>>nonEmptyWordOn: aString

  aString isEmpty ifTrue: [^nil].
  ^self on: aString
```

Answer to exercise 3.15. There are a variety of solutions that involve spawning multiple processes. For example, spawn the `catchTheNextTrain` process in the context of an observer process, e.g.: `waitForValueUntilPatienceRunsOut`. Then your internal process can wait until either the train comes or you run out of patience.

Answer to exercise 3.17. Create a stream object on the pattern and the string to keep track of the matching interval. In this way, 6 arguments would be replaced with 2. The code becomes much more clear to read, and it runs faster than the original tokenizer implementation by a factor of at least 2. However, this also means the stream implementation run between 3 and 4 times slower than **Dc** and **De**. Interestingly, the resources spent in stream object creation do not contribute significantly to this difference. Rather, it is mostly because stream contexts are carved in such a way that their boundaries cause critical position information to cease being immediately accessible to the traversal decision process.

Answer to exercise 3.18. Delegate finding the range indices to the characters themselves.

```
nextWord

  ^input nextWord


String>>nextWord

  | endIndex |
  endIndex := self nextWordEndIndexStartingAt: 1.
  endIndex < 1 ifTrue: [^nil].
  ^Word on: (self copyFrom: 1 to: endIndex)


String>>nextWordEndIndexStartingAt: anInteger

  anInteger > self size ifTrue: [^anInteger].
  ^(self at: anInteger)
      nextWordEndIndexIn: self
      startingAt: anInteger


AlphanumericCharacter>>
  nextWordEndIndexIn: aString
  startingAt: anInteger

    ^aString nextWordEndIndexStartingAt: anInteger + 1
```

```
AnyOtherCharacter>>
  nextWordEndIndexIn: aString
  startingAt: anInteger

    ^anInteger - 1
```

The statement count (including those inside blocks) went down from 11 to 9.

Answer to exercise 3.20. Clearly, every traversal will alternate between each perpendicular direction at each step. This means that each up-down context has the knowledge to create a left-right context that can help further, and so on. Therefore, create a hierarchy of lines as follows:

```
DiscreteLine
  HorizontalDiscreteLine
    LeftDiscreteSemiline
    RightDiscreteSemiline
  VerticalDiscreteLine
    DownDiscreteSemiline
    UpDiscreteSemiline
```

Then, let instances of `DiscreteTraversalFinder` accumulate stops dictated by the semilines. In other words, if you start from `0@0` going left, then the starting semiline is `(0@0) leftLine`. This line can then be asked how much it should go left from `0@0` to reach `(4@7) downLine`. Once it gives its answer, it can also create an `upLine` or `downLine` because `(4@7) downLine` knows the correct approach direction. This new semiline can be asked how much to travel towards `(4@7) downLine`, etcetera.

Use double dispatch and polymorphism to their last consequences so you can avoid excessive occurrences of `ifTrue:ifFalse:`.

Answer to exercise 3.21. The context does not provide immediate access to the pieces that determine the answer. A solution would be to delegate part of the traversal to a more specific context. For example,

```
Matrix>>isNull

  ^self rows allSatisfy: [:each | each isNull]
```

```
MatrixRow>>isNull

  ^self allSatisfy: [:each | each isNull]
```

Besides drastically reducing the code size, this implementation even suggests implementing the message `allRowsSatisfy:`.

Answer to exercise 3.22. Unfortunately, enumerated class based characters are not literals, so you cannot write a method to answer one of them without sending a message — unless it is stored in an instance name.

Answer to exercise 3.23. In VisualWorks, characters are immediate objects just like small integers. This means that, from the virtual machine's point of view, standard characters are much easier to work with than our class based characters. This may have an effect in our benchmarks.

Since the value of the character is stored in the pointer, in theory one could refine the virtual machine to reify instances of class based characters, as opposed to instances of a single character class. The feasibility and usefulness of this, however, is left unspecified.

Answer to exercise 3.24. Reuse the solution for case insensitive test cases and implement the single message shown below.

```
basicPattern: aPattern matches: aString

  ^aPattern asCondensedCharacterArray
    matches: aString asCondensedCharacterArray
```

Answer to exercise 3.25. The following messages implemented in `Character` would benefit from being rewritten with enumerated class based character classes.

1. `asInteger`. Would return the proper integer value.
2. `asLowercase`. Would return an instance name instead of performing more expensive lookups.
3. `asUppercase`. Same as above.
4. `basePart`. Would return a character of `self`, as appropriate.

5. `composeDiacritical:`. Could be simplified considerably by using double dispatching.
6. `diacriticalPart`. Same as `basePart`.
7. `digitValue`. Would answer a literal integer as appropriate.
8. `isAlphabetic`. Would answer a boolean as appropriate.
9. `isAlphaNumeric`. Same as above.
10. `isComposed`. Same as above.
11. `isDiacritical`. Same as above.
12. `isDigit`. Same as above.
13. `isLetter`. Same as above. The existing implementation is quite lengthy, yet it would reduce to nothing.
14. `isLowercase`. Same as above.
15. `isSeparator`. Same as above.
16. `isUppercase`. Same as above.
17. `isVowel`. Same as above.
18. `printOn:`. Receivers would know their constant names.
19. `storeOn:`. It would not be necessary to output the representation of the receiver in terms of an integer, as it would be implied by the class name.

In addition, I have found `Character>>asString` quite handy in some situations. If enumerated characters were made to be literals, it would be possible to return a literal string instead of creating a new one every time.

Answer to exercise 3.26. The majority of comparisons performed in messages implemented in class based characters can be eliminated as the situations they try to detect are eventually covered by comparisons in messages implemented in class based character arrays.

Answer to exercise 3.27. When the redundant checks are removed, the actual implementation of the messages matching `*afterLiteralLet:*` becomes equal to that of the messages matching `*let:*continueForwardMatches:*`. As such, their implementations can be merged into a single message.

Answer to exercise 3.29. Expressions such as

```
someCharacter == $*
  ifTrue: [...]
  ifFalse: [...]
```

cannot be employed by **VW+e**. This is because class based characters are not recognized as literal objects by the compiler. Without changing the compiler, it is possible to resort to polymorphism once more and proceed as shown below.

```
someCharacter
  ifStar: [...]
  otherwise: [...]
```

However, in this case the expression below is faster because the compiler inlines `ifTrue:ifFalse:` and thus blocks do not have to be sent as arguments.

```
someCharacter isStar
  ifTrue: [...]
  ifFalse: [...]
```

You may want to take a look at the `RBBytecodeTool`.

Answer to exercise 3.31. The issue is that one has to do, essentially, a class check to see if addition and recursion should be performed. But that knowledge can be given to the receiver. Therefore,

```
Behavior>>depth

  ^self depthCountingFrom: 0


Behavior>>depthCountingFrom: anInteger

  ^self superclass depthCountingFrom: anInteger + 1
```

```
UndefinedObject>>depthCountingFrom: anInteger

  ^anInteger
```

Answer to exercise 3.32. Polymorphism refers to multiple implementations of a message making it possible to send a single message to objects belonging to many classes, particularly in the case where the multiple implementations are not necessarily in the same class hierarchy. An example of this would be the implementation of `class` in subclasses of `nil`. As such, it follows that there can be no polymorphism when there is only one implementor of the message in question.

Answer to exercise 3.33. Any message the virtual machine relies on must be understood by every object. Because such messages must be understood by all subclasses of `nil`, it turns out that such messages are, by definition, polymorphic. One such message is `doesNotUnderstand:`.

Answer to exercise 3.34. Implement two different methods for the message `includesBehavior:`, one in `Behavior` and another one in `UndefinedObject`, and thus take advantage of polymorphism.

```
Behavior>>includesBehavior: aClass

  self == aClass ifTrue: [^true].
  superclass == nil ifTrue: [^false].
  ^superclass includesBehavior: aClass


UndefinedObject>>includesBehavior: aClass

  ^false
```

D.4 Validation revisited

Answer to exercise 4.1. A truly interesting and beautiful implementation is possible by letting the pretty print signal travel across the class based character array being pretty printed. Otherwise,

```
CharacterArray>>prettyPrint

  | input output useCaps next addSpace |
  addSpace := false.
  useCaps := true.
  input := self readStream.
  output := WriteStream on: self species new.
  [input atEnd] whileFalse:
    [
      next := input next.
      addSpace := next isUppercase.
      useCaps
        ifTrue: [next := next asUppercase.  useCaps := false]
        ifFalse: [next := next asLowercase].
      (addSpace and: [output position > 0])
        ifTrue: [output nextPut: Character space].
      output nextPut: next
    ].
  ^output contents
```

Answer to exercise 4.2. Let individual exceptions understand a message such as `recordOccurrenceInTestResult:`, and use polymorphism so that you can implement `runCase:` as shown below.

```
TestResult>>runCase: aTestCase

  | testCasePassed |
  testCasePassed :=
    [
      aTestCase runCase.
      true
    ]
      on: GenericException
      do: [:ex | ex recordOccurrenceInTestResult: self].
  testCasePassed ifTrue: [self passed add: aTestCase]
```

Polymorphism eliminates the need for nesting.

Answer to exercise 4.3. Inline the last expression inside the guarded block. Since the value of the temporary name is known at that point, `ifTrue:ifFalse:` becomes unnecessary.

```
TestResult>>runCase: aTestCase

  [
    aTestCase runCase.
    self passed add: aTestCase
  ]
    on: GenericException
    do: [:ex | ex recordOccurrenceWhileRunning: aTestCase]
```

Answer to exercise 4.4. Have `runCase:` raise a test case passed notification. Subsequent refactoring allows removing the code that handles passes.

```
TestResult>>runCase: aTestCase

  [aTestCase runCase]
    on: GenericException
    do:
      [:ex |
        ex
          recordOccurrenceWhileRunning: aTestCase
          inTestResult: self
      ]
```

Answer to exercise 4.5. The issue is that `GenericException` is the superclass of `UnhandledException` (which should be handled) as well as of the exceptions used by the debugger (which should be left alone). Implement a message called `exceptionsToTrap` that creates a composite exception handler which excludes the hierarchy starting at `ControlInterrupt`, and use that instead in `runCase:`.

Answer to exercise 4.6. As an example, whether instances of `TestSuite` or `ValidationSuite` are created should depend solely on the implementation of a single message, e.g.: `suiteClass`.

Answer to exercise 4.7. The exception caused by `halt` goes unhandled, so an exception of the class `UnhandledException` is raised. This last exception,

however, is not an `Error`. Depending on how the exception handlers are set up, either `UnhandledException` will be caught in which case the test will be recorded as ending in error, or it will not be caught in which case you will see a debugger pop up.

Answer to exercise 4.8. One could implement the following messages and use them accordingly...

```
TestCase class>>shouldBeShownInTestRunner

  ^true


AbstractValidator class>>shouldBeShownInTestRunner

  ^false
```

The issue with this is that other extensions of `SUnit` could require further use of this or other messages. Instead, rename `TestCase` to `AbstractTestCase` and rearrange the class hierarchy as shown below:

```
AbstractTestCase
  AbstractValidator
  ...
  AbstractRootsForOtherExtensions
  ...
  TestCase
```

In this way, each specialized test runner can look at the right test cases by looking at the root of the test cases it knows how to run, without needing a single send of `ifTrue:ifFalse:`.

Answer to exercise 4.9. In `SUnit`, individual test case classes can be marked as abstract. A suite built off from an abstract test case does not execute tests in the context of abstract test case classes. Instead, it will run tests for all its non-abstract subclasses. So if we mark all abstract validators as abstract, they will not be executed in the test runner even if we ask to run all the tests.

We can write an `SUnit` test for this bit of behavior, and then implement the class method shown below.

```
AbstractValidator class>isAbstract

    ^true
```

Answer to exercise 4.10. The first time the author addressed this exercise was between the years 2001 and 2002, for a project that consisted of about 12 megabytes of source code. The experience was that the project started on Tuesday and was done by Friday. This includes the time to design the framework itself, which tells how much `SUnit` lends itself to validation!

Many thanks go to Lee H. Brown, the author's manager at the time, for giving the freedom to try something new.

Answer to exercise 4.11. Unfortunately, the catch all exception handler that detects errors in `TestResult` will also catch validation failure exceptions, and thus prevent any handlers we add in the refined implementation of `runCase:` from executing. Unless some heavy refactoring happens, it is not possible to take advantage of `super` in this case.

Answer to exercise 4.12. Add an instance name such as `originalObject` to validators. Then, you can implement messages like `originalValue` and so on. Also, you could add `AbstractValidator class>>validate:against:`.

Answer to exercise 4.13. Reuse the validation message finding mechanism to build a dynamically built list of methods to keep, then provide this list to the runtime packager.

Answer to exercise 4.14. Add a layer of indirection to be able to pass the object to be validated as an argument. In this way,

```
AbstractValidator>>valueIsDefined

    self value isNil
      ifTrue: [self failValidationBecause: self isRequired]
```

would become

```
AbstractValidator>>valueIsDefined

    self isDefined: self value
```

```
AbstractValidator>>adaptorValue

  ^self value value


AbstractValidator>>adaptorValueIsDefined

  self isDefined: self adaptorValue


AbstractValidator>>isDefined: aValue

  aValue isNil
    ifTrue: [self failValidationBecause: self isRequired]
```

Answer to exercise 4.15. Use polymorphism and push the determination of validity to the objects in question. Be careful when encountering objects such as proxies.

Answer to exercise 4.16. Raise exceptions of a different class, then create a resumable kind of validation failure and reuse the framework. A good name for the new exception class would be `ValidationSuggestionNotification`. Finally, implement the message `AbstractValidator>>suggest:`.

Here is an example of how it could be used in validators. For the sake of expressiveness, you may want to implement `AbstractValidator>>inform:` in terms of `AbstractValidator>>suggest:`.

```
SomeValidator>>validatePrice

  self aspect: #price.
  self valueIsDefined.
  self valueIsPositive.
  self value < 10 ifTrue:
    [self inform: self prettyPrint, ' is very convenient.']
```

Answer to exercise 4.17. Do as in the previous exercise and create the class `ValidationWarningNotification`. Implement `AbstractValidator>>warn:`. In practice, warnings can be used as follows.

```
SomeValidator>>validateFilename

  self aspect: #filename.
  self value exists ifFalse:
    [self warn: self prettyPrint, ' not found; using default.'].
  self aspect: #fileExtension.
  self value asLowercase = '.txt' ifFalse:
    [
      self failValidationBecause:
        self prettyPrint, ' must correspond to a text file.'
    ]
```

Answer to exercise 4.18. Do as in the previous exercise and create the class `ValidationConfirmationException`. As customary, provide an implementation for `AbstractValidator>>confirm:`. Then, use polymorphism to let different kinds of validation failures know a default way to be handled. This avoids many occurrences of `ifTrue:ifFalse:` where `SUnit` based validation is used.

Answer to exercise 4.19. This allows removing the instance name that holds which test message to run from test cases, and makes test result objects hold on to individual results. As a consequence, the class `TestResult` should be renamed as `TestSummary`.

Answer to exercise 4.20. Implement a message named `aspectToWidgetMap` in the user interface class. Have it answer a dictionary. Then,

```
AbstractUserInterfaceClass>>widgetForAspect: aSymbol

  ^self aspectToWidgetMap
    at: aSymbol
    ifAbsent: [self builder widgetForAspect: aSymbol]
```

Answer to exercise 4.21. Implement `defaultValidator` and `validate` in the abstract user interface class as follows.

```
AbstractUserInterface>>defaultValidator

  ^AbstractUserInterfaceValidator
```

```
AbstractUserInterface>>validate

  ^self defaultValidator validate: self
```

The rest is straightforward.

Answer to exercise 4.22. A possible approach is to add a visual clue as to the state of the object being edited. You could use, for example, a icon on the top left of each self-contained interface. A round, green light could mean everything is ok. A yellow triangle with an exclamation point would mean there are validation failures. When clicking on the icon, you could open an independent window listing the issues. In this way, the user would be able to go through the list fixing the issues, and then try again. You may even periodically update the list as validation failures are corrected.

Answer to exercise 4.23. Implement the following messages.

```
Symbol>>sentTo: anObject

  ^anObject perform: self


Message>>sentTo: anObject

  ^anObject
    perform: self selector
    withArguments: self arguments


Object>>runValidation: aSelector with: anArgument

  | validator |
  validator := self validatorClass new.
  validator object: self.
  ^validator
    run: aSelector
    with: anArgument
```

```
AbstractValidator>>run: aSelector with: anArgument

  | suite test message |
  message := Message
    selector: aSelector
    argument: anArgument.
  test := self new
    setTestSelector: message;
    object: self object;
    yourself.
  suite := self suiteClass new.
  suite addTest: test.
  ^suite run


TestCase>>performTest

  testSelector sentTo: self
```

Answer to exercise 4.24. One could use SmallLint to detect things like the singleton pattern, but what about something more elaborate such as the factory pattern?

Answer to exercise 4.25. Invoke a new validator with the corresponding set of rules and obtain a new validation result. Then, raise a resumable exception to adopt all the failures and errors from the resulting validation result. Let `runCase:` handle this exception.

Answer to exercise 4.26. At this time, there is not enough information to give an answer. Please let me know what you find.

Answer to exercise 4.28. Modify `SUnit` to have more than one `setUp` per test case. Change the test selector prefix to `measure`. Add pretty printing for each set up and the implementation being measured. Change how a test case is run so that `SUnit` itself takes care of counting iterations per second, and report the results using a new kind of exception. Create a test runner for this new framework.

For example, when the author first measured the performance for all the different implementations of `match:`, the unintended consequence was over 70

kilobytes of workspace code. With this new framework, messages like the one below could be implemented instead.

```
MatchBenchmarkTestCaseVWPlusC>>measureVWPlusC

  | localPattern localString |
  self prettyPrintForMeasurement: 'VW+c'.
  localPattern := self pattern.
  localString := self string.
  self stopWatch:
    [localPattern inlinedCaseInsensitiveMatches: localString]
```

If one wanted to compare the execution speed of different pieces of code instead, then one could implement the following message:

```
itTakesLessThan: anInteger
messageSendsToEvaluate: aBlock
```

By using a simple message send as a universal measuring rod, we do not have to adjust your performance expectations depending on the hardware used to run the benchmarks.

Eliot Miranda benchmarking procedure

To ensure the measurements are really precise and repeatable, we can take into account the following suggestions by Eliot Miranda.

First, use some object memory zone manipulation tricks.

1. Invoking a garbage collection (either `ObjectMemory garbageCollect` or `ObjectMemory globalGarbageCollect`) has the side effect of voiding the native code cache because its memory is used for the mark stack (I know, but that is the way it was and I am not changing it without better reason than it is a little strange).

2. A `SomeClass someInstance` moves all instances of `SomeClass` to the old object memory space so that the enumeration of instances is stable.

3. An `ObjectMemory someObject` moves everything in the new object space to the old object space so that the enumeration is stable.

4. Alternatively, there is a hidden zero-argument primitive 322 that flushes the new object space, e.g.:

```
ObjectMemory class>>flushNewSpace
  "Flush all objects from new space.
  Answer if the attempt was successful."

  <primitive: 322>
  ^self primitiveFailed
```

With these you can

- Ensure the code cache is empty so that no code cache compactions occur when running a benchmark.
- Ensure the new object space is empty so that no tenuring happens when allocating objects.

So to run code repeatably one wants to

- Void the native code cache (via GC, which also voids the new object space).
- Run the bechmark once to compile all the code it uses into the native code cache.
- Run the benchmark several times to actually measure.

For example,

```
| n bm |
n := 1.
bm := [Time microsecondsToRun: [n timesRepeat: ["some code"]]].
ObjectMemory garbageCollect.
bm value. "runs benchmark once, compiling all its code"
n := 1000. "for example"
bm value "to collect timing"
```

One final detail — on some machines the system clock has much lower resolution than microseconds. On these machines it is wise to start the timing as close to the start of a clock tick as possible to avoid rounding errors, e.g.:

```
| start now |
now := Time microsecondClock.
[(start := Time microsecondClock) = now] whileTrue.
"clock has now ticked..."
```

D.5 An efficient reference finder

Answer to exercise 5.1. The answer is no. If distinctions knew their name, then they would duplicate the naming facilities provided by forms. This would cause headaches should the name of a distinction change in one place but not both.

But there is a more fundamental reason. A distinction models a boundary. The abstract idea of boundary itself does not need a name. It is only when drawn on forms that the idea of naming a distinction makes sense.

Answer to exercise 5.2. While it would avoid the duplicated implementation of a few collection messages, it would also avoid inheriting all the unnecessary and sometimes undesirable collection behavior. For example, the act of distinguishing is not compatible with the idea behind `add:`.

Answer to exercise 5.3. Surprisingly, the need for `Distinction` disappears! However, it becomes necessary to name objects with arbitrary names. Either the dual property of Smalltalk objects cannot be used, or the associations in the dictionary have identical keys and values. Neither situation is acceptable.

Answer to exercise 5.4. Let the scan root be the object being weighed, and refine how objects are visited so that proper statistics are collected.

Answer to exercise 5.5. Implement a message such as `ifExcludedAdd:` in `ObjectRegistry` so that if the object passed as an argument is not already included in the receiver, then it is added and the answer is `true`; but such that if the argument is already included in the receiver the answer is `false`. Then, it becomes possible to rewrite the original method as follows.

```
PathTracer>>shouldCrossInto: anObject

  self target == anObject ifTrue: [^true].
  ^self storage ifExcludedAdd: anObject
```

Answer to exercise 5.6. From the point of view of the passenger, we could say that objects that are not interesting because they have no distinctions to traverse are *empty*. Therefore, refine `shouldCrossInto:` as follows.

```
PathTracer>>shouldCrossInto: anObject

  self target == anObject ifTrue: [^true].
  (self isEmpty: anObject) ifTrue: [^false].
  ^self storage ifExcludedAdd: anObject
```

Profiler runs show that the object that causes most of the inefficiency is `nil`. Other empty objects would be those that cause instances of `ObjectForm` to think that there are no names. In other words,

```
PathTracer>>isEmpty: anObject

  nil == anObject ifTrue: [^true].
  anObject basicClass isBits ifTrue: [^true].
  anObject basicSize > 0 ifTrue: [^false].
  ^anObject basicClass instSize = 0
```

Ideally, each object should know if it is empty or not, but that feature requires extra thought. In particular, subclasses of `nil` such as proxies, are not easy to deal with. This is also why the first comparison is written such that the receiver is `nil`.

Answer to exercise 5.7. The path tracer's target is always interesting enough to cross into, but its contents are never interesting enough to examine.

Answer to exercise 5.8. The key constraint to start the implementation from could be the following.

```
FormUI>>basicTreeModel

  ^TreeModel
    on: self form
    displayRoot: false
    childrenWith: [:aForm | self childrenFor: aForm]
    testHasChildrenWith: [:aForm | self hasChildren: aForm]
```

Answer to exercise 5.9. Add a configuration switch to the reference finder, so that when enabled, initializing the storage will collects all instances of all classes in the package `Tools.Trippy`. In this way, inspectors will be seen as known objects, and their contexts will not be examined. For example,

```
PathTracer>>initializeStorage

  self storage: self newStorage.
  self storage
    add: self;
    add: self scanRoot;
    addAll: self objectsToSkip


PathTracer>>objectsToSkip

  ^ObjectMemory allGeneralInstancesOfClasses:
    self classesToSkip


PathTracer>>classesToSkip

  | classes |
  classes := OrderedCollection new.
  self skipTrippyInspectors ifTrue:
    [classes addAll: Tools.Trippy classes].
  ^classes
```

Answer to exercise 5.10. Refine `classesToSkip` as follows.

```
PathTracer>>classesToSkip

  | classes |
  classes := OrderedCollection new.
  self skipTrippyInspectors ifTrue:
    [classes addAll: Tools.Trippy classes].
  self skipPDPDebuggers ifTrue:
    [classes addAll: Smalltalk.CraftedSmalltalk classes].
  ^classes
```

Answer to exercise 5.11. Refine the message `classesToSkip` once more as shown below.

```
PathTracer>>classesToSkip

  | classes |
  classes := OrderedCollection new.
  self skipTrippyInspectors ifTrue:
    [classes addAll: Tools.Trippy classes].
  self skipPDPDebuggers ifTrue:
    [classes addAll: Smalltalk.CraftedSmalltalk classes].
  self skipProcessMonitorService ifTrue:
    [classes add: ProcessMonitorService class].
  ^classes
```

Answer to exercise 5.12. Since `FormUI` can display form objects, it is possible to make a subclass that refines it by adding a middle button menu that makes it easier to inspect the selected object, and maybe other things such as to browse the object's class and so on.

Once this improved form interface is available, then all we need to do is to enhance the class `PathTracer` as follows.

```
PathTracer>>openBreadthPathsTo: anObject

  self
    openPaths: (self breadthPathsTo: anObject)
    to: anObject


PathTracer>>openPaths: aForm to: anObject

  PathTracerResultsUI new
    target: anObject;
    form: aForm;
    open
```

Finally, use these enhancements to rewrite the message `inspectReferencePath` in the class `Tools.Trippy.Inspector` to use the reference finder described in the chapter as shown below.

```
Tools.Trippy.Inspector>>inspectReferencePath

  Cursor execute showWhile:
    [
      PathTracer openBreadthPathsTo:
        self selectedObjectOrInspectedObject
    ]
```

D.6 A pattern of perception

Answer to exercise 6.1. One could for example try the suffix `Reflexive`. This leads to nice class names such as those shown below:

```
ReflexiveBehaviorProcess
  ReflexiveInterfaceProcess
    ReflexivePerceptionProcess
```

Answer to exercise 6.2. The Smalltalk Solutions 2006's Coding Contest was about writing a program that would play four different varieties of Pong. The contestants' programs had to interact with an HTTP server running the game in real time. The server provided the status of the game, and each of the contributed solutions had to decide how to play the game according to their criteria.

The pattern of perception applies directly to this situation. In fact, it was first implemented to solve the contest challenge. An interface object encompasses the perception and action processes that talk to the HTTP server. The eyes read through the interface's perception process and produces objects such as the playing field, the bats and the balls so these can be used by the strategies. The strategy objects produce objectives based on what the eyes perceive so that things like ball interceptions can be predicted and executed. Note that while strategies choose things like which ball to go after, the actual movement of the bats is left unspecified. The hands object takes these objectives and, by means of letting the objectives traverse the game seen as an information field, chooses which action to perform so the objectives can be met. These actions are pushed to the action process in the interface so they can be sent to the HTTP server.

Note how the Laws of Form are pervasively present throghout this design. The perception process sees a blob coming out of the HTTP server. The eyes

perceive differences in value within this blob and create objects. The strategy is a behavior map between those objects and objectives to meet. The hands simply turn around and let the objectives traverse the information field implies by the perceived objects.

With humility, the author can say that this approach is what allowed him to finish first in Smalltalk Solutions 2006's Conding Contest. In part, participating in the contest was taken as a personal challenge, but more importantly as a validation experiment for what had been learned about Laws of Form.

Based on the experience, it is the author's opinion that the assumptions made by G. Spencer-Brown seem to be correct with amazing accuracy, and directly applicable to the practice of software development.

Note that, in the first 63 hours, the pong player was implemented as a direct application of the perception pattern. After the conference, the abstract pattern was separated from its first practical application. This refactoring allowed further reduction of code and the creation of a quite refined implementation. This is the implementation that was described in the chapter.

What is noteworthy is that the decision process was "because there is no time, it has to be done right". Typically, a lack of time is used as an excuse to carelessly hack something together.

Answer to exercise 6.3. In the case of the Smalltalk Solutions 2007 Coding Contest, the problem selected was to play the memory game via a computer that had essentially two hands and a number of memory registers in the most efficient manner possible. As in 2006, the game ran in an HTTP server.

Since the computer hid the position of the cards from the participants, the problem becomes one of judiciously managing the memory registers as if they were a cache memory. The issue that participants had to deal with is that different games have different card appearance distribution, and therefore the best cache management strategy is not immediately clear.

The reference solution follows the same pattern as the one for the 2006 Coding Contest. The perception process interprets the state of the HTTP server and creates game objects such as the computer, the registers, and its contents. These objects are used by the strategy to determine what objectives to meet (such as to solve a pair of cards or to memorize more cards). The hands take these objectives, and traversing the state of the computer seen as an information field, determine which assembler instructions to execute. These may be complicated at times, since it may be necessary to move cards around in the computer so that a pair can be legally solved according to the rules of the computer. Note how separating what is the objective to meet from the way in which it is accomplished pays off in these situations because the hands do not have to second guess what the strategy already decided. Finally, the actual assembler instructions are sent to the HTTP server via the interface's action process.

The contest finals introduced several complications to the original problem. Every card figure used in the qualifier round was rotated in four different ways, it became much more efficient to uncover new cards with one of the computer hands but not the other, and the number of memory registers was reduced. Furthermore, to exchange cards from the hand registers to the memory registers, the computer imposed using an exchange register. In the finals, reading memory registers into the exchange register did not always work properly, and previously unseen cards would be read into the exchange register instead. This mimicks what usually happens during the memory game: we may think we remember what a covered card is, but we really do not. In terms of the contest, if the competing program did not catch this situation and continued execution of assembler instructions, the computer would be eventually told to solve an illegal pair. The penalty for generating illegal assembler sequences was a very expensive computer crash which would bring the overall participant score down.

The reference solution dealt with these changes in the following manner. As rotated cards match in the memory game, then rotation does not really matter for the purposes of the contest. The eyes simply ignored the rotation information when creating objects for the strategy. No further changes were needed due to this change.

Since uncovering a card became more efficient when done in a particular way, objectives were changed to generate different sequences of assembler instructions.

The strategy did not depend on the number of memory slots, so the reduced memory capability did not affect the reference solution.

Finally, the faulty memory registers would eventually work after repeated attempts to remember their information properly into the exchange register. Therefore, the objective that read a memory slot was amended to make sure the expected result would be in the exchange register by asking the eyes to perceive and distinguish the state of the computer and retrying the memory slot read as necessary.