Chapter 1

Objects, Smalltalk, Dynabooks, and Squeak: Where the Objects Come From

1 Object-Oriented Programming

Object-oriented programming is a popular slogan these days. People who aren't even sure what programming is talk about "using that object stuff" as what they want in their programs. It has become a sign of our times that the promise of technology is so great that people know the words but don't really know where the words came from or what they meant.

Objects came to be, in the way that we know them today, in the late 1960's and early 1970's. Think about what computer science was like then. Windows were made of glass, and mice were undesirable rodents. Microsoft didn't exist, and Bill Gates was still in school. Good programming in those days was "structured programming," and people talked about it in much the same way that people talk about objects today. Odds are good that even today, in your first programming classes (even if it was in an object-oriented programming language), you learned how to attack problems in a style that is like structured programming.

When you first learned to program a computer, you were probably taught to go about the process something like this:

1. Define the tasks to be performed by your program.
2. Break these tasks into smaller and smaller pieces (typically, functions) until you reach a level that you can implement.
3. Define the data structures that these functions will manipulate.
4. Design how the functions interact by defining what data each function will need to accept as input and what data should be output.
5. Group these functions into components (such as "units" or even "classes").
6. Write the code for the functions you have defined and the data structures that they will implement.

It's not important whether you tried to define abstract data types or not, or whether the order of the steps were a bit different. Generally speaking, this is not an *object-oriented process*. The key difference can be generalized in terms of the focus of the process. In the above process, you focused on what the program would *do*: The tasks and the functions. Data gets defined as something that the program acts upon. We might call this a *verb*-oriented process.

An object-oriented process starts out by defining the *nouns* in the part of the world that is relevant to the program that you are developing. You as the *object-oriented designer* identify the *objects* in the world of your program. You define what those objects know (that's their data), you define what they do (that's their behavior or operations), and you define how these objects interact. The definitions you develop are meant to *model* the world at large, or at least, the world in which your program is meant to function. That portion of the world in which your program must function is called the *domain* of the program. If your goal is a student registration program, your domain is the world of students and the registration system. The objects that you identify might include students, teachers, classes, pre-requisites, and other elements common in student registration.

An object-oriented approach is the structured approach sideways. Rather than starting with the program's task, you start with the world that the program will deal with. You still end up doing a kind of structured analysis when it comes to the structure of the behaviors of a given object. But that level of analysis is much later.

The process of object-oriented development goes something like this:

- *Object-oriented analysis*: The goal of object-oriented analysis is to define an object-based model of the domain in which the program will live. The analysis produces a list of objects with data and behaviors and the relationships between these objects. We say that the focus of object-oriented analysis is on the problem.
- *Object-oriented design*: The focus of object-oriented design is on the solution. You take what you have learned about the problem from the analysis and map it to an implementation in a language. The goal is a description of the final program in enough detail to actually implement it.
- *Object-oriented programming*: Here’s where you actually build the program you designed previously.

A good way to come to understand object-oriented development is to understand where it came from, and how and why it developed in contrast to structured development. The rest of this chapter goes back to the beginning of objects, explains how object-oriented development came to be, and describes how Squeak fits into the whole story.

2 Birth of Objects

There were two pieces of software whose ideas most influenced the birth of object-oriented programming. The first wasn't a language at all, but a brilliant graphical editor called *Sketchpad* by Ivan Sutherland at MIT in 1963. Sketchpad was the first object-oriented graphics editor, in the sense that we know it today.

You didn't just put colored bits on a canvas with Sketchpad, where the bits all merge into a single canvas once placed. You created objects that could be manipulated distinct from any other object. Even more significantly, Sketchpad allowed one to define a "master drawing" from which one could define a set of "instance drawings." Each of the instance drawings would be just like the master drawing, and if you changed the master, all the instances would change in the same way. In some ways, Sketchpad was better than even today's high-end drawing editors on much faster computers. Sketchpad was amazingly fast and offered the user the ability to draw on a virtual canvas about one third of a mile square.

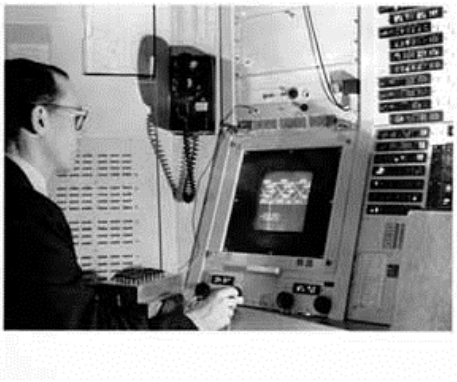

Figure 1: Ivan Sutherland using Sketchpad (from http://www.sun.com/960710/feature3/sketchpad.html)

The second piece of software was a programming langauge designed to make simulations easier to implement. It was called *Simula* developed in 1966 in Norway. Simula allowed one to define an *activity* from which any number of working versions of that, called *processes* could be created. Essentially, this is very similar to the master and instance drawings of Sketchpad. Simula, however, was a general-purpose, procedural programming language that allowed users to create these objects as a way to model the world.

Each of Simula's processes was a distinct *object*: It had its own data and its own behavior, and no object could mess with the data and behavior of another object without permission. This is important in a language designed to build simulations. Real world objects cannot mess with the internals of other objects. You cannot touch the insides of an animal, nor can you reach inside a room's wall and change the wirings. The concept that objects have their own data and behaviors, and that no other object can access that data without the objects' permission, is called *encapsulation*.

Each Simula object could act entirely on its own at the same time as the others. That is an important part of creating simulations. Two people in the same

room can both talk and act at the same time. There is no universal time share system that each gives everyone little segments of time to talk and act in. (One can almost imagine a *Universal Scheduler* announcing, "Okay, now you take a breath. Now you can say one syllable. Now you…") In the real world, things act all the time, simultaneously. Any system that supports simulation must support at least the illusion of multiple, simultaneous processes.

These two ideas met in 1966 with Alan Kay at the University of Utah. Alan was a graduate student in computer science who had just read about Sketchpad and was asked to get Simula running on the local computer. He saw how these two things were related, and in fact, were the key to creating large, complex, and robust systems. The best way to use a computer is as if it had thousands of little computers inside it — each independent, but interacting in clearly defined ways.

The metaphor of simulation was quite powerful. In a sense, all software is simulating a piece of the world, and the job of the designer is to model the world in the software. Since programmers live in the real world and seem to understand it well enough to get around in it, a program that is explicit about modeling itself on the real world has a good chance of being understandable when it's maintained later. The real world becomes the common framework between the original programmer and the later ones.

One of Alan's undergraduate majors was in biology. Biology really knows how to make complex things. Consider that a bacterium has about 120 megabytes of information in it, and it's 1/500th the size of a normal cell, and we have about 10 to the 13th power (10^{13}) of these bacteriums in our bodies. Now think about *any* man made, engineered item. How many of these scale up thousands, millions, and even trillions of times? If you take a simple doghouse, can you make a skyscraper by duplicating the doghouse a few million times, sticking them together, and expecting it to work? The Internet is perhaps the closest that any engineered artifact has reached with respect to this level of scaling, in that it has grown incredibly over many years, and still works. Alan wondered how we could make that kind of ability to handle growth and complexity the norm in software and not the exception.

Alan saw objects as the way to do it. Each object can be like a biological cell: independent, indivisible, but able to interact with its peers along standard mechanisms (such as absorbing food, expelling waste, etc.) By combining thousands or more of these cells, we can build very complex and robust systems that can grow and support reuse.

- The complexity is handled through each object performing its own functions, without undue interference from others.
- The robustness comes from the fact that the loss of an object does not damage other objects, except those that rely on the services or roles of the lost object.

The lost object can be quickly replaced, by creating a new cell in an organism and a new instance in the computer system. We need to have a blueprint for how to build the cell or instance. We call that blueprint in an object-oriented system a class. A class defines how to build new objects of the same kind.

- Supporting growth comes from using the same structuring and communication mechanism throughout. All biological systems are made of cells. If all software was made up of uniform objects, it may scale better than did software built via structured analysis. Further, objects could be combined, in the same way that organs are made up of cells. An object can contain other objects through aggregation.
- Finally, the reuse comes from each object performing its own role, with only minimal connections to other objects. If the designer does a good job and makes the software objects model well the real world objects, then those objects will probably have a future use. The same objects show up in lots of different forms in the real world: Pencils, paychecks, cars, customers. Model the objects well once, and you can use that model over and over again.

3 "A Personal Computer for Children of All Ages"

The first attempt at object-oriented programming system was FLEX, the focus of Alan Kay's dissertation. Sketchpad was an object-oriented drawing system, and Simula was a language that had the start of object-oriented programming. The FLEX machine was a complete *personal* computer, based on objects and completely programmable. It wasn't the first personal computer, but the notion of a personal computer was still radical and even invited ridicule in some corners. In the late 1960's, computers were huge, room-filling machines that were tended to by a priesthood of administrators. The notion of one of these monstrous mechanisms being at the disposal of a single individual seemed to many like an enormous waste of resources. But Alan and others already knew that Gordon Moore's now-famous law of integrated chip density was proving true[1]: Computers were going to get much cheaper. Alan wanted his FLEX machine to serve the needs of an individual, and to serve Alan as a platform to explore what one *would* do with a personal computer.

It was already possible to list several things that one might want in a personal computer like FLEX. It was a given that it needed to be programmable, with a flexible and scalable language. Certainly both Sketchpad and Simula were on that list, though neither would fit in the 16K of 16-bit words available on the FLEX.

[1] While Moore's Law strictly refers only to the number of transistors on a chip at the same price, the impact of Moore's Law is that processing power at the same price roughly doubles every eighteen months.

As Figure 2 shows, the FLEX was designed to support freehand sketching. Douglas Engelbart's NLS (oNLine Systems) was another item on the wishlist.

Douglas Engelbart had a vision for computers as "augmentation of human intellect." He wanted computers to serve to help users perceive and manage their world differently. Engelbart demonstrated NLS in 1968 and blew the audience away with his use of a pointing device (Engelbart invented the mouse), multiple paned views, outline processing, and even interactive collaboration with live video connections!

Figure 2: Kay's depiction of Flex from *Early History of Smalltalk*

While working on FLEX, Alan also learned of the pioneering work of Seymour Papert and his colleagues at MIT who were having children program computers in a programming language called Logo. Again, in those days, this was just as radical a notion as personal computers. What would *children* ever want with the powerful and gargantuan computer? The Logo developers were having children explore issues of representation (graphical and symbolic) and knowledge by having them build programs. Children would program a graphical "turtle" that would draw sketches on the screen per their commands.

With this piece, Alan saw that the role of the personal computer was that of personal dynamic *media.* The computer can be something that one could explore representations with (Logo), could draw with (Sketchpad), and even simulate anything in the real world with (Simula). Later, with Adele Goldberg, he described the personal computer as the first *meta-medium*, the first medium that could encompass all other media: Text, sound, graphics, animations, and others not yet invented. This was the vision of personal computing that Alan was exploring with FLEX.

FLEX didn't achieve all of the goals that Alan had, but it met several of them. It certainly served as a springboard to the ideas of what a *truly* personal computer might look like: small, even handheld; supporting keyboard or stylus for drawing; wireless networking, and so on. Alan called this goal a *Dynabook*, and

he talked about it as being a “personal computer for children of all ages” in his 1972 ACM Conference paper.

In 1970, Alan Kay joined Xerox’s new Palo Alto Research Center to lead the Learning Research Group where Smalltalk was created. Smalltalk was the first object-oriented programming language, in the way that we think of object-oriented programming today (as opposed to the earlier Simula). Smalltalk eventually encapsulated all of the pieces that we think about today when we think about personal computer—as well as including many of the features that were desired in the Dynabook. Smalltalk systems were the first to have bit-mapped displays, overlapping windows, menus, icons, and a mouse pointing device. Microsoft Windows, UNIX X-Windows, and the Macintosh operating system all have their roots in Smalltalk. In a very real sense, modern user interfaces have evolved hand-in-hand with object-oriented programming.

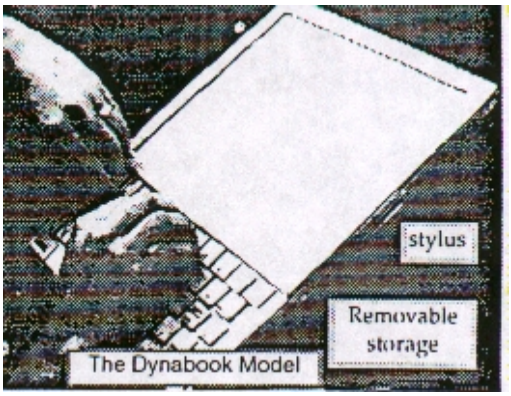

Figure 3: Cardboard model of Dynabook

Smalltalk evolved through several iterations. From Smalltalk-71 (which looked a good bit like Logo) and Smalltalk-72 (in which many of the media-oriented features were first implemented, from drawing to music programs and even iconic programming languages), Smalltalk development led to Smalltalk-76 was the first modern Smalltalk. Dan Ingalls was the main implementor of the Smalltalk implementations during this time, and the creator of Smalltalk-76. Ted Kaehler was another of the original Learning Research Group members, who built the music system for Smalltalk-72, the Logo “turtle” for Smalltalk, and Smalltalk’s object-oriented memory structure.

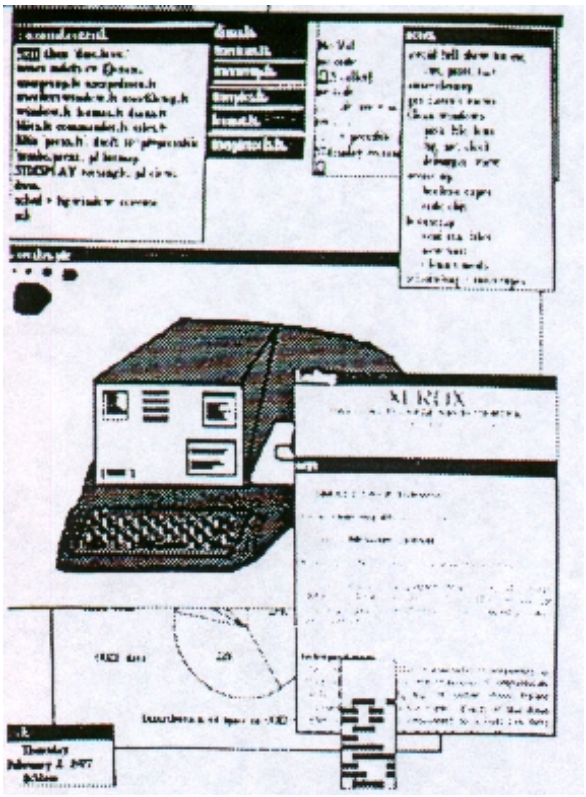

Figure 4: Smalltalk-72 User Interface (scanned from Kay, need a better version)

Smalltalk-80 was released to a handful of computer companies (DEC, Apple, IBM, Tektronix) as a test of the *portability* of Smalltalk. Smalltalk-80 was implemented as a *bytecode compiler*. Smalltalk code was actually compiled, but it was not compiled into the machine language native to the computer it was running on. Instead, it was compiled into a machine language for a machine that did not exist in hardware, a *virtual machine*. It was easy to write a small program (an interpreter) in the native machine language that would execute the virtual machine *bytecode.* This interpreter would make it appear as if the native machine really *were* the virtual machine, and thus could run any program written for the virtual machine. The advantage of this scheme is that Smalltalk-80 was highly portable. Indeed, all of these companies were able to easily create versions of Smalltalk-80 on their systems.

A basic Smalltalk-80 implementation consisted of four files:

- A virtual machine interpreter, executable on the native machine.
- An image file, which was a program in virtual machine object code that provided the Smalltalk compiler, development environment, and associated tools.

- A sources file which contained the Smalltalk source to all of the base objects in the image file.
- A changes file which contained the Smalltalk source to all of the objects that the user had added to the image.

Smalltalk-80 has become the standard from which all current Smalltalks are measured. Xerox spun off Smalltalk into a separate company, called ParcPlace, which marketed various versions of Smalltalk as ObjectWorks and later VisualWorks. Adele Goldberg, of the original Learning Research Group, shepherded the new company and wrote the definitive books on Smalltalk-80. Other versions of Smalltalk were created apart from Xerox, such as Digitalk's Smalltalk/V and Quasar's SmalltalkAgents. All have similar syntax and object structures, though the user interface code differs dramatically.

4 Back to the Future

In 1995, Alan Kay, Dan Ingalls, and Ted Kaehler all found themselves working at Apple Computer. Despite the intervening decades, they were still all interested in the vision of a Dynabook, “a development environment in which to build educational software that could be used—and even programmed—by non-technical people, and by children” (Ingalls, Kaehler, Maloney, Wallace, and Kay, 1997). While the user interface of Smalltalk had been copied and passed down through many other systems, the core ideas of the Dynabook as personal dynamic media had been lost. They considered developing with Java, but felt that it wasn’t stable enough. Smalltalk would be great, but the commercial Smalltalks at the time didn’t have the flexibility that they wanted for a real Dynabook. For example, sound had been removed since the commercial release of Smalltalk-80. They also wanted to build upon the strengths of Open Source Software that had appeared and changed radically how people thought about software development.

The idea of Open Source Software is to let the source code be freely available on the Internet and use the contributed code (bug fixes, enhancements, redesigns) to develop the code. Open Source Software is best known as being the development methodology of the Linux operating system, but many other pieces of software have been developed in the open source model, including the Apache web server and the Python programming language. Open Source has the advantage over more traditional development methologies of using the enormous creativity distributed among the programmers on the Internet to advance software.

The group at Apple decided that if the right Smalltalk didn’t exist, they’d have to build one. After all, they had done it before. And even better, they didn’t have to start over—they still had the original port of Smalltalk-80 that Apple had made years before. This was the beginning of *Squeak*, the programming language used in this book.

The philosophy of Squeak was to write everything in Smalltalk. For Squeak to succeed as open source software, all of the source code to everything, including the virtual machine interpreter, had to be freely available. If some of the code were in C and other parts were in Smalltalk (for example), the system would be harder to understand and extend. But the virtual machine interpreter, as already mentioned, had to be written in the *native* machine language. The Squeak Team (Kay, Ingalls, and Kaehler, and also John Maloney and Scott Wallace) came up with a novel solution:

- They wrote the Smalltalk virtual machine in Smalltalk. This wasn't as hard as it sounds: Adele Goldberg's book on Smalltalk-80 had already described the virtual machine interpreter in Smalltalk. They only had to type in the code and get it running.
- They wrote a small Smalltalk-to-C translator. Now, even code that had to be executed as compiled C could be originally written as Smalltalk.

Once the Smalltalk virtual machine was written in Smalltalk, they could use the translator to convert it to C. They compiled the C code to create a native machine executable. They could then run the image from the new executable. From here on out, almost all of Squeak could be written in Squeak.

In September 1996, Squeak was released to the Internet. Within five weeks, it had been ported to several variants of UNIX, Windows 95, and Windows NT. It now runs on a huge range of computers, from handheld Windows CE devices to some set-top boxes to most major computing platforms. The Squeak Team is now (as of this writing) at Disney Imagineering, but the Squeak license from Apple allows Squeak users to create anything that they want with Squeak. But any enhancements to the base system must be released back to the network, in the tradition of Open Source Software.

In the intervening years, more and more of the original Dynabook features are appearing in Squeak. Squeak has powerful 2-D and 3-D color graphics, multi-voice sampled and synthesized sounds, support for animations and even video, and tools for managing a wide variety of media formats including MIDI, Flash, JPEG, and GIF.

One of the key advantages of Squeak as a language for learning computer science is that it continues to follow the philosophy of "Everything in Squeak." Consider the image below:

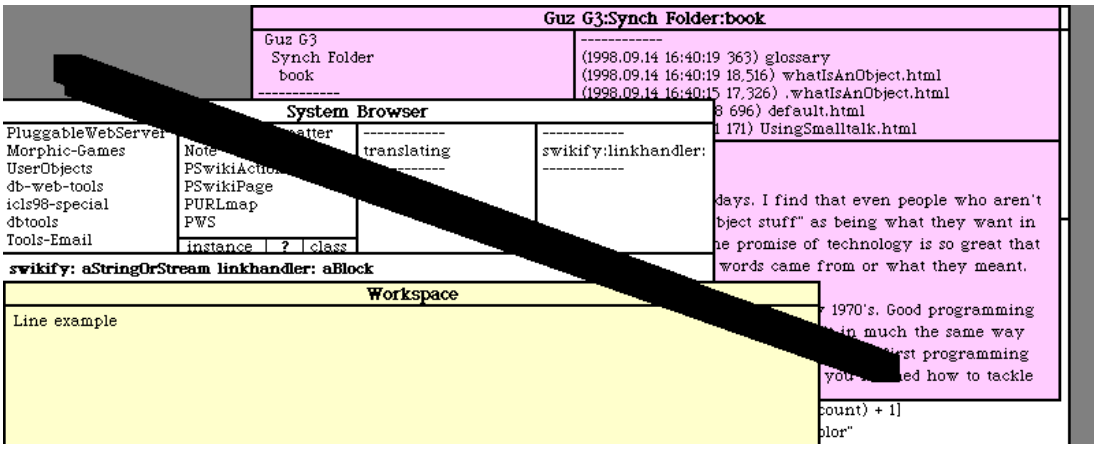

Figure 5: A Line Example in Squeak

The message **Line example** created this image. What you see here are a bunch of windows with a big fat line crossing all of them, breaking the title bars, and even messing up on the desktop. On all modern windowing systems, this is next-to-impossible. But Smalltalk-80 was the predecessor of these modern windowing systems. In Squeak, the window drawing code is written in Smalltalk! The same operation used to make that line is the same one used to make the lines of the windows, so neither one has precedence over the other.

For much that you would like to explore in computer science, from graphics to garbage collection, Squeak provides a wonderful workbench. You inherit all the great programming tools developed by the Smalltalk group, you have access to megabytes of source code that implements these features, and you can program as deep as you like while staying within Squeak. If you want to re-invent windows (or Windows), garbage collection, or even the virtual machine, all the pieces are there to do it all from within Squeak. Consider that the VM-in-Smalltalk is not just input to the translator for C output—it's also executable Smalltalk that you can use in debugging new variations on the virtual machine. Of course, running bytecodes on top of a bytecode interpreter that is itself interpreted on top of a bytecode interpreter is fairly slow, but it's a much nicer experimentation environment than dropping down into machine language debuggers. Dan Ingalls, in the OOPSLA paper that introduced Squeak, wrote that one of the goals for Squeak was to allow anyone to understand the whole system from a single language:

> Squeak stands alone as a practical Smalltalk in which a researcher, professor, or motivated student can examine source code for every part of the system, including graphics primitives and the virtual machine itself, and make changes immediately and without needing to see or deal with any language other than Smalltalk.

5 Common Ancestry of Other Object-Oriented Languages

Back in 1979, a separate thread of the object-oriented programming languages story was launched by Bjarne Stroustrup, who wanted to create a highly-efficient version of Simula. Bjarne worked at Bell Labs, where the programming language C had been invented. He created several versions of a programming language like C but with object-oriented extensions. In 1984, the language C++ was born.

C++ and Smalltalk both started from the ideas of Simula, but they are very different approaches. C++ is a compiled language uses a traditional notion of functions and stack-based scoping, while Smalltalk is a dynamic language (feels more like an interpreter than a compiler) and has persistent objects that have nothing to do with what function is in scope. C++ is a strongly typed programming language, in that all variables have a data type associated with them and only values of the right type can be stored in these variables. Smalltalk has no type declarations at all, and all storage is managed automatically by the system using a process called *garbage collection*. You can't explicitly destroy any variables in Smalltalk.

Java is a more recent programming language that starts merging these two threads of object-oriented programming. In 1991, Sun Microsystems began an internal project to produce a language that could run on intelligent consumer electronic devices -- everything from set-top boxes to toaster ovens. James Gosling created the programming language Oak through this project, as a highly portable, object-oriented programming language. When the Web came along, it was obvious that a highly-portable language could also be used to send around executable code over the Internet. The language, by then named *Java*, had a new focus. Java was announced to the world in May, 1995. Java looks very much like C++, and is even more strongly typed. Yet, Java also has features of an interpreter, it uses bytecode compilation, and it offers automatic storage management.

Exercises

1. Given all of the above, what are the key features of an object-oriented system? Think about the various analogies have been used: Objects as cells, software as simulation, objects as little computers. What do each of these analogies point out as key features?

2. What does inheritance buy you as a software designer? Do biological cells have inheritance? It is interesting to note that the earliest forms of Simula and Smalltalk did not have any form of inheritance.

3. Do biological cells have classes?

4. What is the most important aspect of object-oriented programming? Several have been identified here: Inheritance, encapsulation, aggregation. Which do you argue is most important?

References

The discussion in this chapter only touches on the influences and characters that led to object-oriented programming, Smalltalk, and Squeak. The below references provide much more detail.

For a great discussion of the history of Smalltalk, see

Kay, A. C. (1993). The early history of Smalltalk. *History of Programming Languages (HOPL-II)*. J. E. Sammet. New York, ACM: 69-95.

Bjarne Stroustrup's history of C++ appears in

Stroustrup, B. (1993). A History of C++. *History of Programming Languages (HOPL-II)*. J. E. Sammet. New York, ACM: 699-769.

Alan Kay's Scientific American article where Joe the Box first appeared is in

Kay, A. C. (1977). "Microelectronics and the Personal Computer." *Scientific American*(September): 231-244.

My favorite paper that describes the Xerox PARC vision of the personal computer is

Kay, A., & Goldberg, A. (1977). Personal dynamic media. *IEEE Computer*, March, 31-41.

The Squeak *Back to the Future* OOPSLA paper is at ftp://st.cs.uiuc.edu/Smalltalk/Squeak/docs/OOPSLA.Squeak.html. The original reference is:

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., & Kay, A. (1997). Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself, *OOPSLA'97 Conference Proceedings* (pp. 318-326). Atlanta, GA: ACM.

Probably the most influential paper in the Open Source movement has been Eric Raymond’s *The Cathedral and the Bazaar* which is available at http://www.tuxedo.org/~esr/writings/cathedral-bazaar

The main Squeak site is http://www.squeak.org. The main discussion site for Squeak is http://minnow.cc.gatech.edu/squeak.1.