6 Object-Oriented Programming

Teacher: Object-oriented programming is the dominant programming paradigm today, so I suppose we all have some intuitive idea about it. But it may be useful to make this intuition more explicit. Can someone give a quick summary of what the main concepts of object-oriented programming are?

Archimedes: I hope we will get a chance to talk about the various interesting past takes on object-oriented programming, but there are four main concepts that they all share. Those are dynamic lookup, abstraction, subtyping and inheritance.[1]

Diogenes: Sorry, but this is just too abstract for me! I thought object-oriented programming is about programming with objects and classes. Can you explain what you mean by all those terms?

Archimedes: Certainly! Dynamic lookup means that behaviour is determined at runtime, abstraction means that objects can hide their implementation details, subtyping means that you can use one object in place of another if it has all the required functionality and inheritance means that you can reuse object definitions to define new objects. Object-oriented programming based on classes is one way of implementing these.

Socrates: This is what the textbooks today say, but it is not right. Object-orientation is all about message sending. To build a system that can scale and evolve, it is not useful to constrain what the system is made of. This will need to change anyway as the system evolves. Objects and classes are the uninteresting aspect of object-oriented programming. We need to instead think about the communication within the system.

Pythagoras: Sorry, but I think that the concepts described by *Archimedes* are more comprehensible than this vague idea. They are also consistent with the origins of object-orientation. You can find all of these four concepts in some form in SIMULA 67, which was in many ways the first object-oriented language.

Socrates: SIMULA was great, but it was just a better old thing. It was an improvement over Algol, but it was based on the same basic old ideas. The authors of Smalltalk understood that you can see SIMULA as almost a new thing and started to look for the actual new thing.[2]

Archimedes: This is a nice lofty explanation, but it does not really tell me much. Smalltalk is different from SIMULA in that it is dynamically typed, but it also has objects and classes, so I do not think calling it a 'new thing' is justified...

Teacher: We should be careful about incommensurability in our debate. We know already that the two of you are looking at the problem of programming from a different perspective. To understand each other better, it would help if you, *Socrates*, could explain your thinking about Smalltalk from a broader perspective.

Socrates: Smalltalk was born as the software side of Dynabook, 'a personal computer for children of all ages'.[3] The aim was to make the software highly adaptable, which is the only way to build something that can accommodate the diverse ideas and needs of children and other users. Smalltalk's focus on messaging, reflection and late binding is what makes it possible to adapt it to your needs. It becomes a meta-medium that you can turn into any specific medium you desire.[4]

Diogenes: I can certainly appreciate the self-modifiability of Smalltalk. It is quite amazing. But you do not need objects for that. You can get the same capabilities in Lisp. In a sense the UNIX and C ecosystem also gives you this capability!

Xenophon: There must be something about object-oriented programming that made it so widely popular. I suspect this has more to do with how object-oriented languages can model the real world than with their obscure technical characteristics.

Teacher: Let's focus on this question for a moment. How did object-oriented become so popular? Did it come as a major paradigm shift in how large software is built?

Archimedes: I think it was a much more gradual transition than *Socrates* suggests. Many were looking for better ways of creating large software systems in the 1970s and work on decomposing software systems or abstract data types[5] introduced many of the same ideas as object-oriented programming.

Pythagoras: This makes me think that the textbook definition of object-oriented programming that you started with may well be a historical accident rather than some essential characterisation. Wouldn't it be fair to say that abstract data types are a kind of object-oriented programming?

Archimedes: We could spend the whole remainder of this lecture discussing this question![6] Technically speaking, original implementations of abstract data types were not object-oriented, because they did not support all of the four features from my earlier definition, but I think they share the spirit of object-orientation.

Xenophon: In the late 1970s, what counts as object-oriented was far from settled. The Ada programming language, designed at the US Department of Defense, was seen as object-oriented, even if it did not have inheritance. And the designers had good reasons for seeing inheritance as a dangerous feature for high-integrity systems![7]

Pythagoras: This is an interesting historical observation! I do not think inheritance is necessarily a problem, but it may be pointing at an issue with many object-oriented systems where the behaviour of components is not defined precisely enough. This is because the object-oriented paradigm threw the baby out with the bathwater when it moved away from the mathematical thinking of Algol! Fortunately, the Eiffel language brought this back in the 1980s and made it possible to specify the behaviour of components precisely in the form of contracts.

Socrates: The problem is that 'software engineering' is an oxymoron. It is futile to try to impose rigorous structure on software development because we are still building primitive structures with primitive tools. The only way to build software is to keep the programming system as dynamic as possible, so that you can adapt in response to unexpected needs and changes.

Teacher: It seems that there is an interesting tension between whether we should keep the system more flexible and easy to adapt or more rigid and more precisely specified. Can these two directions be used to frame most of the important developments around object-oriented programming?

Diogenes: There is another notion of flexibility that we should not ignore. It is more concerned with the machine the program runs on. When you know what you want to do, the language needs to let you do that! This is exactly what you get in C++, which extended the C language with object-oriented programming in the 1980s. It is a language for those who take programming seriously and lets you write stable and efficient code that behaves in clearly defined ways. Of course, you need to know what you are doing, but if you do not know that, no tools can help you.

Xenophon: Hearing our discussion, I want to get back to the question that *Teacher* raised earlier, whether object-oriented programming was a major shift in how software is built. I think the answer is that it was not and anyone listening to our discussion can see why! We are talking about language features and technical details, but keep ignoring other aspects of software engineering such as requirements gathering or testing.

Diogenes: Well, we are talking about programming. I don't think your managerial methodologies for 'how to program if you cannot'[8] are relevant here.

Archimedes: To be fair, understanding what the customer needs should be programmer's concern and object-oriented programming does, in fact, help with that.

Xenophon: I'm curious to hear how...

Archimedes: One idea made possible by object-oriented programming is to use English language description of a program as the starting point for your design. If you analyse the description and look for nouns and verbs, nouns will become the classes in your design and verbs will become their operations.[9] This needs to be done more carefully, but it is a powerful guiding principle for object-oriented design.

Xenophon: This is a clever idea, but it still covers only the design and implementation phases of the software development lifecycle. You still need to embed object-oriented design within a more complete process.

Socrates: If you look at how people developed software for the early commercial Smalltalk systems in the 1980s, you will see that they often did not need this kind of heavyweight processes. They were able to leverage the dynamic nature of the system.

Xenophon: How did they manage to get anything to work then?

Socrates: A famous example is the work of Ward Cunningham and Kent Beck at Tektronix research labs. They would come up with an idea, quickly prototype it over a few days or a week, reflect on the prototype and then either spend another week on the project or pursue some other direction.

Xenophon: That sounds like an Agile development methodology to me. Now, I'm not personally a fan of those ideas, but I'm sure they are useful in a more exploratory environment like an industrial research lab. But still, they are processes that control the rest of the application lifecycle!

Archimedes: Mind you, the Agile manifesto was published in 2001 and we are talking about object-oriented programming and software engineering as it was done in the 1980s, so it is interesting that you are making this connection.

Socrates: There is definitely a connection. Many of the authors of the Agile manifesto knew Smalltalk. This includes Ward Cunningham, who I just talked about. It is not surprising that programmers used to the live editing and rapid feedback of Smalltalk systems would want those features in other environments. Methodologies like Extreme Programming are a way of keeping the Smalltalk spirit in languages like Java or C++.

Teacher: Looking at the origins of object-oriented programming was definitely revealing and it prepared the ground for my last question. I will intentionally keep this short. Has object-oriented programming succeeded?

Archimedes: Well, for me, the main idea is that of managing the complexity of software. This is what you find in the early work on decomposing software into components, abstract data types and, eventually, object-oriented programming. In this, I think object-orientation has been a success. The scale of what we are able to build today cannot be compared with the scale of software that we were able to build in the 1980s. And object-orientation played a key role in this step change.[10]

Xenophon: I would like to add that object-oriented programming also sparked a valuable change in the management of software development. We did not talk about this, but object-orientation led to the development of the Unified Modelling Language (UML) in the 1990s. UML was not concerned with just architecture and coding. It transformed the entire software development lifecycle through things like the Use Case diagrams.

Socrates: You can define success in different ways. Object-oriented programming succeeded in that it is everywhere. But I think it has not succeeded if we ask whether it achieved its original aims. It was about creating systems that are flexible, can be adapted by their users and qualitatively extend the notions of reading, writing, sharing and publishing of ideas.[11] In this sense, the vision behind object-oriented programming has failed to materialise.

Teacher: Do you have any hopes for the future?

Socrates: It is hard to see how this vision can come true under the current economic conditions, but I still have a hope. There are people who believe in the original vision and are working on making it happen. The Squeak and Lively Kernel projects are two fantastic examples of this![12]

Diogenes: It is remarkable how technically close are those systems to the original Smalltalk implementations from the 1970s. I wonder if the same characteristics can be achieved in other ways, or even in the context of our current computing infrastructure...[13]

6.1 The Computer Revolution Hasn't Happened Yet

In 1997, the object-oriented programming language Java that would soon dominate the industry was publicly available for just two years. It was already talked about so much to cause a reader of a professional magazine to complain about the magazine's contribution to 'all the Java hype'.[14] Meanwhile, C++ was 12 years old and one year away from the first ISO standardisation. The object-oriented programming paradigm was the industry standard and thousands of academics and practitioners regularly gathered at the annual conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).

The keynote speaker at the annual conference in 1997 in Atlanta was Alan Kay, the creator of the programming language Smalltalk that gave object-oriented programming its name. In 1997, Smalltalk was 25 years old, but it had become a very different thing from the Smalltalk of the 1970s that I talked about in Chapter 3. Smalltalk was now available in a range of commercial products with names like VisualSmalltalk Enterprise and the Smalltalk tools market was worth $56 million in 1994.[15] The programming language and environment may not have been so different from that of the 1970s, but its surrounding culture has changed. Whereas object-oriented programming in the Smalltalk of the 1970s was the product of the hacker and humanistic cultures, object-oriented programming in the industry-scale Smalltalk systems of the 1990s was a much more engineering and managerial undertaking.

The keynote delivered by Alan Kay highlights this cultural shift. An oft-quoted passage from the talk condemns much of 'what is actually being done in the world under the name of object-oriented programming (OOP)':

> I've been shown some very, very strange-looking pieces of code over the years by various people, including people in universities, that they have said is OOP code, and

written in an OOP language – and actually, I made up the term object-oriented, and I can tell you I did not have C++ in mind.[16]

In a less frequently quoted follow-up passage, Kay makes it clear that he is not criticising just the C++ language, although the reference certainly elicits much laughter from his audience. Kay admits that he has 'many of the same feelings about Smalltalk'.

Alan Kay remains loyal to the ideals of the humanistic culture of programming. There are two aspects of his keynote that illustrate this well and highlight the cultural difference between the object-oriented programming of the 1970s and 1990s. First, the keynote includes an eclectic mix of references and metaphors, typical for the humanistic culture of programming. Kay uses parallels with molecular biology to suggest how complex systems can emerge. He makes references to the media theorist Marshall McLuhan and the philosopher Arthur Schopenhauer to discuss the adoption and evolution of object-oriented programming. Last but not least, he concludes with a story about a pipe organist E. Power Biggs. When asked how to play a piece of organ music written for some of the largest organs on a 'dinky little organ', Biggs says 'Just play it grand. Just play it grand.' Similarly, Kay wants to inspire his audience to play their systems grand.

Second, Kay's keynote is not about the history of Smalltalk, its successes or particular engineering issues. It is about finding new ways of thinking about computers and building systems that would scale 'by a factor of trillions'. This is not a mere rhetoric figure. Kay believes that we need to think extremely large scale in order to design systems of the future. He quotes one example where this happened. The ARPANET, which became the Internet, was built with the same set of beliefs in mind and was inspired by J. C. R. Licklider's vision of Intergalactic Computer Network. As Kay notes in his talk, the architecture of ARPANET enabled it to expand 'by about a factor of a hundred million' between 1969 and 1997. He encourages the audience to learn from this example. 'That is the metaphor we absolutely must apply to what we think are smaller things'.

Kay uses the issue of scale to point at the 'deepest thing' that he wants his audience to take away from his talk. It is that 'we don't know how to design systems yet'. This point is going very much against the belief, broadly accepted at the time, that object-oriented design is ultimately the right way of building complex software systems. Smalltalk has made its contribution to the search for a better answer, but it was not a particular object-oriented language design. 'Pretty much the only thing' that Kay is proud of is that Smalltalk 'has been so good at getting rid of previous versions of itself'. It enabled the basic iterative search for better answers, but when it went commercial, it ceased changing and stopped being used 'to bootstrap the next thing'.

To a member of the humanistic culture, imagining the future of computing, the keynote is a stimulating inspiration. To an engineer or a manager who thinks about building software systems over the next couple of years or even a decade, Alan Kay remains the same researcher described by Stewart Brand in the 1970s: brilliant, uninterested in conventional goals and with plenty of time for messing around.[17]

Although Smalltalk was not the first programming language that featured some form of object-oriented programming, Alan Kay's 1997 keynote is an appropriate opening for this chapter for two reasons. First, although the term ‘object’ have been used by others even before Smalltalk, the notion of object-orientation as understood by Smalltalk largely ended up being the one that computer science students are taught today. Second, the keynote illustrates the cultural shifts around object-oriented programming that I will follow in this chapter. But before I get to Smalltalk and its contributions rooted in the humanistic culture, we should start with the mathematical origins of object-oriented programming in work on computer simulations.

6.2 A Language for Describing Discrete Event Systems

One theme that we will encounter repeatedly in this chapter is modelling of the real world. Proponents of programming languages that we now call object-oriented often claim that objects provide a natural way of representing real-world entities in a computer program.[18] This argument suggests a way of thinking about programming that is one step further from the idea of high-level programming languages developed in the 1960s. The first programming systems made it (somewhat) easier to enter instructions for the computer, while high-level programming languages made it possible to model computations. Object-oriented languages, or so it is claimed, make it possible to model not just computations, but the behaviour of the real world itself. The theme of modelling is present directly in the ancestor of object-oriented programming languages, SIMULA. The language, as the name suggests, was explicitly developed for creating computer simulations. The idea of modelling the real world was not a post hoc claim, but a design motivation.

SIMULA was created by Kristen Nygaard and Ole-Johan Dahl at the Norwegian Computing Center in Oslo in the 1960s. Already during his military service at the end of the 1940s, Nygaard carried out numerical computations for Norway’s first nuclear reactor, but he later turned his attention to Monte Carlo methods and operations research. This often required creating computer simulations of physical systems that consisted of multiple entities or processes that interact and evolve over time. Such simulations were useful in multiple areas, including science, military research and optimisation of manufacturing processes and there was a growing interest in creating computer simulations throughout the 1950s. The usefulness of dedicated programming languages for creating simulations was only recognised with the turn of what Richard Nance[19] calls ‘The Period of Search’ into ‘The Advent’ around the year 1960. As pointed out by Nance, the developments of the early simulation programming languages ‘took place concurrently without a clear progression of conceptual convergence’. SIMULA is no different, in that it was developed largely independently of other work in the area.[20]

Nygaard started working on a programming language for writing simulations in 1961. He developed the first ideas on how the programming language should look by the following year, when he described the project progress in a letter to a colleague:

> The status of the Simulation Language (Monte Carlo Compiler) is that I have rather clear ideas on how to describe queueing systems, and have developed concepts which I feel allow a reasonably easy description of large classes of situations. … The work …

could not start before the language was fairly well developed, but this stage seems now to have been reached. The expert programmer who is interested in this part of the job will meet me tomorrow.[21]

The expert programmer that Nygaard was going to meet was Ole-John Dahl, who worked on an Algol-like programming language at the time and joined Nygaard at NCC in 1963. The two became long-term collaborators, working on SIMULA so enthusiastically that a new employee at NCC once worriedly reported that 'two men are fighting violently in front of the blackboard in the upstairs corridor'.[22] In retrospect, Nygaard regarded the letter as naively optimistic. Creating the first version of SIMULA took both serious technical effort and 'entrepreneurial maneuverings'.[23] Two years after writing his optimistic letter, Nygaard managed to secure support from the Univac, which was one of the eight major computer manufacturers of the era. In 1963, the implementation work on SIMULA could finally get started.

What is interesting for this book is the cultural background of the SIMULA programming language. Its origin is firmly rooted in what I refer to as the mathematical culture. Both Kristen Nygaard and Ole-John Dahl received master's degrees in mathematics. More importantly, the SIMULA programming language was based on Algol, a prominent programming language from the mathematical culture of programming that I discussed in Chapter 2. This may have partly been for practical reasons. Algol was popular at the time in Europe and Ole-John Dahl had experience with implementing an Algol-like language, but it was also an intentional design choice. 'The elegant and powerful concepts of this language appealed to Dahl and Nygaard and made it, in their opinion, the perfect match for SIMULA'.[24]

There are a number of other aspects of SIMULA design that align it with the mathematical culture of programming. In a retrospective history of the language, Nygaard and Dahl[25] describe their encounter with SIMSCRIPT, which included unsafe pointers, as a 'cultural clash' that made the design goals of SIMULA clear. It 'had to provide programming "security" to the same extent as Algol 60 itself: Any erroneous program must either be rejected by the compiler (preferably), or by runtime checks (if unavoidable), or its behaviour must be understandable by reasoning based entirely on the language semantics, independent of its implementation.'[26]

Dahl made the link to mathematics even more explicit in his talk 'Programming languages as tools for the formulation of concepts' presented at a Congress of Scandinavian Mathematicians in 1969.[27] He likens 'the art of programming digital computers' to 'the science of constructive mathematics' and argues that the computer plays the role of the set of primitives and axioms. Last but not least, Nygaard and Dahl can also be associated with the mathematical culture through personal connections and collaborations. One of the foremost members of the mathematical culture, Donald Knuth, wrote an introduction for their paper on SIMULA published in the *Communications of the ACM*,[28] starting 'a long and lasting friendship',[29] while Dahl wrote a chapter to a later book on *Structured programming*,[30] with another foremost member of the mathematical culture, C. A. R. Hoare.

The design of SIMULA evolved significantly over time. The two major versions were SIMULA I (originally called just SIMULA), developed between 1961 and 1965 and SIMULA 67, developed subsequently. However, even the original SIMULA I went through a number of designs. I will look at just two of those here.[31] In the first stage, a simulation in the SIMULA language was modelled as a *system*, consisting of a fixed number of active *stations* and a variable number of passive *customers*. Those concepts were also the keywords of the new syntactic structures added to the underlying Algol programming language. In this model, the code associated with stations received customers, checked and updated their state and, eventually, routed them to another station.

Experience with the first stage version prompted Nygaard and Dahl to look for a more general model and they eventually unified the active stations and passive customers under a single notion. The revised version of the language[32] replaces stations and customers with a single language construct called *activity* declaration. The language is still far from being what we might now call a general purpose object-oriented language, but we can already find traces of important object-oriented ideas that would take shape in SIMULA 67:

> The description of a process is called an activity declaration. The concept of an 'activity', which is a class of processes described by the same declaration, is distinguished from the concept of a 'process', which is one dynamic instance of an activity declaration.[33]

An activity declaration is what would later become a *class*. It consists of some processing logic that can be long-running and simulates the activity alongside with attributes that represent parameters and the state of the activity. The description in the paper mentions the term class, but only in its ordinary English-language sense. The paper does not yet talk about objects, but it already uses the term instance. At this stage, processes (i.e., instances of an activity) could be assigned to a special kind of variables called *elements*. For implementing any interesting simulations, processes also need to be able to access the attributes of other processes. In SIMULA I, this was known as 'remote accessing'.

Remote accessing, being an equivalent of member access in modern programming languages, seems surprisingly cumbersome in retrospect. A reference to a process in SIMULA I does not keep track of what activity the process is an instance of. In modern terms, it is untyped. But the authors of SIMULA I wanted to guarantee safety and so remote accessing had to be done using a construct that provides code for all possible kinds of activities:

$$\begin{array}{ll} \texttt{inspect}\ \langle\text{element expression}\rangle & \texttt{when}\ A_1\ \texttt{do}\ S_1 \\ & \ldots \\ & \texttt{when}\ A_n\ \texttt{do}\ S_n \\ & \texttt{otherwise}\ S \end{array}$$

As the authors note in their historical reflection on SIMULA, try to compute $X.A + Y.A$ using the `inspect` statement![34] Another notable technical idea that made SIMULA I possible was a change in the treatment of code blocks from Algol. Blocks in Algol represent sequences of statements and were a key innovation that enabled structured programming. Blocks correspond to some part of program logic. They can be nested and form the body of loops or conditionals. In Algol, blocks merely structure the program code. The execution entered a block, executed all the statements in the block and then left the block, whose state could then be discarded. However, in SIMULA, newly instantiated activities needed to outlive the call, because they could represent a process that has been started and should continue running. This caused a major headache to the SIMULA designers. There was no hope of implementing SIMULA as an Algol preprocessor as it would involve 'breaking up the blocks, making local variables nonlocal, etc'.[35] Instead, Dahl set out to make a non-trivial change to Algol, replacing the stack-based treatment of blocks with one based on dynamic memory allocation. The allocated blocks also needed to be deallocated when they were no longer used. For this, the authors used a reference counting mechanism with a 'last resort' garbage collector.[36]

SIMULA I included many of the ideas of later object-oriented languages, but in a form that was closely linked to the problem of programming simulations. We can see traces of object-oriented programming in the language, but most would not call it object-oriented yet. But in the summer of 1965, Nygaard and Dahl started thinking and talking about 'new, improved SIMULA',[37] which would eventually became SIMULA 67. Although the language was defined in 1967, it took several more years for the first compilers to appear. In a history of SIMULA written later by the authors, we can see both how the notion of a 'programming language' has became a standalone entity (as discussed in Chapter 2) and a reflection on the assumptions of the mathematical culture by someone who went through the tedious process of actually implementing a programming language:

> As language designers we often feel that people believe that one has 'succeeded' and that a language 'exists' as soon as it is defined, and that the subsequent implementation is a mere triviality … In fact it is when the language is reasonably well described and thus the implementation task defined, that the main, tedious, and frustrating part of the struggle for a language's existence really starts.[38]

The design of SIMULA 67 aimed to address a number of shortcomings of SIMULA I. Many of those resonate with what a programmer today may think about SIMULA I. The special variables referring to processes (elements and also sets) were cumbersome and the remote accessing mechanism was hard to use. The authors encountered many 'processes' that did not have their own logic, but represented data with multiple associated procedures and they also felt the need for a mechanism for sharing common data and actions between related processes.

An important inspiration for the new design of SIMULA came from a fortuitously timed meeting. In September 1966, Ole-John Dahl gave a lecture on SIMULA I at a Summer School on programming languages organised by the NATO Advanced Study Institute in Villard-de-Lans. Among the five speakers was also C. A. R. Hoare, whose lecture focused on the topic of 'Record Handling'. Hoare was interested in finding a way of adding a mechanism for representing rich data structures to Algol 60, an issue that I discussed in Chapter 5.

Hoare talks about records representing objects with several attributes. He points out that 'objects of the real world are often conveniently classified into a number of mutually exclusive classes, and each class is named by some collective noun, such as "person" or "bankloan"'.[39] Hoare uses the same model for records, which are 'grouped into mutually exclusive record classes' that determine the fields of the record. For example, a record class declaration for the aforementioned 'bankloan' looks as follows:

```
record class bankloan(integer loan number,principal;
                     real rate of interest)
```

Hoare discussed a number of other noteworthy ideas. He introduced a reference type, such as `reference(bankloan)`, representing a reference to a record of a given class. This makes it possible to access members of a record safely without runtime checks, and without the cumbersome remote access construct of SIMULA. His proposal also makes it possible to split the values of a record into multiple mutually exclusive *subclasses*.

Hoare and Dahl likely discussed the issue of record handling at length at the NATO Summer School. The revised version of Hoare's lecture notes, published after the event, includes an extensive report on how activities of SIMULA I (which have attributes that can be accessed via remote accessing) can be seen as records.[40] Nygaard and Dahl were likely inspired by the meeting and they have reported that 'much time was spent during the autumn of 1966 in trying to adapt Hoare's record class construct to meet our requirements, without success'.[41]

Eventually, they succeeded and produced a design for SIMULA 67 that combined the programming model of SIMULA I with many of the ideas from Hoare's work on record handling. Nygaard and Dahl adopted the term 'class' for the activities and also replaced the term 'process' with a more general term 'object'. They introduced a reference type akin to that of Hoare, making it possible to easily access attributes of an object of a known class. The trickiest construct to support was that of subclasses. This was eventually implemented using a mechanism called prefixing, where a class can be prefixed with another class, inheriting its attributes and procedures. Nygaard and Dahl do not themselves use the term 'inheriting' and, instead, write about 'building the properties of the prefix into the objects defined by the new class declaration'.[42] Figure 6.1 offers a glimpse of what SIMULA 67 programs looked like, illustrating both prefixing and virtual methods, which were a 'last minute extension' to the language.[43]

Looking at SIMULA 67, a present-day programmer would recognise much of the terminology and many of the constructs of a general-purpose object-oriented programming language. But the flexibility of SIMULA had not generally been recognised at the time. In an influential book *Programming Languages: History and Fundamentals*, written by the programming pioneer and co-author of COBOL, Jean Sammet, SIMULA appears in the section IX.3.1.7 in a subsection on specialised languages for discrete simulation. A couple of languages from this group are included, but with some reluctance. Sammet explains the reasons for her relatively limited account of the topic. First, 'usage [of simulation programming languages] is unique and presently does not appear to represent or supply much carry-over into other fields'.[44] Second, 'unless an individual has actually tried to simulate a system or a process, he is not apt to appreciate or completely understand the importance or subtleties of the facilities being provided'.[45]

```
Process class machine (inq. outq. own crane);
        ref(head)inq. outq;ref(crane)own crane;
        virtual: procedure service:
        begin
        ref (order) served; real setup time;
        procedure service;
            hold (setup time + served.processing
            time);
work:   if inq. empty then passivate
        else begin served: - inq. first;
        served. out; service; served. into(outq);
        activate own crane delay message time end;
        go to work
        end machine;
```

Figure 6.1 Incomplete SIMULA 67 illustration, showing 'how a simple "machine" in a job shop simulation may be described'.
The code defines a class `machine`, using the generic `Process` class as its prefix. 'Since the procedures [sic] `service` is specified as a "virtual quantity" it may be redefined in subclasses to "machines"'.[46]
(Source: Dahl et al. (1968))

It took some time, but the power of the constructs introduced in SIMULA was eventually widely recognised and they have certainly carried over into other fields. The key role in this was, however, played by a quite different culture of programming than the mathematical one from which SIMULA has originated.

6.3 Past Ghosts and Present Spectres

It is now time to slowly return to Smalltalk, the programming language from the opening of this chapter. According to the folk history of object-oriented programming, while SIMULA was the first object-oriented programming language, Smalltalk was the programming language that gave the programming style its name and made it popular. I will question some of those oversimplifications in the next section, but there is no doubt that Smalltalk has played a major role in shaping the object-oriented programming paradigm.

As is often the case with work that has roots in the humanistic and hacker cultures, it is difficult to disentangle exactly the network of influences that has shaped Smalltalk. The early writing about the system is sparse and focuses more on its humanistic aims of building 'a personal computer for children of all ages', than on the formal academic design influences. My starting point are thus the first-person retrospectives, written by Alan Kay and Dan Ingalls,[47] which document the cultural and social context and also technical influences on Smalltalk. According to Kay, the design of Smalltalk was influenced, through his preliminary work on the FLEX machine, by a number of systems that were ‘almost a new thing’ from the 1960s, including SIMULA, but also Ivan Sutherland’s Sketchpad, discussed in Chapter 3, and two systems that Kay encountered as a programmer in the US Air Force.[48] Looking at the publications on the FLEX machine suggests a number of other less widely accepted influences.

I already talked about the ARPA community and Xerox PARC that provided the context from which Smalltalk emerged in Chapter 3. Kay became a part of the community when he joined the graduate school at the University of Utah. While the computing world outside the ARPA community has been dominated by large batch-processing computers and some early terminal-based time-sharing systems, Kay lived in a world where computers were used by children (Papert’s Logo system), their role was ‘augmenting human intellect’ (Douglas Engelbart’s and Vannevar Bush’s visions), they had interactive graphical displays (Sutherland’s Sketchpad project) and were used to build live collaborative environments with video-conferencing (Douglas Engelbart’s oN-Line System). Imagining a personal computer ‘which could be owned by everyone and could have the power to handle virtually all of its owner’s information-related needs’[49] still required a stretch of the imagination, but it was at least conceivable.

The first source of ideas that Kay mentions is a system for transporting files at the US Air Force for the Burroughs 220 computer. The data was stored in a self-descriptive format where the actual records were accompanied with procedures for reading and manipulating the data, that is, using the grouping of data and procedures that is typical for object-oriented programming. Although likely unbeknown to Kay, the Air Force was also an unexpected source of thinking on personal computers. A report on the future computer technology, presented in 1965, envisioned what almost seems as the military version of the Dynabook (Figure 6.2). Fittingly, the envisioned computer was the size of a cigarette package rather than the size of a paper notebook:

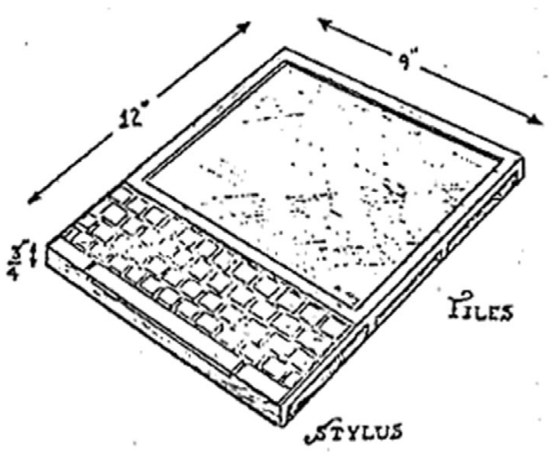


Figure 6.2 A sketch of ‘Dynabook’ from ‘A personal computer for children of all ages’.[50] (Source: Courtesy of Alan Kay. Reproduced with permission.)

> [We] will be able to build a machine the size of a cigarette package. As in civilian life, all communications will have become digital …. Everyone can have private communications by using a small personal computer for scrambling. The individual may … use communication satellites …. Computers with wideband data links can provide graphical communications, allowing widely separated Air Force elements to discuss plans, documents, maps, etc.[51]

The other, more direct, influences on the Smalltalk design came through systems that Alan Kay encountered during his graduate studies. According to his reflections, he was handed the Sketchpad thesis to read by his advisor Dave Evans when he joined. Soon after, he was given ‘a pile of tapes and listings’ to get to work. The code was supposedly Algol, but it turned out to be ‘doctored to make a language called Simula’.[52] The two encounters certainly influenced Kay’s later work, but his writing on the FLEX system suggests that the influence was gradual.

Kay started his studies in 1966 and submitted a master’s thesis titled ‘FLEX: A Flexible Extendable Language’[53] in 1968 before completing his PhD thesis titled ‘The Reactive Engine’[54] a year later. His work focused on the programming side of a small computer developed in the group that Kay and his collaborators called FLEX. The aim of the FLEX project was to build a small computer that an individual could own and use. It was inspired by LINC, which I briefly mentioned in Chapter 3 and is sometimes treated as the first ‘personal computer’.[55] In his master’s thesis, Kay focuses mostly on the programming language, which is ‘intended to be a simple yet powerful and comprehensive notation to express computer-oriented algorithms’. The FLEX system ‘follows the traditions set by Algol 60 and several generations of EULER’, the latter being an extension of Algol 60 with support for data types developed by Niklaus Wirth. In many ways, the thesis follows the style of the mathematical culture of programming. The language description is written in the style of the language definition of Algol that I examined in Chapter 2. It focuses on the extensible syntax of the language, such as the ability to define new symbolic operators. Notably, the master’s thesis does not yet talk about graphical user interface and only reports ‘several partially successful attempts’ to ‘combine [FLEX] compilers with a number of the operating text editors at the University of Utah’. The brief list of references also does not yet include Sketchpad or SIMULA.

In 1968, Kay participated in a graduate student meeting of the ARPA community, saw a demo of the tablet-controlled graphical flowchart programming system GRAIL and visited the Logo group at MIT. He ‘realized that the FLEX interface was all wrong’[56] and tried to find ways of fitting some of what he saw into the tiny FLEX machine. His PhD thesis reflects the new ideas. It pays much more attention to the interaction with the user and positions the work using Kay’s typical broad-stroke references. In this case, the references are to the ‘basic framework of the Indo-European languages’ and the philosophical tradition where ‘the world has objects which endure over time’.[57] Finally, the thesis also sketches a graphical interface for interacting with the program (Figure 6.3) through a tablet. In the ‘Implementations to date’ section, the thesis however, does acknowledge failures (‘due to the inadequacies of Algol as a

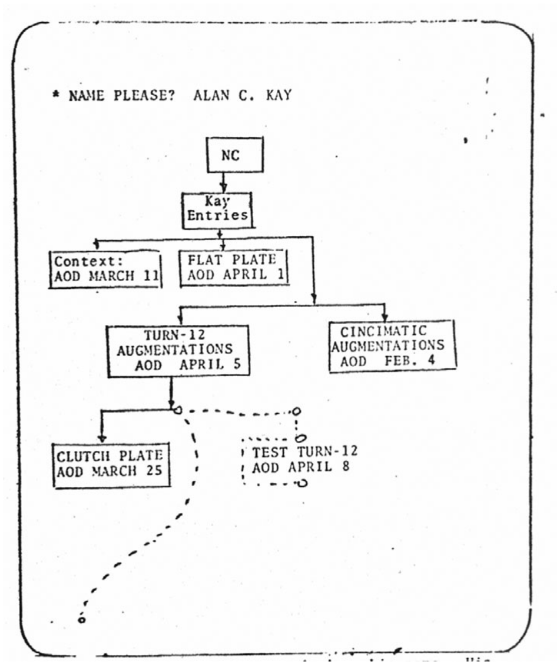

* NAME PLEASE? ALAN C. KAY
NC
Kay Entries
Context: AOD MARCH 11
FLAT PLATE AOD APRIL 1
TURN-12 AUGMENTATIONS AOD APRIL 5
CINCIMATIC AUGMENTATIONS AOD FEB. 4
CLUTCH PLATE AOD MARCH 25
TEST TURN-12 AOD APRIL 8

Figure 6.3 'A Sample Program' from the FLEX thesis.[58] The diagram shows the 'previous context' that the user sees after giving their name.
(Source: Courtesy of Alan Kay. Reproduced with permission.)

real-time language') in integrating the FLEX compilers with interactive editors and a graphics system developed at Utah.

The description of the FLEX system does not talk about objects or object-oriented programming, but it exhibits some of the characteristics that will later define the paradigm. The key abstraction is a *property list*, which is associated with a name through a binding. The notion has likely been inspired by property lists in LISP 1.5, where each atomic symbol has an associated property list that stores basic information related to the symbol, such as its name and its raw value or machine language code. The point of departure in FLEX is that property symbols are not primarily internal. They are the mechanism through which the programmer would use the system. The properties stored in the property list can be used in a number of ways, including the familiar dot-notation, although in a prefix rather than postfix form. In FLEX, `age.Bob` refers to the property 'age' of 'Bob' as in the following example, which represents an interactive session (ending with the `type` command, which instructs the system to print the result of an expression):[59]

```
Bob ← ':age, :weight, :height.[if age = 40 then sleep]';
Bob(40, 175, 70);

type age.Bob;
```

Names with associated property lists, which are the basis of FLEX, are in many ways similar to later objects. The properties associated with a name may include values, but also procedures that may access the associated properties directly without quantification (as when accessing the `age` property in the above example).

Looking at the references in Kay's PhD thesis suggests that his doctoral work was influenced by the mathematical culture of programming at least as much, if not more than, by the humanistic culture. He references Church's lambda calculus and also the work of Peter Landin (discussed in the context of types in Chapter 5) on lambda calculus models of Algol.[60] By contrast, references to SIMULA and Sketchpad are sparse. SIMULA is mentioned in the context of record handling (and only the 1966 paper describing SIMULA I) and the only citation of Sutherland's work is a reference to a specific clipping mechanism.

6.4 A New Medium for Communication

After completing his doctoral studies, Alan Kay joined the Xerox PARC research centre. As I already wrote in Chapter 3, the centre brought together a non-conformist group with strong backgrounds in the hacker, engineering and humanistic cultures, many of whom had earlier links with the ARPA community or Douglas Engelbart's group. The work on Smalltalk and what would later become object-oriented programming was happening amidst the backdrop of interesting hardware developments as well as political struggles against the Xerox management.[61] Kay himself devoted as much attention, if not more, to thinking about small computers for kids and their use in education, as to the Smalltalk language. In this chapter, I focus on the language, although its development is inseparable from this broader technical and humanistic context.

In 1971, Kay created the first version of a new programming language that was, by then, called Smalltalk. The account from Kay's later reflections suggests that his starting point was an interest in building a flexible programming language 'that could handle a variety of styles of programming'[62] in a completely uniform way without built-in language constructs that are treated in a special way. One such construct is the conditional. In Lisp, a conditional is not an ordinary call and needs to be evaluated in a special way. For example, consider an expression `if` (> `a` 10) (`setq` `a` 0) that sets the variable `a` to 0 if it is greater than 10. This requires that the second element of the list (`setq` `a` 0) is evaluated only when the first element (> `a` 10) is true, rather than being evaluated automatically along with all other arguments as is done for ordinary function calls.[63]

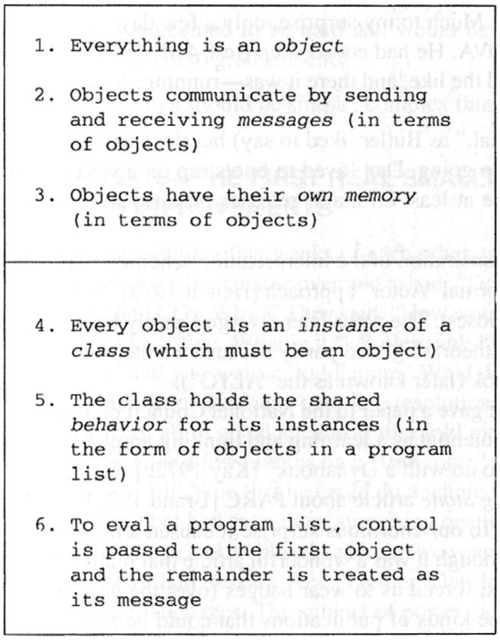
Supporting a variety of programming styles in FLEX required turning different parts of the program into entities that could be passed around and evaluated in different contexts, leading to the idea of objects and messages. Despite making yet another step in the direction of object-oriented programming, the motivating problem would fit well into the ongoing, more academic and mathematical, research on extensible programming languages that I will return to in the next section. At this point, Kay was arguably still solving an 'old problem', but using what was gradually becoming a 'new thing'.

The key ideas of what later became known as object-oriented programming were crystallised in Smalltalk-72. Because of the political fallout from the 1972 Spacewar article by Stewart Brand in *Rolling Stone* magazine, the group was prohibited from publishing until 1975[64] and so a historical account of Smalltalk-72 has to rely on the handbook published in 1976[65] and a handful of internal documents from the era.

Unlike in the earlier publications about FLEX that emphasise many other aspects of the language design, the Smalltalk-72 handbook has a clear focus on objects. This is clear in chapter III, which follows a number of example-oriented walkthroughs and provides 'a direct discussion of the basic Smalltalk concepts: classes, instances, and message sending and receiving'. In particular, the subsequent paragraph very closely captures the design principles described in Kay's retrospective paper (Figure 6.4):

> Every entity in Smalltalk's world is called an object. Objects can remember things and communicate with each other by sending and receiving messages. Every object belongs to a class (which is also an object). The class handles all communication (receiving a message and possibly producing a reply) for every object which belongs to it.[66]

The evolution of the FLEX system (and to some extent Smalltalk-71) into Smalltalk-72 is in many ways similar to the evolution of SIMULA I into SIMULA 67. It is gradual and progresses via reasonable responses to specific design challenges, but it


1. Everything is an *object*
2. Objects communicate by sending and receiving *messages* (in terms of objects)
3. Objects have their *own memory* (in terms of objects)

4. Every object is an *instance* of a *class* (which must be an object)
5. The class holds the shared *behavior* for its instances (in the form of objects in a program list)
6. To eval a program list, control is passed to the first object and the remainder is treated as its message

Figure 6.4 Six Smalltalk-72 principles, as reconstructed in Kay's account of the history of Smalltalk:[67] 'The first three principles are what objects "are about" – how they are seen and used from "the outside". These did not require any modification over the years. The last three – objects from the inside – were tinkered with in every version of Smalltalk (and in subsequent OOP designs).'
(Source: Used with permission of ACM Special Interest Group on Programming Languages,[68] permission conveyed through Copyright Clearance Center, Inc.)

significantly changes the narrative to one that is much closer to the modern notion of object-oriented programming.[69] It is likely that many of the concepts emerged independently in the two environments, but the terminology of Smalltalk-72 may have been influenced by SIMULA 67. An acknowledgements document, written in March 1973, 'shortly after Smalltalk-72 started working',[70] cites many influences on the language design including Logo and Lisp, but also 'SIMULAs ('65 and '67)'. The latter is recognised for its 'epistemology that allowed a class to have any number of parallel instantiations'. The FLEX language then 'took the SIMULA ideas (discarding most of the AGOLishness [sic]), moved "type" from a variable onto the objects', providing the basis of the model fully embraced in Smalltalk-72.

The fact that a class is also an object in Smalltalk-72 is one of the cornerstones of the Smalltalk language and the treatment of classes provides two nice examples of cultural roots of the language. The first example is the attempt to unify as many concepts in the language as possible. We can see this unification in the treatment of classes, but also in Kay's attempt to unify multiple programming styles in a single language in the FLEX system. Those designs were less motivated by engineering concerns and more by aesthetic aims that are typical for the humanistic (and sometimes mathematical) culture. As Key recalls, 'much of the pondering' during Smalltalk-71 design 'had to do with trying to understand what "beautiful" might mean ...'.[71] The second example is the treatment of (to use an anachronistic term) class inheritance. Smalltalk-72 did not adopt the prefixing mechanism of SIMULA 67. Kay recalls that he 'just decided to leave inheritance out as a feature in Smalltalk-72, knowing that we could simulate it back using Smalltalk's Lisp-like flexibility'.[72] In line with the hacker tradition of Lisp, the Smalltalk design gives the programmer enough control to do things in the way they want. Whereas the focus of SIMULA was on security and use of types to prevent potential errors, Smalltalk exposes the internals of the system and lets its users tinker with them.

Another glimpse of the affinity to the hacker culture of programming can be found in the internal Smalltalk memos. The document 'Summary of Smalltalk Message Forms and Intentions',[73] which was prepared to accompany an internal class on Smalltalk, provides a high-level overview of many of the messages that the students may need. Despite being a documentation for a Smalltalk tutorial, the document is focused on source code. It follows the formatting of Smalltalk class definitions, using the actual (non-standard) symbols used in Smalltalk-72 (Figure 6.5). Moreover, in places where the documentation is not complete, the memo makes reference to concrete people from the team (following the individualism of the hacker culture) and the source code of the core Smalltalk library as, for example, in 'Diana and SYSDEFS can tell you the rest'.

The opening comment of the referenced SYSDEFS file[74] is also written in the style of the HAKMEM memo, which I used to illustrate the style of writing of the hacker culture in Chapter 1. An annotated version of the file opens with:

```
HEREWITH A SOMEWHAT WHIMSICAL ANNOTATED VERSION OF
SYSDEFS. ANNOTATIONS ARE IN ITALICS. WHILE IT IS HOPED
THAT THIS WILL
```

to turtle z: x y dir pen ink

isnew ⇒ (...)	creates a new turtle
◁ goto ⇒ (▤▤...)	from the current position, traces to x y position: ▤▤. Returns distance.
◁ penup ⇒ (...)	picks the 'pen' up. No ink will flow
◁ pendown ⇒ (...)	puts the 'pen' down. No ink will flow
◁ home ⇒ (...)	take the turtle 'home'; currently x y position 256 256.
◁ ink ⇒ (◁ white ⇒ (...) ◁ black ⇒ (...)	On the CSL graphics color display, ink can be any of 256 colors.

Figure 6.5 A part of the documentation for the 'turtle' class.
(Source: Kay (1974); custom symbols replaced with nearest Unicode alternative)

```
PROVIDE SOME ELUCIDATION OF THE CODE ESCAPES, OBSCURITIES
WILL
NO DOUBT PERSIST. THE ANNOTATIONS ARE INTENDED TO BE BUT
DIMLY
LIGHTED MARKERS ON THE ROAD TO TRUE ILLUMINATION.
```

Smalltalk-72 was developed for the experimental Alto computer built at Xerox PARC at the same time and the combination was used internally at Xerox for a wide range of projects. The Smalltalk team now included Dan Ingalls, who did much of the engineering to make Smalltalk actually work and Adele Goldberg who joined Xerox PARC in 1973 because of her interest in computer education. She built a range of class materials based on Smalltalk, despite later recalling that she initially found Smalltalk-72 an 'extremely odd language' and thought that 'this wasn't going to be teachable to anybody'.[75] The language was also used by various other Xerox PARC members for projects ranging from the Logo turtle library, text editor and music synthesiser to the Pygmalion programming system discussed in Chapter 3.

As Dan Ingalls later noted, much of the team's 'gratification was tinged with a feeling that things were missing or not quite right'.[76] The lack of support for inheritance led to duplication of code, classes and activation records, which were not real objects, the system was not very efficient and would easily run out of space. The dissatisfaction together with interest in building something fresh eventually resulted in a new, cleaner, better engineered and more efficient implementation that became known as Smalltalk-74. One interesting, but invisible, technical innovation in Smalltalk-74 was the OOZE (Object Oriented Zoned Environment). The OOZE was a virtual memory system that made it possible to create more objects than would fit into the limited memory available on the Alto. The basic idea of storing objects that are not actively needed to disk sounds simple, but required ingenious implementation tricks.

For the history of object-oriented programming the OOZE is interesting for another reason. It offers evidence of the use of the term ‘object-oriented’ by the Smalltalk team. There is no reference to ‘object-oriented’ in the publications on SIMULA and Smalltalk is often credited with using the term for the first time, not just by Kay himself. For example, in a paper that tried to characterise object-oriented programming in 1982,[77] Tim Rentsch argued that ‘the explicit awareness of the idea – including the term “object oriented” – came from the Smalltalk effort’. It seems likely that the term was in use at Xerox PARC during the ‘publication blackout’ (1972–75) as there are quite a few publications appearing in the second half of the 1970s that use the term, without feeling the need to define it. That said, the definition of what is object-oriented may not be as straightforward as the popular history suggests and I will return to this later.

6.5 Let’s (Not) Burn Our Disk Packs

The two personal retrospectives on the history of Smalltalk, written by Alan Kay and Dan Ingalls and published in the Proceedings of the History of Programming Languages conference,[78] offer two perspectives on the developments after Smalltalk-74. They also illustrate how different cultures of programming joined forces to give birth to the Smalltalk programming language, but then led their proponents down separate paths.

Alan Kay’s thinking about Smalltalk is closely aligned with the interests of the humanistic culture. He cared about building a personal computer that could be used by children and saw programming as a basic way of interacting with the machine, much like Seymour Papert treated programming as ‘communicating with a living native speaker of mathematics’.[79] Dan Ingalls who led the implementation of Smalltalk invented many of the tricks that made it actually work and his writing on the history of Smalltalk often talks about interesting technical challenges that motivated certain developments. Adele Goldberg joined the team with interest in education, but later became manager of the group in 1979, leading the public release of Smalltalk, and she went on to establish a company, ParcPlace Systems, to develop Smalltalk into a commercial product.

Both Kay and Ingalls express some unease about the state of Smalltalk after Smalltalk-74 became widely used. For Kay, the unease is about losing the original focus. By the end of 1975, he felt ‘that “the Dynabook for children” idea was slowly dimming out – or perhaps starting to be overwhelmed by professional needs’. He organised a three-day offsite in January 1976 where he tried to convince the group to ‘burn our disk packs’ and start afresh. Kay wanted a new machine with a new system. He ‘did not see how [object-oriented programming] by itself was going to solve our end-user problems’ and he started designing a new language and system called NoteTaker. For Ingalls, the unease about Smalltalk-74 was mainly technical. The ‘brewing forces and frustrations came together’, during a weekend at the beach in August 1976, ‘to suggest an entirely new language and architecture that promised true commercial speed while preserving [the] object-oriented style'.[80] The work resulted in a new language implementation called Smalltalk-76, which became the basis of all 'modern Smalltalks'.

The design of Smalltalk-76 is technically interesting in a number of ways, but most are outside the scope of this book. One remarkable aspect is that it was written in Smalltalk-74, which is one of the cases where Smalltalk 'has been so good at getting rid of previous versions of itself', a quality that Alan Kay praised in the keynote speech discussed in the opening of this chapter. (This required writing a parser and a compiler first in Smalltalk-74, testing it and then transliterating it into the new syntax and structure of Smalltalk-76.) It is also worth looking more closely at a paper about Smalltalk-76 that Dan Ingalls wrote and presented at the ACM POPL symposium in 1978.[81] Although a few publications about Smalltalk started to appear at the time, those were mostly focused on the humanistic visions behind Smalltalk (such as the 'Personal Dynamic Media'[82] paper). Ingalls' paper is, by contrast, a detailed technical account of the language and appeared in a programming language research venue.

The paper keeps some of the humanistic motivations behind Smalltalk. The introduction opens with the purpose of Smalltalk, which is 'to support children of all ages in the world of information'. It argues that the 'SIMULA notion of class and instance is an outstanding metaphor for information structure' and adds the notion of message sending as a similarly powerful metaphor for processing. Following the humanistic introduction, the paper focuses on engineering notions. It describes Smalltalk as 'object oriented rather than function oriented' and talks about modularity, which is that 'no part of a complex system should depend on the internal details of any other part'. The 'class is the natural unit of modularity, as it describes all the external messages understood by its instances'. Smalltalk-76 also (finally) implements inheritance and adopts the terms 'subclass' and 'inheritance'.

The notions of object-oriented programming, classes and inheritance that are familiar to programmers today can easily be found in Ingalls' paper. However, it is perhaps telling that those took final shape only after a redesign and rewrite of the language that was motivated by a mix of hacker and engineering motivations and has played down some of the humanistic visions. Alan Kay himself was, for example, 'not completely thrilled with'[83] inheritance in Smalltalk-76 and as his 1997 keynote makes clear, valued the Smalltalk ability to evolve into new systems more than any specific version of the language design.

6.6 What Is an Object-Oriented Programming Language?

The object-oriented programming paradigm, as understood at the end of the 1970s, was the result of yet another meeting of multiple cultures of programming. It brought together the mathematical culture, from which the work on SIMULA originated, with the humanistic motivations of Alan Kay and a mix of hacker and engineering skills and interests of the other members of the Smalltalk group. The influences were gradual in that the ideas that emerged in one context were later adopted and adapted in another context. As we will see, object-oriented programming also kept its multi-faceted nature after the 1970s. Languages influenced by SIMULA and the mathematical culture emphasise safety and reasoning about programs, whereas languages arising from the hacker and humanistic cultures of Smalltalk emphasise the flexibility of the paradigm. The engineering culture, positioned somewhere in the middle, will interpret object-oriented programming as a pragmatic tool for building software and will look for ways of using it effectively. Before I move to a more recent history of object-oriented programming, it is worth revisiting the origins of some of the terms and ideas of the paradigm and look at parallel ideas that were overshadowed by the later rise of object-oriented programming.

The terms 'object' and 'class' were first used in SIMULA 67 and were motivated by everyday uses of the terms in English. Nygaard and Dahl talked about real-world 'objects' classified into 'classes'. They used the term 'class' in passing even earlier when talking about activity declarations in SIMULA I. The term 'inheritance' has a more interesting history. It was not used in SIMULA, which referred to the mechanism as 'prefixing,' but has already adopted the term 'subclass' for the new class definition created through prefixing. In the context of Smalltalk, the term 'inheritance' is certainly used in Smalltalk-76,[84] but when exactly how it entered the group's vocabulary is hard to trace.

One possible source is the Knowledge Representation Language (KRL) developed by Daniel G. Bobrow (in another research group in Xerox PARC) and Terry Winograd (at Stanford, but consulting for Xerox PARC). The KRL originated in artificial intelligence (AI) research as a language for describing knowledge that could be used by AI tools for reasoning about the represented concepts, but it was surprisingly closely related to the emerging object-oriented programming paradigm. It structured the knowledge in terms of 'entities with associated descriptions and procedures'[85] and cited SIMULA and early Smalltalk as influences for the idea of procedural attachment. Interestingly for my analysis of terminology, KRL provides 'a notion of subclass which allows objects to inherit procedural as well as declarative properties'[86] and KRL also refers to earlier work on 'property inheritance' in the field of AI research. In his retrospective, Kay acknowledges interactions with the KRL designers too, and so it is likely that this particular term came to object-oriented programming through yet another indirect influence.

The last term I want to revisit is 'object-oriented programming'. Not because the history of the terminology itself is particularly noteworthy, but because it will lead us to related ideas developing in parallel with those in Smalltalk. Kay himself suggested[87] that he use the term 'object-oriented programming' to describe his work as early as 1967 during his time at Utah, but the term was not used in publications about the FLEX machine or Smalltalk until 1978. The term was certainly in use by the Smalltalk team earlier, as evidenced by the naming of the OOZE (Object Oriented Zoned Environment). However, what exactly 'object-oriented' stands for was by no means settled by the end of the 1970s. Another contender for the term originated in programming languages built around the idea of abstract data types.

The idea of abstract data types, which I discussed in Chapter 5, is to define entities from which programs are composed by the operations that they support. The concept was introduced by Barbara Liskov and Stephen Zilles alongside a programming language Clu that implemented the idea. Their definition from 1974[88] uses a language that follows much of the object-oriented terminology. According to Liskov and Zilles, ‘an abstract data type defines a class of abstract objects which is completely characterised by the operations available on those objects’. They did not call this style of programming object-oriented in the original paper, but a subsequent report co-authored by Liskov from 1976[89] uses the term ‘object-oriented languages’ to refer to SIMULA 67, Clu and Alphard (another language implementing the notion of an abstract data type). A 1979 PhD thesis supervised by Liskov on ‘Machine Architecture to Support Object-Oriented Language’ likewise uses the term object-oriented languages for SIMULA and Clu, adding Lisp but still omitting perhaps not yet universally known Smalltalk.

In an email exchange on the topic of object-oriented programming,[90] Kay pointed out that SIMULA has catalysed two main paths. The first being the work on abstract data types. According to Kay, this line of work is a ‘better old thing’ in that it preserves the programming style with ‘bindings and assignments’.[91] The second path is exemplified by Smalltalk, which replaces data and operations with messaging. In an oft-cited (but historically inaccurate) quote, Kay expressed his regrets that he ‘coined the term “objects” for this topic because it gets many people to focus on the lesser idea. The big idea is “messaging”.’[92] He has also frequently criticised modern object-oriented programming languages for following the former rather than the latter path. It is thus a curious historical accident that the folk history of object-oriented programming today refers so strongly to Smalltalk. The term object-oriented programming has been equally used to talk about programming languages centred around the notion of abstract data type, which are arguably closer predecessors to many modern object-oriented programming languages.

The difference between the two ways of thinking that trace back to SIMULA has not been widely acknowledged in the computer science community and it remains an issue worth discussing. A 2009 essay by William R. Cook[93] opens by pointing out that the typical answer to the question ‘What is the relationship between objects and abstract data types?’ is ‘objects are a kind of abstract data type’. According to Cook’s detailed analysis ‘an abstract data type is a structure that implements a new type by hiding the representation of the type and supplying operations to manipulate its values’. Whereas ‘an object is a value exporting a procedural interface to data or behaviour’. The difference is somewhat subtle, but interesting. Abstract data types implement some precisely specified mathematical interface. This also makes it possible to prove properties of programs that use abstract data types through the interface. Objects are ‘autognostic’ meaning that they know only themselves. Interaction with any other object is a matter of sending a message, but it is entirely up to the other object to decide how to handle such message.

A close look at Alan Kay’s early work on the FLEX machine and early Smalltalk reveals another interesting aspect of the history of programming. Although Kay himself always presents those as early steps towards his vision of Dynabook and object-oriented programming, much of the work on those systems could equally be seen as contribution to the work on ‘extensible programming languages’. This now largely forgotten research area emerged at the end of the 1960s and aimed to develop programming languages where ‘one can tailor a language facility to very closely mirror the unit data, unit operations, and ways of writing that are “natural” to some problem area’.[94] The research theme covered topics such as definitions of new data types, custom operators, new ways of composing primitives and customisable syntax. What may seem like a hotchpotch of programming language features to a present-day computer scientist, combining work on basic language semantics, syntax and libraries, formed a relatively coherent research field for a number of years.

Early work on FLEX and early Smalltalk fits very well with the theme. The very name, FLEX, is an acronym for ‘FLexible EXtendable’. The extensibility applies both to the basic vocabulary of the language and to its syntax. As noted in the FLEX description, ‘the system does not distinguish between its own entries and the user’s. Therefore, a verb (standing for a subroutine, perhaps) created by the user has as much power, scope and usability as the system-defined verbs.’[95] The syntax can be extended by defining new custom operators, but also by modifying the system itself ‘via the compiler-compiler contained in the language’.[96] When designing Smalltalk-71, Kay found it ‘annoying that the surface beauty of Lisp was marred by some of its key parts having to be introduced as “special forms” rather than as its supposed universal building block of functions’.[97] He was dissatisfied with the fact that constructs like conditionals could not be fully defined by the user. He later recalls finding an answer in a PhD thesis ‘Control structures for programming languages’ written by David Fisher at Carnegie-Mellon University in 1970. In his thesis, Fisher cites numerous other works on extensible languages and notes that applications of the work to extensible languages is ‘one obvious area for further research’.[98]

Kay attended the Extensible Languages Symposium in 1969, but later recalled that it was, in his view, ‘a religious war of unimplemented poorly thought out ideas’.[99] However, he did find inspiration in one of the presented (actually implemented) systems where ‘every procedure in the system defined its own syntax’. Kay’s reinterpretation of the idea in FLEX (also present in Smalltalk-72 but abandoned in later Smalltalks) was to see ‘each object as a syntax directed interpreter of messages sent to it’.[100] In retrospect, Kay was probably fortunate not to attach his work to the area of ‘extensible languages’ as the area largely lost traction by the mid 1970s. In a somewhat ironic closing of the research agenda, some felt that it has already achieved its goals, while others felt that its goals were ‘a bit over-ambitious’.[101]

The birth of object-oriented programming is entangled in curious ways with other work, happening at the time, for which some have used the term ‘very-high-level’ programming languages. The research relates to the framing that I mentioned when discussing SIMULA. After machine languages, assembly languages and high-level programming languages like FORTRAN and Algol, programmers started exploring ways of bringing the programming languages even closer to the problem domain. SIMULA was (initially) specifically designed to model simulations, but later found use in general purpose modelling of the real world. In extensible languages, the idea was to adapt the language to the problem domain. Similarly, abstract data types aimed to 'ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area'.[102] Out of these three, object-oriented programming is the best known survivor. The notion of abstract data type has became primarily a theoretical idea of the mathematical culture. Extensible languages have been mostly forgotten as a concept, even if many of their technical innovations continue to exist, some as commonplace (data types) and some as sophisticated (macros) programming language features.

6.7 Structured Programming of the 1980s

In the late 1970s and 1980s, object-oriented programming became a household name in the computing industry. There were new object-oriented languages, object-oriented design methodologies and Intel even introduced a processor, Intel iAPX 432, that provided hardware support for object-oriented programming. This rising popularity of the term in various contexts also inspired critical reflections on its meaning:

> What is object oriented programming? My guess is that object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favour of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.[103]

There was much truth in this prediction. I will not attempt to trace how exactly object-oriented programming gained popularity and widespread adoption. There were likely multiple contributing factors. The perceived industry crisis made developers eager to look for new approaches, object-oriented programming had some early success stories and objects were closely linked to the emerging graphical user interfaces. The technology magazine *Byte* contributed to the popularisation of Smalltalk and object-oriented ideas in 1981, when it dedicated an entire issue of the magazine to the system.[104]

An interesting aspect of the rise of object-oriented programming in the 1980s for my analysis is how the different cultures of programming influenced the subsequent developments. In a way, the developments mirror those of early programming languages that I followed in Chapter 2, where multiple cultures contributed to the idea of a programming language but then developed it in different directions. In the case of object-oriented programming, the origins of the idea also bring together multiple cultures of programming. And similarly, the later developments of object-oriented programming languages take the idea in multiple diverging directions.

The idea of cultures of programming resonates with a distinction that has been made by Peter Wegner during a panel at the ECOOP conference and that was elaborated by Steve Cook in a later conference report.[105] Wegner distinguishes between the Scandinavian and American school of object-orientation. In Scandinavia, 'research seems to be very concerned with putting object-oriented programming on a theoretical footing by making analogies with physics'. This was the case with SIMULA, but multiple languages that followed in the SIMULA tradition adopted the same view. By contrast, 'the American school of thought does not appeal to any fundamental theory … the primary criteria for thinking about object-oriented programming in the USA are pragmatic. What the American school wants to know is, does it work, and if it does, then how do we make it better?'

The so-called American school closely matches the engineering culture of programming, but finding a match for the Scandinavian school is harder. It exhibits a combination of mathematical striving for rigorous foundations with humanistic concerns about the real world. The contrasting views are not merely an interesting historical fact, but they keep shaping discussion about object-oriented programming today. Two of the object-oriented pioneers of the Scandinavian school, Ole Lehrmann Madsen and Birger Møller-Pedersen, wrote an essay in 2022 in response to the 'increasing criticism of object-oriented programming [in] recent years' arguing that there 'are indeed issues with object-orientation as practised by mainstream' and that 'the primary reason for these issues is that reuse is considered the main advantage of object-orientation at the expense of modelling'.[106]

To the dichotomy between Scandinavian and American schools that later became well known, Cook also added the British school. This disappeared from later discussions about object-oriented programming, but it perfectly matches with the mathematical culture as characterised in this book. For the British school, 'the ability to treat programs as mathematical entities is of paramount importance'. Cook points out that, because of its focus on side-effects and sharing, object-oriented programming 'has had something of a hard ride with British researchers'.[107] Perhaps because of the 'hard ride' with mathematically oriented researchers, there are only a few early contributions to object-oriented programming rooted solely in the mathematical culture of programming. The earliest attempts to use formal methods were motivated by the desire to explain how exactly certain practical aspects of object-orientation worked. In the mid 1980s, Luca Cardelli used models based on the lambda calculus to study data abstraction[108] and Cardelli and William R. Cook both used formal programming language semantics to model inheritance.[109] A couple of years later, the book 'A theory of objects'[110] provided a treatment of object-oriented languages that used the standard methods of the mathematical culture and followed the style of the ML paradigm, which I discussed in Chapter 5. However, the mathematical culture was also more directly shaping object-oriented programming in combination with other ways of thinking.

In Scandinavia, Kristen Nygaard kept working on object-oriented programming, collaborating with colleagues from Norway and Denmark, through a series of projects that eventually resulted in the language BETA. Their focus remained on modelling, but more in the sense of modelling real-world concepts than in the sense of numerical simulations. Consequently, BETA was seen both as a programming language and as a modelling language. The designers even had a rule that a language construct had to be useful both as a modelling tool and as a programming feature.[111] The modelling work of the Scandinavian school was, by no means, a dry mathematical activity. Nygaard himself worked with trade unions in Norway and was interested in co-designing systems directly with their users. This way of thinking fits with the humanistic culture of programming and it later resulted in the participatory design methodology that also influenced Christiane Floyd and her work on software engineering, which I talked about in Chapter 4. Nevertheless, many aspects of the BETA programming language design associate it closely with the mathematical culture. One example is the mathematical focus on simplicity and formal elegance. The central feature of the BETA design is a single abstraction mechanism, the pattern, which unifies records, classes, functions and methods. This comes at a usability cost in that the uniform syntax of BETA is 'often claimed [to be] awkward'.[112]

The Smalltalk language also continued to be influential. It did so in multiple ways. After some initial difficulties, it was reimplemented for a new generation of computers and was adopted commercially. I will return to this thread later in this chapter. However, its original ideas also inspired (and continue to inspire) new designs and developments. A programming language that illustrates this influence well is Self. Developed at Xerox PARC by David Ungar and Randall Smith in the mid 1980s, the language aimed to 'improve upon Smalltalk by increasing both expressive power and simplicity, while obtaining a more concrete feel'.[113] Last but not least, the original creators of Smalltalk recovered and open-sourced an implementation of Smalltalk-80, 'finally doing, in September of 1996, what [they] had failed to do in 1980'.[114] The new version, called Squeak, enabled a new wave of interest in Smalltalk in the 2000s, including both research around programming systems in the original humanistic and hacker traditions, as well as a more engineering-oriented development around Pharo, a fork of Squeak created in 2008.

6.8 The Power of Simplicity

Like BETA, the Self programming language also set out to reduce the number of different concepts needed in object-oriented programming. One aspect of this was eliminating variables. Object state was accessed by sending messages to 'self', a design feature that gave the language its name. Another aspect was the use of prototypes instead of classes. In SIMULA and Smalltalk, objects are created as instances of classes, introducing a distinction between a description of objects and objects themselves. Prototypes eliminate this distinction. An object in Self describes its own behaviour, but can refer to another object as a prototype for shared behaviour.

The idea of prototypes in Self relates to its other stated goal, which is a 'more concrete feel'. The language was born when graphical user interfaces were becoming the norm. Smalltalk adopted graphical interface for programming in the form of a class browser, but Self imagined the next step. The designers 'wanted the programming environment's graphical display of an object to be the object for the programmer'. That is, the entities on the screen would be the concrete objects themselves. The idea was implemented in the graphical user interface later called UI1 in 1988/89. The interface represents objects as nested boxes (reminiscent of the Boxer project discussed in Chapter 3) and the prototype-based design of the language made it possible to design an interface where there was ‘nothing on the screen (except for pop-up menus) that was not an object’.[115] Because there is no distinction between objects and classes, the prototype of an object is just another object on the screen. Although UI1 achieved the goal of ‘concrete’ representation of objects, it did not help users build graphical interfaces with elements such as buttons or shapes. This was addressed in a redesigned version of the interface, UI2. Here, the thing displayed on the screen is not a box representing the internals of the object, but a graphical representation of the object called ‘morph’, a term derived from a Greek word referring to ‘physical form’.[116] Morphs could take the form of buttons, shapes and other visual entities that could form a graphical application. As the authors put it later ‘every visual element in Self, from the largest window to the smallest triangle is a directly manipulable object’.[117] The visual elements are Self objects and some can be directly modified through interacting with them. To inspect and modify the objects, the user also has the option to display an ‘outliner’ (accessible through a pop-up menu), which creates a representation of the object similar to that of UI1.

The fact that Self focused on concrete graphical representation of objects and, at the same time, adopted a variant of object-oriented programming based on prototypes is not a coincidence. The same combination existed in other systems that influenced Self. The direct predecessor was ThingLab, developed at Xerox PARC by Alan Borning. ThingLab was a graphical ‘simulation laboratory’ where users could create graphical objects on screen and specify their behaviour using constraints. ThingLab was itself inspired by the even earlier Sketchpad, which I talked about in Chapter 3. ThingLab used the idea of prototypes in its internal representation of objects, but Borning also envisioned using the mechanism in an object-oriented programming language in 1986, at the time when the Self design was starting, likely directly influencing its design.[118]

The innovative technical characteristics of Self, such as the prototype-based object-oriented model and concrete user interface, can easily overshadow the humanistic vision of the system. This was, perhaps, not emphasised as strongly as in the case of Smalltalk, but a later article by the Self designers makes the vision behind the system explicit:

> [Past papers about Self] have not completely articulated an important part of the work: our shared vision of what programming should be. This vision focuses on the overall experience of working with a programming system, and is perhaps as much a feeling thing as it is an intellectual thing.[119]

The authors explained that ‘the aspects of [programmer] experience which are beyond the domain of pure logic’ have traditionally been separated from the logic of the programming language. They are the ‘domain of the programming environment’. However, just like in Smalltalk, the programmer experience in Self was at least as important as the logic of the language. According to the vision of the authors, ‘the Self programmer lives and acts in a consistent and malleable world’. This later reflection makes it clear that Self follows Smalltalk in its allegiance to the humanistic culture of programming. The interesting technical innovations are the logical consequences of a new grand vision (enabled by graphical displays) about what programming should be.

The malleability of the world implies the ability of the user to adapt the environment to their needs and is, to some degree, in contrast with the design requirements typical for other cultures. The malleability does not make it possible to build safeguards required by the engineering culture and it makes the program open to modification, complicating mathematical reasoning about it.[120]

Self has suffered the same curse as many other systems rooted in the humanistic culture. It became influential through its technical contributions, but its humanistic vision was lost along the way. Self is known for its innovative graphical interface, sophisticated implementation techniques and prototype-based object-oriented programming model, but systems that leverage those do not necessarily subscribe to Self's vision of programming. JavaScript would be the prime example. It adopted both the prototype-based object-oriented programming model and also virtual machine implementation techniques that were pioneered in Self. Yet, its links to the humanistic culture of programming are only minimal. The more immediate path to a broader adoption of object-oriented programming led instead through work rooted in the engineering culture.

6.9 Making Programming Enjoyable for the Serious Programmer

Multiple programming languages that emerged in the late 1970s and the 1980s in the USA were developed with the aim of tackling the rising complexity and cost of software development. They paralleled the developments in the field of software engineering, which I talked about in Chapter 4 and that were the reaction to the same perceived industry crisis.[121] To a large extent, the object-oriented languages rooted in the engineering culture followed the approach later identified as the 'American school'. The Clu programming language developed by Barbara Liskov at MIT had academic origins and served as a vehicle for research on data abstraction and abstract data types. It brought together mathematical thinking of academic computer science and managerial thinking about the best development methodologies with the engineering focus on developing specific programming abstractions to support the practical task of programming. As discussed earlier, Clu and abstract data types can now be seen as an approach competing with object-oriented programming, but the authors themselves saw Clu as an object-oriented language. In a 1993 retrospective,[122] Liskov argues that 'Clu is an object-oriented language in the sense that it focuses attention on the properties of data objects and encourages programs to be developed by considering abstract properties of data'. However, she also acknowledges the difference from other object-oriented languages, especially the fact that Clu did not have support for inheritance (a feature she 'would probably try to include' today).

A similar approach to object-orientation can be found in the Ada language, which was developed at the US Department of Defense (DoD) in the late 1970s.[123] Its objectives were to reduce the costs of software development at the DoD and increase the reliability of the developed software. The effort incorporated both managerial and engineering techniques. On the one hand, its funders hoped to reduce costs by having a single language for the entire DoD, reducing training costs and enabling reuse. On the other hand, the language implemented numerous features that can help avoid errors, including strong type system, packages and exceptions. Although Ada added further support for object-orientation in a version standardised in 1995, the initial version has retrospectively been also called object-oriented. It was ‘designed to support and encourage object-oriented design’[124] through packages. Yet, at the same time, the designers were cautious about a principle that later became a key aspect of object-orientation:

> During the eighties the use of the ‘object-oriented’ buzzword changed; one now implies ‘object-oriented programming’ features such as ‘inheritance’, which are even dangerous in high-integrity systems.[125]

By the turn of the 1980s, the object-oriented ‘buzzword’ not only gained a more specific meaning, but it also became recognised as a de facto approach to developing software. The programming language that first tried to combine the original object-oriented ideas with the focus on rigorous software engineering was Eiffel. The language was created in 1986 by Bertrand Meyer who had a combination of academic and industry background. Mirroring the background of its designer, Eiffel also had dual origins. It was conceived while Meyer was at UC Santa Barbara, but was developed by a commercial company Interactive Software Engineering that Meyer later established.

The Eiffel language embodies ‘the belief that it is possible to treat [the task of software construction] as a serious engineering enterprise’. Meyer did not see his work as creating just a new programming language. The aims of the language ‘lead to a new culture of software development, focusing on the reuse of industrial-grade components’.[126] The allegiance to the engineering culture is reflected in the very name of the language and is a reference to Gustave Eiffel and the Eiffel tower. Unlike many software systems of the 1970s, the 1880s tower was built on time and within budget, using a small number of simple principles. The association between a programming language and a new culture is not unique to Eiffel. We saw the same association earlier, when talking about the Logo programming language in Chapter 3. Logo was likewise envisioned as a computer culture, but one aimed at learning and thinking about computers.

The focus on reusability and the development of reusable components became recognised as another tenet of the American school of object-orientation. It also shaped the design of Eiffel in a number of ways. In fact, Meyer claimed that the language ‘is nothing else than these principles taken to their full consequences’. Eiffel included many language features that computer scientists associate with good engineering practices and that emerged from the work on software engineering discussed in Chapter 4. This includes structured exception handling, a static type system that eliminates the `null` value and automatic memory management.

In one notable way, Eiffel embraced ideas that were previously the domain of the mathematical culture of programming. A key part of its *design by contract* methodology was a close integration between program code and its formal specification in the form of assertions. To illustrate this, consider the following example, which defines a deferred feature (abstract method in a more common terminology today) that puts a value at the *i*th location of a collection:[127]

```
put_i_th(v : like first; i : INTEGER) is
  -- Put item v at i-th position.
  require
    index_large_enough : i >= 1;
    index_small_enough : i <= count;
  deferred
  ensure
    not empty
end
```

The key parts of the code sample are the `require` and `ensure` clauses. The former specifies the pre-conditions of the operation. It can only be performed if the argument `i` is between 1 and the collection size represented by `count`. The latter specifies a post-condition, which is that the collection is not empty after the value is set. The idea of pre-conditions and post-conditions appeared in the mathematical culture in 1969 in the form of Hoare triples, written as $\{P\}c\{Q\}$. As we saw in Chapter 2, they could be used to prove properties of programs. With Eiffel, the idea takes a new form. Pre-conditions and post-conditions now become a part of the program, are checked during program execution, and their role is to help programmers create ‘industrial-grade components’ that have a precise specification and behave accordingly. With Eiffel, the pre-conditions and post-conditions thus shift from a mere mathematical tool into an engineering entity, much like proof assistants turned proofs about programs from an external mathematical entity into technical artifacts.

Whereas Eiffel took the engineering approach to object-oriented programming and enriched it with ideas from the mathematical culture, the most influential object-oriented programming language of the 1980s took the engineering approach and added it to a language born from the hacker culture. The language I’m referring to is C++.

The development of C++ very closely reflects the personal experience of its author, Bjarne Stroustrup. During his PhD in Cambridge, Stroustrup used the SIMULA programming language and was ‘impressed by the way the concepts of the language helped [him] think about the problems in [his] application’. This was due to the object-oriented concept of a class, which allowed him ‘to map [his] application concepts into the language constructs in a direct way’, making his code more readable than in any other language.[128] Unfortunately, the SIMULA implementation was inefficient and Stroustrup eventually reimplemented his system in BCPL, a predecessor of the C language. Stroustrup later attributed the inefficiency of SIMULA to a number of its features, including runtime type checking and garbage collection. It is perhaps not a surprise that when Stroustrup left Cambridge and decided ‘never again to attack a problem with [unsuitable] tools’,[129] he opted to build ‘C with Classes’ as the tool to use for his next major project. The new language was a ‘medium success’ and it served as the first step towards a more ambitious project, which was a ‘cleaned-up and extended successor to C with Classes’.

As documented by Stroustrup later, ‘C++ was designed to provide Simula’s facilities for program organisation together with C’s efficiency and flexibility for systems programming’.[130] Stroustrup saw object-orientation as a useful tool for organising large programs, but he was not interested in building a language where everything was an object or where computation was done solely by sending messages. He considered ‘computation … a problem solved by C’ and had no intention to change the basic model (later remarking that ‘Had I wanted an imitation Smalltalk, I would have built a much better imitation. Smalltalk is the best Smalltalk around’). Today, the C++ language is most directly associated with the engineering culture and is seen as a language suitable for systems programming and building large, reliable and efficient systems. In 2018, the US National Academy of Engineering awarded Bjarne Stroustrup the Draper Prize, given for ‘the advancement of engineering’, joining the ranks of engineers who developed the integrated circuits, turbojet engines or rechargeable batteries. (The list of recipients also includes Alan Kay, although not for his work on Smalltalk, but as part of a team that created Xerox Alto, the ‘first practical networked personal computer’.)

In contrast to languages like Eiffel, there is remarkably little direct acknowledgement of engineering concerns or visions in the works that document the history and design of C++. The initial language design certainly shows some preference for reliable engineering, including the use of static typing for the object-oriented programming model and ‘systematically eliminat[ing] the need to use [low-level] features except where they are essential’. C++ later evolved according to pragmatic engineering principles based on the needs and constraints of its real-world users. However, the early design shows many more traits of what I associate with the hacker culture of programming. This includes both technical and social aspects. C++ emphasises the efficiency of C, access to low-level operations and ability to deliberately circumvent the type system through unchecked type conversions. The early development process was also closer to what we might find in the hacker culture. In 1986, Stroustrup could still write that ‘there never was a C++ paper design; …. There never was a “C++ project” either, or a “C++ design committee”.’[131] Ironically, soon after this was written, the process to standardise the language design started and, today, C++ is documented in an ISO C++ standard, whose evolution is governed by the ISO/IEC JTC1/SC22/WG21 committee and its 23 or so sub-groups.

The later thinking of Bjarne Stroustrup also aligns the C++ language more closely with the engineering culture of programming. One example can be glimpsed from a talk that he gave in 1986, when he was invited to present at the conference of the Association of Simula Users in Stockholm. The talk, later published as a paper ‘What is Object-Oriented Programming?’[132] is a reflection on the question that I attempted to examine earlier. Stroustrup criticised the tendency to call almost any language object-oriented. (‘Could there somewhere be proponents of object-oriented programming in Fortran and Cobol? I think there must be.’) and argues that the ‘basic support a programmer needs to write object-oriented programs consists of a class mechanism with inheritance’. The feature that the designers of Ada saw as optional and dangerous is now a central feature required for object-orientation. The conclusions of the paper relate object-orientation to data abstraction, an issue that Alan Kay would likely view as worrying about a ‘better old thing’:

> Object-oriented programming is programming using inheritance. Data abstraction is programming using user defined types. With few exceptions, object-oriented programming can and ought to be a superset of data abstraction.

The history of C++ suggests that the allegiance to a particular culture of programming may not necessarily be fully determined when a new technical development starts. In its early days, C++ was perhaps closer to the hacker culture, but its later evolution attached it firmly to the engineering culture. This development fits with the principle identified by Bruno Latour. To paraphrase: ‘The fate of programming languages is in the hands of its later users.’[133] The early technical developments leading to Smalltalk may also fit this scheme as the initial work on the FLEX machine could well be linked to the more mathematical and engineering research on extensible programming languages.

6.10 The Enterprise Age of Visual Smalltalk

In the computer science community, Smalltalk is perhaps best remembered for its pioneering days in the 1970s when it emerged and evolved at Xerox PARC, developing the ideas that would later be seen as the ‘pure’ object-oriented programming paradigm. But there is also a later era of ‘commercial Smalltalk’ in the 1980s and 1990s, which is less well known, but equally interesting and perhaps even equally influential. For this era to start, Smalltalk first had to be released from its ivory tower at Xerox PARC. This was not so easy.

First, Smalltalk was developed for a series of workstations created at Xerox PARC, starting with the Xerox Alto. At the start of the 1970s, those were impressive machines, showing that personal computing has the potential to become a reality. At the end of the 1970s, the Xerox PARC workstations were still technologically impressive, but they were also inaccessible and expensive when compared to the technologically basic, but low-cost and increasingly popular microcomputers such as Apple II. Smalltalk utilised the unique hardware capabilities of the Xerox workstations to run efficiently, because much of the virtual machine could be implemented directly using the microcode instructions of the processor.[134] It also used its own character set that included a ‘•’ (bullet) operator for indexing and ‘←’ (left arrow) for assignment. This made releasing Smalltalk for other computers practically tricky.

Second, Smalltalk was not very well publicly documented. After the end of the Xerox publication blackout in 1975, it was described in a couple of academic papers and Alan Kay gave a number of talks featuring Smalltalk. One notable talk was at the banquet of the second West Coast Computer Faire in San Jose, incidentally also the event where the Apple II microcomputer was presented a year earlier. But many who saw Kay’s futuristic talks about Smalltalk were confused and left wondering if it actually existed.[135]

Last, but not least, Smalltalk still existed in an industrial research lab that had to decide what it wanted to do with it. At the end of the 1970s, Adele Goldberg became the manager of the laboratory developing Smalltalk as Alan Kay gradually left Xerox PARC after his failed attempt to convince the team to start afresh. Goldberg had been an active member of the ACM and believed all the research done over the last decade or so should be published. With the help of her manager, Bert Sutherland, she convinced Xerox that 'nobody in the company was interested in Smalltalk',[136] meaning that they were free to publish all the research, as long as they did the necessary work. The strategy that Goldberg settled on was to offer access to Smalltalk to a limited number of selected companies (including Hewlett-Packard, DEC, Tektronix, Intel and later Apple) in exchange for their assistance with the publication. The companies were asked for feedback on a book on Smalltalk design and implementation that Goldberg's group would produce and for a contribution to an experience report documenting their attempts to re-implement a basic Smalltalk system.

The initial four companies were carefully selected. The Smalltalk virtual machine was designed to work well on the Xerox workstations and Xerox selected companies that, or so they thought, were capable of producing a similar workstation. However, none of the companies attempted to do that. Instead, they tried running Smalltalk on conventional machines available at the time and all got 'really dismal performance'.[137] The one company out of the four that eventually succeeded and became the epicentre for commercial Smalltalk activity was Tektronix. In the 1970s, the company was a major electronics manufacturer, known mainly as the producer of oscilloscopes. This may seem an unusual background for a computer producer, but Tektronix had a lot of experience with graphical displays and increasingly complex processing units, making it an excellent fit for a Smalltalk system. In 1981, Tektronix also shifted its focus and started developing a personal workstation internally called Magnolia.

Allen Wirfs-Brock and Paul McCullough, who created the initial inefficient Smalltalk virtual machine implementation at Tektronix, transferred to the company's industrial research lab, Tek Labs, and tried to get Smalltalk to run again. This time, they did not directly follow the reference implementation documented in the Smalltalk 'blue book'[138], which they helped to edit. In many ways, the reference implementation followed the idealistic elegant visions behind Smalltalk, such as the principle that 'everything is an object'. So, for example, when sending a message (calling a method) the implementation created a context for the call, which was also an object. However, this was almost never accessed as an ordinary object from the program. The idealistic representation is useful for debugging tools or more advanced programming tricks, but not during ordinary program execution. Wirfs-Brock and McCullough took an engineering-oriented approach, favouring efficiency over uniformity and simplicity. In their new virtual machine, a 'tentative context' was not an object and it only became one if it was accessed as an object. Alongside other similar pragmatic implementation choices, this gave Tektronix a reasonably efficient Smalltalk implementation for the Magnolia.

The new Smalltalk implementation first became available to some 60 researchers at Tek Labs. One of its first users was Ward Cunningham, who also became the most prominent evangelist for Smalltalk at Tek Labs, helping others build Smalltalk projects ranging from mathematical software and expert systems to CAD design and even 3D graphics systems.[139] Many of the new Smalltalk projects were showcased at an internal Tek Labs science fair, convincing the company to turn the system into a product. Aligning with the hype at the time, Tektronix went on to release a series of workstations, starting with Tek 4404, marketed as ‘Artificial Intelligence System’. Some 10 years later, commercial Smalltalk users who ‘knew that the workstation division [of Tektronix] was really making Smalltalk machines’ referred to those systems as ‘the first integrated Smalltalk environments, disguised by Tektronix as AI workstations’.[140]

The activity around the Magnolia implementation of Smalltalk at Tek Labs serves well to illustrate the ongoing transition to commercial era of Smalltalk. The system still encouraged a creative exploration of new ideas, but most of the focus has now shifted to building new computer applications and the methodology of object-oriented software design. The humanistic visions of Smalltalk as a medium for augmenting human intellect and a tool through which children would learn became gradually replaced with Smalltalk as a powerful object-oriented software development system. Similarly, concerns about the elegance and simplicity of its object-oriented design principles were replaced by concerns about implementation efficiency and development methodologies.

The new engineering era of Smalltalk was in many ways as fruitful as the initial humanistic era. At Tek Labs, Ward Cunningham was joined by Kent Beck, who had recently graduated from the University of Oregon. The two would form a close working relationship where ‘Ward would come up with an idea, and then Kent would go off and implement enough of it to be able to demo the idea’.[141] Their collaboration was the starting point for later developments, including Extreme Programming (XP) and Test-Driven Development (TDD), which I talked about in Chapter 4. Cunningham and Beck also got interested in what would later be called ‘design patterns’. In software development, those would only become influential later, but the investigation inspired Cunningham to develop the idea of a wiki. The growing number of people learning and using Smalltalk also got the Smalltalk advocates at Tek Labs to reflect on how to structure object-oriented programs. This led to the birth of new development methodologies, starting with the Responsibility-Driven Design technique created by Rebecca Wirfs-Brock, which I will return to later in this chapter.

Tektronix is a good example of a commercial Smalltalk provider, particularly because of the number of influential people and ideas that emerged from its Tek Labs, but it was not the only company developing a commercial Smalltalk system in the 1980s. In the late 1980s, Goldberg started a company ParcPlace Systems, where she was joined by L. Peter Deutsch, a Lisp hacker we encountered in Chapter 3, who came up with his own way of making Smalltalk run efficiently. Another company, Digitalk, ‘delivered what few thought possible’ and built a system that ran on a low-cost 8086 DOS machine around the mid 1980s. Smalltalk inspired other commercial object-oriented programming systems, most notably the Objective-C language that added Smalltalk-style objects to C in 1984 and was later adopted by Apple.

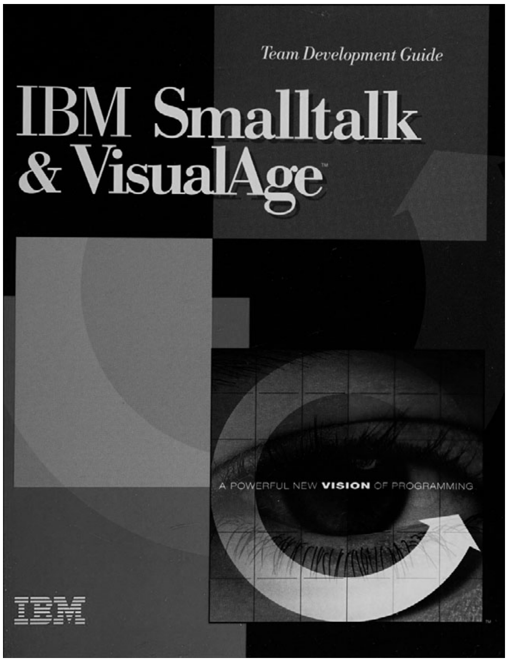

Team Development Guide
IBM Smalltalk & VisualAge
A POWERFUL NEW VISION OF PROGRAMMING
IBM

Figure 6.6 A cover of the 'Team Development Guide' manual of the IBM Smalltalk & Visual Age.
(Source: Internet Archive.[142] Reprint Courtesy of IBM Corporation ©)

In the 1990s, the companies producing commercial Smalltalk implementations have merged in various ways. Through a process that perhaps deserves a further investigation, all Smalltalk products have adopted the term 'Visual' as part of their name. The implementation from ParcPlace Systems became 'VisualWorks', the one from Digitalk became 'Visual Smalltalk Enterprise' and IBM joined in with 'IBM Visual Age Smalltalk'. On the one hand, the adoption of Smalltalk by IBM served as a significant endorsement for the commercial use of the technology. It even led some to view Smalltalk as 'the natural successor to COBOL (as opposed to C++) in many organizations'.[143] On the other hand, it also contributed to a further departure from the humanistic ideals and mathematical simplicity of Alan Kay's original vision. This can be illustrated by a quick peek into the 46-chapter-long 'Team Development Guide' (Figure 6.6) of IBM Smalltalk & Visual Age, published in 1994. The guide lists some 15+ different 'development browsers' (evolved from a single browser in Smalltalk-74) including the 'Class Browser', but also the likes of 'Application Changes Browser', 'Editions Manager' or the 'Application Configurations Browser'. The guide discusses browsers, components, applications, sub-applications, component ownership and also team roles and the development process, before showing the first Smalltalk code snippet on page 65. Despite all the complexity, the 1990s Smalltalk products served their purpose. They made it possible to use Smalltalk for commercial software development and also addressed issues such as version control, which were conceptually challenging problems in Smalltalk.

But contrary to the future of Smalltalk envisioned in the mid 1990s, the rise of commercial Smalltalk was soon to stop and the commercial sector initially occupied by Smalltalk was quickly dominated by the object-oriented programming language Java which was announced and released in 1995.

6.11 The Better Way Is Here Now

In the early 1990s, object-orientation was the dominant programming paradigm and the commercial Smalltalk industry was on the rise. Many were thinking how to enable sharing of objects across different systems, leading to the development of ‘middleware’ systems and standards such as CORBA and the establishing of the Object Management Group (OMG).[144] It is perhaps surprising that the origins of the object-oriented programming language that would soon became the de facto language of enterprise software development had initially very little to do with any of these developments.

An oft-told story about the origins of Java is that a Sun Microsystems software engineer, Patrick Naughton, was about to leave Sun and mentioned this over beers to his hockey teammate, Scott McNealy, who was also the CEO of Sun. McNealy listened to Naughton’s criticism of Sun and established a small team with unprecedented freedom to explore new avenues for the company.[145] The members of the resulting project ‘Green’ noticed that computer chips were appearing in many household appliances and decided to build a ‘device that would work as an interface to cyberspace’.[146] To create a prototype, James Gosling developed the Oak programming system. The system was built around a virtual machine, which made it easy to support multiple processor architectures. Gosling utilised his earlier experience, which included building a hardware emulator during his PhD, but he also talked with Smalltalk VM developers, including L. Peter Deutsch.[147] The ‘interface to cyberspace’ project never materialised, but the technology became useful when the web emerged in the mid 1990s. The team built a Java interpreter embedded in a web browser that made it possible to distribute Java ‘applets’ through the web. Applets contributed to Java’s popularity and enabled further developments that soon turned Java into a multi-platform object-oriented development system that quickly took over the sector previously occupied by commercial Smalltalk.

What preceded the oft-told story is an equally interesting story of how James Gosling ended up at Sun. Bill Joy, who was a co-founder of Sun Microsystems together with Scott McNealy, was interested in developing a new graphical user interface for Sun workstations. He decided to use a Smalltalk-based system and approached ParcPlace Systems, led by Adele Goldberg, to acquire a licence for their implementation. The deal never materialised because of disagreements about the royalties to be paid per machine sold. After the negotiations failed, Joy hired James Gosling to develop the system. The user interface project was ultimately unsuccessful, but the result was that Gosling was around to start working on what would eventually become Java.[148]

Although Sun occasionally mentioned possible use of Java applets for creative purposes, the programming language was built to solve a specific engineering problem and its marketing also emphasised its engineering qualities. A Java white paper published in 1995[149] first explains the ‘burden’ of the software developer who is developing applications in C or C++, faces the growth of multiple incompatible hardware architectures and user interface toolkits and now also has to cope with the Internet and the World Wide Web. According to the white paper, ‘The Better Way is Here Now’. Java and its runtime system will make ‘your job as a software developer’ much easier. The introduction talks primarily to the software engineers, uses primarily engineering arguments, but also contrasts Java with C++, rather than Smalltalk which it ended up replacing.

The rising popularity of Java inspired companies and individuals that were actively involved in the rise of the commercial Smalltalk to move to Java. Kent Beck reimplemented some of the tools developed initially for Smalltalk to work with Java, creating for example, the JUnit framework for Test-Driven Development. At the same time, IBM adapted its VisualAge programming environment to work with Java, even though it was still implemented in Smalltalk. It later reimplemented the system, creating the widely used Eclipse development environment. The relatively quick move from Smalltalk to Java is perhaps not surprising given the Java hype I mentioned in the opening of this chapter, but it was also likely supported by the cultural similarity between the two. Both commercial Smalltalk and Java in the 1990s took an engineering perspective on programming. A part of this perspective is also a pragmatic choice of suitable tools.

While the technical side of the Java language and platform has engineering origins, the origins of the development methodologies that were initially used to develop object-oriented solutions in Java are more diverse. The Smalltalk community, including Kent Back and his colleagues at Tektronix, was used to gain immediate feedback from the continuously running interactive system. This led them to highly interactive engineering-oriented development methodologies such as Extreme Programming and Test-Driven Development. Those methods were used by the Smalltalk community, who adapted them to the world of Java. But at the same time, Java quickly entered the managerial world of large enterprise systems development that was concerned about team structure and up-front modelling of the architecture of systems.

6.12 Managing the Object-Oriented Project

There is an interesting interplay between the rise of object-oriented programming, discussed here, and the move to the Agile development methodologies, which we saw emerging at Tektronix, but which I also discussed in Chapter 4. To make sense of it, we can take the inspiration from Peter Galison’s analysis of microphysics.[150] Galison argues that dividing the history of physics into distinct overarching periods is too simplistic and instead suggests analysing the history at multiple levels that evolve partly independently. In the case of microphysics, the levels include theory, instrumentation and experimentation.

Each of those evolve independently of the other. When a new instrument appears, it can initially be used in existing experiments to study existing theories. Later, its emergence enables new kinds of experiments and leads to new theories that eventually replace the old ones. Similarly, the development of new programming paradigms seems to proceed somewhat independently of the development of new development methodologies. When object-oriented programming entered the domain of enterprise software development, it was initially integrated with the existing managerial methods. Only later, the new programming paradigm supported a shift to the more lightweight Agile development methodologies. Those new methodologies then, in more recent times, remained intact when some companies embraced the functional programming paradigm.[151]

Throughout the 1970s, many software engineers were occupied with the task of finding good ways to structure large software systems. Structured programming made it possible to organise programs in a more logical way that mirrored their logic, while research on information hiding provided guidance on how to decompose programs into modules or components.[152] However, those ideas were only of limited use in the new object-oriented paradigm, especially in languages like Smalltalk that more significantly departed from the conventional ways of organising programs in structured programming.

The problem was felt clearly at Tektronix. As recalled by Allen Wirfs-Brock, when Smalltalk started spreading from Tek Labs, the researchers ‘had to start developing ways to … teach people how do you actually build something using [an object-oriented] language. … Fundamentally, how do you design software using objects’.[153] Thinking about a new object-oriented programming methodology became a major challenge for a group of Smalltalk researchers at Tek Labs. The group included Rebecca Wirfs-Brock, who moved from the graphics group to work on Smalltalk and Ward Cunningham with Kent Beck that we encountered earlier. In trying to formulate the object-oriented design methodology, the group could rely on their experience using Smalltalk, but even they did not have a single widely accepted methodology yet. The key change in thinking they identified was not to think of control structures that direct objects to do things, but instead focus on the interactions between objects.[154] Ward Cunningham took this design approach to the extreme and had an ‘actor-oriented approach’ with objects playing active roles in the design, producing designs that ‘were quite different from others’. Rebecca Wirfs-Brock didn’t carry her designs to that extreme, but she also started explaining things in terms of responsibilities that objects had.[155]

The thinking about object-oriented design and experience with internal developer training resulted in multiple publications presented at the conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) that was established in 1986 (and was also the place where Alan Kay delivered his keynote speech some 10 years later). Rebecca Wirfs-Brock wrote a paper that captured her approach and referred to it as Responsibility-Driven Design.[156] The paper also started a long-running tradition of using the format of ‘something-driven’ in the naming of development methodologies. Wirfs-Brock contrasted the approach with the established

Data-Driven Design, where the focus was on the different information stored by objects. The work can be seen as another contribution to the debate about the nature of object-orientation that previously contrasted abstract data types (data-driven) with messaging (responsibility-driven). Beck and Cunningham[157] augmented the more abstract approach with the specific tool of Class-Responsibility-Collaboration (CRC) cards which were initially used in teaching, but were also adopted by designers to identify the responsibilities and collaborating classes of a given class.

The reflections on object-oriented design written by Tek Labs researchers had an engineering focus. They were mainly concerned with how to organise code into classes. However, they remained embedded in the broader thinking about development methodologies of the time. This can be seen from Rebecca Wirfs-Brock's book *Designing Object-Oriented Software*[158] that was published a year after her earlier paper and expanded on her design ideas. In the book, she describes how object-oriented design fits with the spiral software lifecycle, which I mentioned in Chapter 4. Here, software is developed in phases well known from the earlier Waterfall methodologies (requirements specification, design, implementation and testing) that are repeated iteratively in a spiral. Rebecca Wirfs-Brock argued that object-oriented design makes the design phase longer and the implementation phase shorter, but the basic lifecycle is not questioned in her book. This may be a surprise today, given that her colleagues at Tek Labs were already using many of the practices that would later become the cornerstones of Agile methodologies, but it is likely a case of 'intercalated periodisation' identified by Galison where a shift at one level (here, development methodologies) does not typically align with shift at another level (here, programming paradigms). The new approach was emerging, and Christiane Floyd has described it already in 1987, but it would take longer for it to fully take place.

Other major contributions to object-oriented design came from a world that was even more remote from the original humanistic origins of Smalltalk. Grady Booch got his undergraduate degree from the US Air Force Academy, had experience building ground support software for the space shuttle, and joined the company Rational Machines (later Rational Software) in 1981 as the Director of Software Engineering Projects. Given his association with the military, it is not surprising that his early work mostly involved the Ada programming language, but he also became a prominent Ada educator. In 1983, Booch wrote a book *Software Engineering with Ada* that has some of the first traces of what would later become an object-oriented design methodology. This includes the idea of looking at English descriptions of systems.[159] In the words of Booch:

> If we examine human languages, we find that they all have two primary components, noun phrases and verb phrases. A parallel structure exists in programming languages, since they provide constructs for implementing objects (noun phrases) and operations (verb phrases).[160]

Booch notes that top-down structured design methods, used in the context of imperative programming, are like trying to communicate with just verbs, while design focused on data structures is like trying to communicate with just nouns. Taking inspiration from Parnas' work on decomposing systems into components,[161] Booch argues for a design that decomposes software systems into objects that capture aspects of both the data structure design (nouns) and the imperative design (verbs). Each object consists of some state, associated actions and a class that determines what kind of object it is. Notably, the term 'class' is used somewhat loosely in the text, because the Ada language at the time did not have an explicit support for what we would today refer to as object-oriented programming constructs.[162] The objects in the book are thus implemented as Ada packages that use local state, type declarations and procedures.

Similar to Rebecca Wirfs-Brock, Grady Booch saw object-oriented design as a part of the broader application development lifecycle. He makes it clear that 'object-oriented development is a partial life-cycle method' that 'focuses on the design and implementation phases of software development'. The design methodology is embedded in the conventional methodology with analysis, requirements gathering, design, coding, testing and maintenance. In the book, Booch illustrated the lifecycle using sketches of the ageing Ada, Countess of Lovelace, with a final sketch of Ada on her deathbed in the 'Operation and Maintenance Phase' section.

The engineering perspective on object-oriented programming dominated the 1980s, but it was not the end of the story. In fact, Rentsch's humorous remark that object-oriented programming was going to be the structured programming of the 1980s had more truth in it than it may seem. Structured programming started as an engineering notion and was later adopted by the managerial culture of programming. Object-oriented programming met the same fate. Over the following decade, approaches built around object-oriented development eventually expanded to other phases of the software development lifecycle and started to be used not just for engineering new systems, but also for managing their development. Following the later work of Grady Booch is a good way to illustrate this gradual transition.

At the turn of the 1990s, Booch wrote an influential book *Object-Oriented Design with Applications*. In contrast to his earlier work, this is not focused on the Ada programming language. It discusses examples implemented (or sketched) in multiple languages, still including Ada, but also Smalltalk, Object Pascal, C++ and the Common Lisp Object System. The broader focus on object-orientation also gained it an endorsement from Bjarne Stroustrup. Like in the earlier work, object-oriented concepts are limited to the design and implementation phases of a conventional lifecycle, this time with reference to the same spiral development model that Rebecca Wirfs-Brock talked about around the same time. The book recognises the difficulty of classifying things and, consequently, the difficulty of identifying meaningful classes and operations. It reviews various approaches, including the one based on identifying nouns and verbs, but offers no single panacea. One important characteristic is that, just like the overall development process, the design process through which the structure of the system emerges needs to be iterative.[163]

To help the process of designing a software system using objects, the book introduces a set of diagrams that can be used to capture the design of the system, including its structure of classes, but also dynamic properties such as state transitions and timing.

As the introduction to ‘The Method’ section makes clear, the book is firmly rooted in the engineering culture:

> If you follow the work of any engineer – software, civil, mechanical, chemical, architectural, or whatever – you will soon realise that the one and only place that a design is conceived is in the mind of the designer. As this design unfolds over time, it is often captured on such high-tech media as white boards, napkins, and scraps of paper.[164]

Booch then explains the benefits of using a shared notation over napkin scribbles. It makes it possible to communicate the design to others, ‘relieves the brain of unnecessary work’ and enables using automated tools for checking the design. The development of unified modelling notation for object-oriented design is exactly what happened over the next couple of years. In the early 1990s, many others were trying to devise methodologies, processes and modelling notations for developing object-oriented systems. This included Ivar Jacobson, who developed the Object-Oriented Software Engineering (OOSE) methodology and James Rumbaugh, the creator of the Object Modeling Technique (OMT). All three methods consisted of a range of notations for capturing the design decisions as diagrams and recommended processes for obtaining the designs. By 1995, Rumbaugh and Jacobson joined Booch at Rational and the three started working on a common language, the Unified Modelling Language (UML), which was adopted by the Object Management Group (OMG) as a standard in 1997. The purpose of the UML was not to replace all the different methodologies for developing object-oriented designs, but merely to provide a common language for the result. However, the ‘Three Amigos’ at Rational, as they became known, eventually joined forces in developing a single Rational Unified Process (RUP).

Even though the Three Amigos initially approached object-orientation from the engineering perspective, the development of the RUP puts object-orientation clearly into the realm of the managerial culture of programming. The shift in the focus is clear from the first paragraph of the preface of their joint book *The Unified Software Development Process* that describes a non-proprietary version of the process developed at Rational Software:

> There is a belief held by some that professional enterprises should be organised around the skills of highly trained individuals. They know the work to be done and just do it! They hardly need guidance in policy and procedure from the organisation for which they work. This belief is mistaken in most cases, and badly mistaken in the case of software development.[165]

The book goes on to discuss the characteristics of the process, which is iterative and covers the entire lifecycle, starting from the capturing of requirements to testing, release and maintenance. The managerial approach is apparent in that the book no longer tries to structure just the classes of an object-oriented system, but the work of the team delivering an object-oriented project. In the attempts to avoid organising the development around highly trained individuals, it is reminiscent of the motivations that led to the 1968 NATO Software Engineering conferences as discussed in Chapter 2. Some 30 years earlier, the managers similarly tried to replace the reliance on ‘geniuses and mavericks’ and the ‘black art of programming’ with ‘science of software engineering’.

The Rational Unified Process dominated the software engineering world around the year 2000. It was ‘in fashion and everything else was considered out of fashion and more or less thrown out’.[166] The process itself was not necessarily heavyweight, but it was often perceived as such. Rational Unified Process was a process framework that could be instantiated in various ways, but most of its instantiations ended up being very complex and that was probably one of the factors that contributed to the 2000s rise of Agile methodologies that restored the more immediate, interactive ways of working that originated at Tektronix in the 1980s.

The clash between the Agile methodologies and processes such as the Rational Unified Process can be interpreted as a struggle between two cultures of programming. Whereas the managerial approach focused on organising teams, the engineering culture preferred focus on individual responsibility. The principle ‘individuals and interactions over processes and tools’ that was one of the points made in the 2001 Agile manifesto[167] can be read as expressing this exact cultural preference. However, some criticism of the managerial methods came from the Three Amigos themselves. In 2019, Ivar Jacobson referred to the evolution of development methodologies since the NATO Software Engineering conference as the ‘Fifty Years’ War’ of methods. His criticism focused on the fact that ‘once you have adopted a method, you get the feeling you are in a method prison controlled by the guru of that method’, which is ‘the craziest thing in the world’.[168] Even Grady Booch once remarked that he ‘cannot be held responsible for all the stupid uses of the UML’.[169] In some ways, much of the criticism of the managerial object-oriented methodologies is nothing new. It is reminiscent of the 1988 remark by Edsger Dijkstra who criticised the software engineering discipline, claiming that it has accepted as its charter ‘how to program if you cannot’.[170]

6.13 Object-Oriented Programming, Systems, Languages and Applications

Object-oriented programming is yet another example of a concept that has been influenced and reshaped by multiple cultures of programming. So far, I followed one particular trajectory in this chapter. The basic object-oriented concepts emerged in the context of computer simulations, where they were mainly shaped by the mathematical culture of programming. They were then reinterpreted as new media for communication during the early days of Xerox PARC, primarily from the perspective of the humanistic culture, while others at PARC found Smalltalk a flexible and powerful tool for tinkering with computers in ways favoured by the hacker culture. This diversification of interest in Smalltalk likely prevented a major language redesign in 1976. I then followed the widespread adoption of object-oriented programming, which saw the emergence of new languages that combined the basic concepts with different mathematical, engineering, hacker and humanistic aspects, and we also saw how Smalltalk itself turned from a humanistic vision into a practical commercial tool and the birthplace of many new engineering approaches. As with structured programming a decade earlier, the managerial culture also found its way of turning object-oriented ideas into a basis for development methodology that aims to reduce risks in software development by controlling the development process.

By following a single path, I have inevitably left out many important events and directions. One such key event was the birth and the evolution of the ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) where Alan Kay delivered the 1997 keynote that opens this chapter.[171] The first OOPSLA conference took place in 1986 in Portland, which was the hometown of Tektronix and one of the centres of object-oriented technology at the time. OOPSLA was born out of a feeling that the Smalltalk community 'needed some sort of conference'[172] and was co-organised by Adele Goldberg, the president of the ACM at the time, Allen Wirfs-Brock from Tektronix, David N. Smith who worked on Smalltalk at IBM Research and Tom Love who was involved in the development of Objective-C. Already in 1986, the conference attracted some 600 attendees and it grew to over 2,500 in the 1990s.

In its early days, the OOPSLA conference attracted a mix of academic and practitioners. As recalled by Allen Wirfs-Brock, 'there was not that much difference between the academic papers … and the papers that originated from industrial research labs'.[173] The early publications were generally engineering-oriented and aimed to present the developing object-oriented technology to the broader audience. Many of the early OOPSLA publications were presentations of new programming systems. The conference brought together like-minded, but somewhat disconnected communities with different interpretations of object-orientation. In addition to contributions from the engineering-oriented commercial Smalltalk community, which I followed earlier, the first OOPSLA featured multiple application papers, theoretical papers, but also multiple papers presenting implementations of object-oriented ideas in Lisp.

The work on object-oriented programming in Lisp emerged in the context of interactive programming systems, which I talked about in Chapter 3. Similarly to Smalltalk, Lisp was not merely a language, but a stateful, interactive programming environment through which the computer was programmed, as well as used. In the 1970s, there were multiple personal connections between the two communities that originated in the earlier ARPA days. Through those connections, ideas about message passing in Smalltalk influenced a version of Lisp built at MIT for the Lisp Machine project. A new language feature, called Flavors, which added object-orientation to Lisp, was created by Howard Cannon and David Moon and was described in a report in 1980.[174] By 1986, there were multiple competing object-oriented extensions to Lisp including New Flavors and CommonLoops, both presented at the first OOPSLA, as well as other systems such as the prototype-based Object Lisp. One of the notable innovations of CommonLoops over competing Lisp designs was that the object-oriented system itself was defined in terms of meta-classes. As the name suggests, meta-classes controlled the features and the representation of the object-oriented programming system. This way, the definition of the object-oriented system itself leverages the object-oriented extensibility and becomes open to extension and modification.

Throughout the 1980s, there was an ongoing effort to standardise the many dialects of Lisp, which also included developing a single object-oriented Lisp extension. At the end of the 1980s, this resulted in the Common Lisp Object System (CLOS), which was mainly modelled after New Flavors and CommonLoops. Although the eventual Common Lisp standard does not specify the details, most CLOS implementations adopted the CommonLoops idea of using a meta-object protocol, which makes the resulting object system customisable. Through the meta-object protocol, CLOS realises one of the visions of Alan Kay who argued that the only way of handling the complexity of software systems, at least until ‘proper engineering methods’ are found, is to have extreme late binding and extreme flexibility so that systems can evolve over time in response to new needs and requirements.[175] The fact that main-stream object-oriented languages lack this flexibility was also one of the points that Kay raised in his 1997 keynote where he recalls looking at Java and thinking ‘[my] goodness … how do they hope to survive all of the changes, modifications, adaptations, and interoperability requirements without a meta-system’.[176]

The meta-object system in CLOS was the starting point for a 1991 book *The Art of the Metaobject Protocol*[177] that described the basic idea, alongside with its concrete simplified implementation. In his keynote, Kay claimed that he recalled the authors saying:

> This is the best book anybody has written in ten years, but why the hell did you write it in such a Lisp centric, closed club centric way?[178]

According to Kay, the book is very hard to read if one does not already know how CLOS is implemented, but it has ‘some of the most profound insights about, and the most practical insights about OOP’. The fact that the OOPSLA conference brought together different communities thinking about object-oriented programming was thus apparently not enough to fully bridge the gaps between their individual peculiar interpretations and ways of talking and, perhaps surprisingly, this was the case even between the Smalltalk and Lisp communities, which both treated programming as interaction with a stateful, living system and had similar cultural origins.

The OOPSLA conference can be also used to illustrate another important shift that occurred in the history of object-oriented programming and, perhaps more broadly, programming. Many of the systems that were discussed in the early days of OOPSLA followed the view of programming as interacting with a stateful, living system that was shared by the Smalltalk and Lisp communities. This is the view emphasising interactivity that I retraced throughout the history of programming in Chapter 3. The view is not strictly linked to a particular culture of programming, although it is easier to reconcile with the hacker mindset, focused on direct engagement with the machine, and with the humanistic vision of augmenting human intellect. Throughout the 1990s, the focus of OOPSLA has gradually shifted from discussion of engineering aspects of new systems to more academic theoretical research. But alongside came a shift from interactive programming systems to static programming languages. The shift alienates some of the earlier members of the community, who felt they could no longer read most of the papers.[179] Richard P. Gabriel interpreted the shift more strongly as a Kuhnian paradigm shift. In a 2012 paper ‘The Structure of a Programming Language Revolution’,[180] he argued that there is a degree of incommensurability between the earlier ‘programming system’ and the later ‘programming language’ perspective.

Through the topic of object-oriented programming, we have completed the full circle. I started the book by retracing the origins of two major different ways of thinking about programming. In Chapter 2, I followed the efforts of the mathematical culture to mathematise programming and establish programs and programming languages as formal mathematical entities. In Chapter 3, I then shifted focus to the influences of the hacker and humanistic cultures that viewed programming as interaction with a stateful system. Object-oriented programming has been reconceptualised in major ways, for better or worse, through the shifts between these two views and their respective cultures.

Notes

1 Mitchell (2003).
2 The exact phrasing used here, including the terms ‘almost a new thing’, ‘old thing’ and ‘new thing’ are due to Kay (1996).
3 Mitchell (2003).
4 Kay and Goldberg (1977).
5 Parnas (1972); Liskov and Zilles (1974).
6 The most detailed recent contribution to the discussion has been written by Cook (2009).
7 Whitaker (1993).
8 Dijkstra (1988).
9 First described by Abbott (1983), but made broadly known by Booch (1983).
10 Paraphrasing an answer given by Allen Wirfs-Brock to an audience question about ‘the failure of object-oriented programming’ at a conference. Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
11 Paraphrasing Alan Kay’s answer from an interview by Greelish (2013).
12 Ingalls et al. (1997, 2008), but note that these are large projects with multiple contributors beyond their original creators.
13 The idea of bringing some of the visions behind Smalltalk to UNIX and C has been explored by Kell (2018).
14 Quoted from a letter to the editor in the Dr. Dobb’s Journal (Passani, 1996).
15 Wirfs-Brock (2020).
16 Kay (1997).
17 Brand (1972).
18 It is possible that this claim is rooted in broader Western culture, where objects play a crucial role as philosophical entities, but examining this assumption and its alternatives is beyond the scope of this chapter.
19 Nance (1993).
20 Nygaard and Dahl (1978) report that they were familiar with the language SIMSCRIPT, which provided some inspiration for list processing, time scheduling mechanisms and the procedure library in SIMULA. They were aware of GPSS, but have not studied it closely.
21 Letter to Charles Salzmann, dated January 5, 1961, quoted in Nygaard and Dahl (1978).
22 Nygaard and Dahl (1978).
23 The social and political aspects of the development of SIMULA have been described by Holmevik (1994).
24 Holmevik (1994).
25 Nygaard and Dahl (1978).
26 Nygaard and Dahl (1978).
27 Dahl (1969).
28 Dahl and Nygaard (1966).

29 Nygaard and Dahl (1978).
30 Dahl et al. (1972).
31 Nygaard and Dahl (1978) identifies four different stages. What I refer to as the 'first stage' is their stage 2 and what I refer to as the 'second stage' is their stage 4.
32 Presented in a Communications of the ACM paper (Dahl and Nygaard, 1966).
33 Dahl and Nygaard (1966).
34 Nygaard and Dahl (1978).
35 Nygaard and Dahl (1978). It is worth pointing out that this is exactly what the implementation of async/await (Syme et al., 2011), in modern programming languages such as C#, JavaScript or F# does.
36 Nygaard and Dahl (1978).
37 Nygaard and Dahl (1978).
38 Nygaard and Dahl (1978).
39 Hoare (1965).
40 Compare the lecture notes 'to be delivered' Hoare (1965) with the published proceedings (Genuys, 1968); the likely influence resulting from the meeting is also suggested by Black (2013).
41 Hoare (1965).
42 Dahl et al. (1968).
43 Dahl and Nygaard (2002), also quoted by Black (2013).
44 Sammet (1969), referenced by Nance (1993).
45 Sammet (1969).
46 Dahl et al. (1968).
47 Kay (1996) and Ingalls (2020), although, I do not accept those retrospectives uncritically and also aim to find supporting or contrary evidence in the available primary sources. A useful high-level account of the history of Smalltalk has also been written by Priestley (2011).
48 The title of this section refers to those influences and ongoing technology changes that make them possible. It is also a quote from a November 1971 panel discussion abstract, Kay (1972a).
49 Kay and Goldberg (1977).
50 Kay (1972b).
51 Ware (1966).
52 Kay (1996).
53 Kay (1968).
54 Kay (1969).
55 The term 'personal computer' is used anachronistically here. The term was used by Fisher et al. (1975) in reference to the LINC computer. It appears earlier, in the 1965 Air Force report, and then in a paper by Kay (1972b), but it was also used around the same time for talking about programmable calculators, for example, by Tung (1974).
56 Kay (1996).
57 Kay (1969).
58 Kay (1969).
59 Kay (1969).
60 Landin (1965a,b).
61 Hiltzik et al. (1999) offers a glimpse of those, although not entirely unbiased.
62 Kay (1996).
63 Using modern programming language terminology, the `if` construct requires lazy evaluation (or call-by-name strategy) rather than eager evaluation (or call-by-value strategy).
64 According to the timeline given in the Appendix IV of the published version of Kay (1996).
65 Goldberg and Kay (1976b).

66 Goldberg and Kay (1976b).
67 Kay (1996).
68 Kay (1996).
69 Correspondingly, writing on SIMULA 67 and Smalltalk-72 is much easier to read for a modern computer scientist than writing on their immediate predecessors.
70 Kay (1996).
71 Kay (1996).
72 Kay (1996).
73 Kay (1974).
74 https://bitsavers.org/pdf/xerox/smalltalk/ALLDEFS_Apr75.pdf, retrieved May 16, 2024.
75 Goldberg (2002).
76 Ingalls (2020).
77 Rentsch (1982).
78 Kay (1996) and Ingalls (2020).
79 See Chapter 3.
80 Ingalls (2020).
81 Ingalls (1978).
82 Kay and Goldberg (1977).
83 Kay (1996).
84 For example, in the paper by Ingalls (1978).
85 Bobrow and Winograd (1976).
86 Bobrow and Winograd (1976).
87 Kay (2003).
88 Liskov and Zilles (1974); As pointed out by Cook (2009), a paper by Zilles (1973) discussed the ideas around data abstraction slightly earlier and was closer to the perspective of Smalltalk. It mentions the term ‘object’ but not ‘object-oriented’.
89 Jones and Liskov (1976).
90 Kay (2003).
91 Kay (1996).
92 Kay (1998).
93 Cook (2009).
94 Quoted from the chairman’s introduction of the May of 1969 Extensible Languages Symposium, (Cheatham, 1969).
95 Kay (1969).
96 Kay (1968).
97 Kay (1996).
98 Fisher (1970).
99 Kay (1996).
100 Kay (1996).
101 Petricek (2023).
102 Liskov and Zilles (1974).
103 Rentsch (1982).
104 The rise of object-oriented programming is contextualised in a retrospective on Objective C by Cox et al. (2020), while the irony of the recurring feeling of a crisis in a financially successful industry has been aptly pointed out by Ensmenger (2012).
105 Cook (1988).
106 Madsen and Møller-Pedersen (2022).
107 Later, the meeting of the object-oriented and British mathematical world was one of the starting points for the development of the F# programming language, as discussed by Syme (2020).
108 Cardelli and Wegner (1985).
109 Cardelli (1984); Cook and Palsberg (1989).

110 Abadi and Cardelli (1996).
111 As discussed in the first-hand retrospective by Kristensen et al. (2007).
112 Kristensen et al. (2007) acknowledges this, but also points out that this view often comes from people not using BETA.
113 Ungar and Smith (2007).
114 Ingalls et al. (1997).
115 Ungar and Smith (2007).
116 Ungar and Smith (2007).
117 Smith and Ungar (1995).
118 Borning (1986) and also an earlier publication on programming aspects of ThingLab (Borning, 1981).
119 Smith and Ungar (1995).
120 The Self authors do, however, acknowledge the influence of the mathematical culture on the design of the language in its striving for simplicity in the language design in Ungar and Smith (2007).
121 For the analysis of the crisis narrative in the field, see the excellent work of Ensmenger (2012); Ryder et al. (2005) relates a number of developments in programming languages to parallel work on software engineering.
122 Liskov (1993).
123 The overall project history is documented in a retrospective by Whitaker (1993). Technical rationale for the design can be found in the extensive 200+ page report by Ichbiah et al. (1979).
124 Whitaker (1993).
125 Whitaker (1993).
126 Meyer (1992).
127 Meyer (1992).
128 Stroustrup (1996); The author has also written more about the design of the language (Stroustrup, 1994), and its more recent evolution (Stroustrup, 2007).
129 Stroustrup (1996).
130 Stroustrup (1996).
131 Stroustrup (1986); incidentally, also the book containing the quote that gave title to this section.
132 Stroustrup (1988).
133 ‘The fate of facts and machines is in the hands of latter users’ (Latour, 1987).
134 Thacker et al. (1982).
135 Goldberg (2002), Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
136 Goldberg (2002).
137 At Tektronix, the performance was about 1% of the Xerox workstation and their first prototype Smalltalk system took an hour to render the welcome screen, Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
138 Goldberg and Robson (1983), nicknamed after the colour used on the book cover.
139 Wirfs-Brock (2013) and Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
140 Thomas (1995).
141 Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
142 https://archive.org/details/ibmsmalltalkvisu0000unse. IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at ‘IBM Copyright and trademark information’ at www.ibm.com/legal/copytrade.shtml.
143 Hunt (1997).
144 PROGRAMme (2022).

145 There is no definitive account of the history of Java. I draw on multiple sources including Thagard and Croft (1999); Bank (1995) and archived articles published by Sun, English (1998); Byous (1998).
146 Bank (1995).
147 Allman (2004).
148 Richard P. Gabriel (personal communication, 17 February 2024).
149 Gosling and McGilton (1995).
150 Galison (1997).
151 An example of this is the Domain-Driven Design methodology that has been initially developed in the object-oriented context, but has been swiftly adapted for functional programming, as described by Wlaschin (2018).
152 The key theoretical reference was the work of Parnas (1972), which later inspired the development of language features such as modules in Ada.
153 Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
154 ‘Most people tend to think of control structures and directing the objects to do things as opposed to having objects interacting’, Rebecca Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
155 Rebecca Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
156 Wirfs-Brock and Wilkerson (1989); the paper was co-authored with Brian Wilkerson.
157 Beck and Cunningham (1989).
158 Wirfs-Brock et al. (1990).
159 The idea was inspired by a paper by Russell Abbott from the same year (Abbott, 1983).
160 Booch (1983).
161 Parnas (1972).
162 Ada made it possible to define subtypes and derived types, but those had a somewhat different meaning than in later object-oriented languages. A subtype defines a new name for a constrained type, i.e., values of another type that satisfy some condition (restricting the range of its values). A derived type is a new distinct type, but with the same structure as another type.
163 The account draws on the excellent overview article Lee (2021).
164 Booch (1990).
165 Jacobson et al. (1999).
166 Jacobson et al. (2019).
167 Beck et al. (2001).
168 Jacobson et al. (2019).
169 Hohpe (2005).
170 Dijkstra (1988).
171 Another part of the history of object-oriented programming that is missing in this book is the development of design patterns that became famous in the form of a book by Gamma et al. (1995), but that also inspired reflection by Gabriel (1996), which is perhaps closer to the origin of the idea. The history, alongside other links between computer science and architecture, has been documented recently by Steenson (2022).
172 Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
173 Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
174 Weinreb and Moon (1980); for more information about the history of object-orientation in Lisp, see the first-hand account by Steele and Gabriel (1996b) and also the earlier incomplete draft of the paper, Steele and Gabriel (1996a).
175 See Alan Kay’s appendix ‘Is “Software Engineering” an Oxymoron?’ (Smith et al., 2002).
176 Kay (1997).
177 Kiczales et al. (1991).
178 Kay (1997).
179 Rebecca Wirfs Brock and Allen Wirfs-Brock, interview by Tomas Petricek, 23 March 2023.
180 Gabriel (2012).