Great Moments in the History of OOP

A Saga in 4 Parts

Oscar Nierstrasz
Software Composition Group
scg.unibe.ch

50 years anniversary of Simula

UNIVERSITÄT BERN

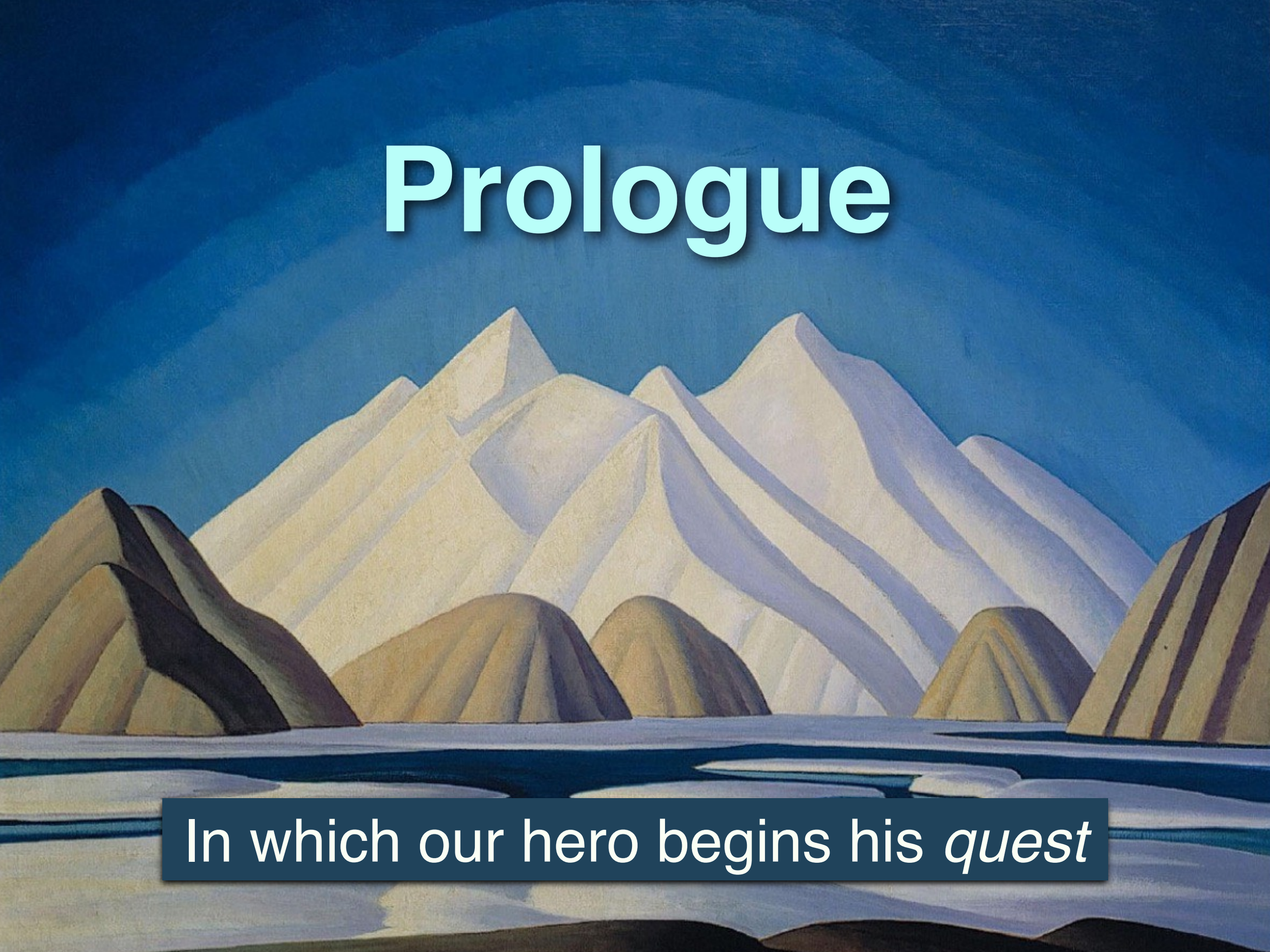
This talk was given at the 50 years anniversary of Simula celebration held in Oslo on September 26, 2017. In it, I present a personal tour of some of the milestones in the history of OOP.

http://simula67.at.ifi.uio.no/50years/

The wild hunt: Asgårdsreien (1872) by Peter Nicolai Arbo

https://en.wikipedia.org/wiki/Wild_Hunt

Prologue

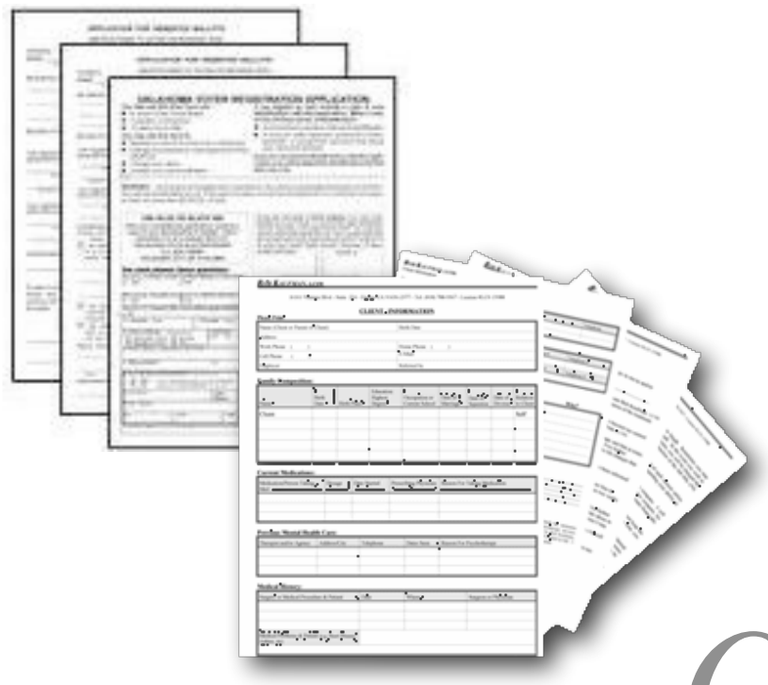


In which our hero begins his *quest*

Lawren Harris, Baffin Island, 1931

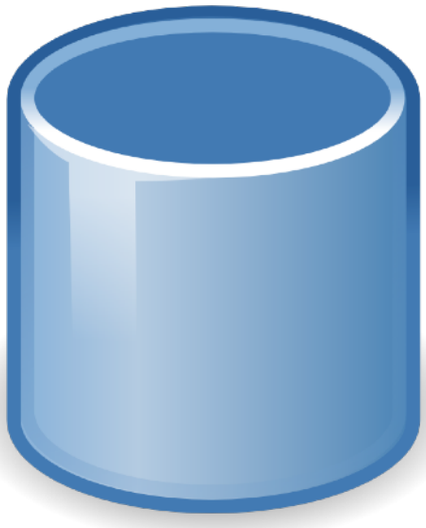
https://www.wikiart.org/en/lawren-harris/baffin-island-1931

How to build the “electronic office”?



TLA

OFS



MRS

Back in 1980, when I started my Masters thesis at the University of Toronto, I was tasked, together with John Hogg, with developing “automated procedures” for OFS, a prototype of an “Office Forms System” implemented in C. OFS was built on top of MRS, a Micro Relational System for Unix, developed within the Office Systems Group led by Prof. Dennis Tsichritzis.

Uh,
where are the
objects?

I did not have much programming experience, and C was new to me, but I thought the task seemed pretty clear. However I was very surprised to open the box and discover that the domain objects of OFS were very hard to find in the code, as they were smeared across many different levels.

I had the nagging feeling that we were using the wrong technology to implement prototypes of advanced office information system tools.

Jackson Pollock, Convergence, 1952

https://www.jackson-pollock.org/convergence.jsp

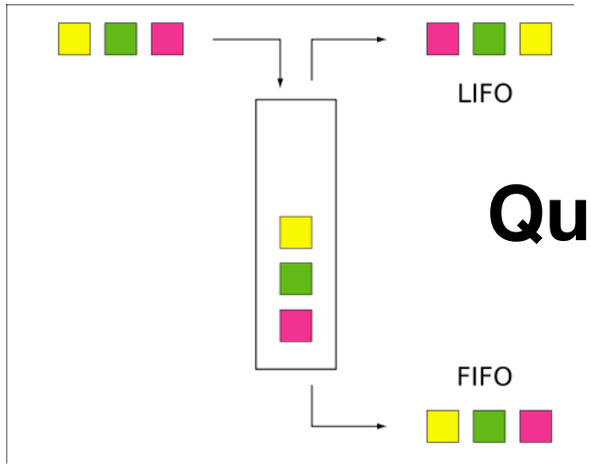
Part 1. A Call to Arms

In which we witness the *origins* of object-oriented programming

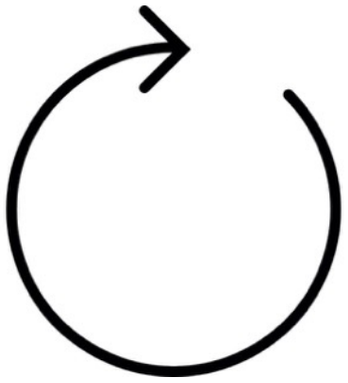
Frank Dicksee, The Funeral of a Viking, 1893

https://www.wikiart.org/en/frank-dicksee/the-funeral-of-a-viking-1893

simula



Queues vs stacks



“Process” (object) as unifying concept

Inheritance (“prefixing”) — adding layers to classes

1962-1967

The Birth of Object Orientation: the Simula Languages. 2004

Back in 1962, Ole-Johan Dahl and and Kristen Nygaard became convinced of the need for explicit support for simulation in programming languages. Over a period of four years, they identified three core ideas. First, queues were needed to model events over time. Second, an explicit notion of a (quasi-parallel, communicating) process (or “object”) was needed as a unifying concept. Finally, “prefixing” (inheritance) added to allow sharing of properties.

The Birth of Object Orientation: the Simula Languages. 2004

http://www.olejohandahl.info/old/birth-of-oo.pdf

The History of Simula, 1995, Jan Rune Holmevik

http://campus.hesge.ch/daehne/2004-2005/langages/simula.htm

Photo:

https://history-computer.com/ModernComputer/Software/Simula.html

http://www.jot.fm/issues/issue_2002_09/eulogy/

simula

Programming
is simulation

Simula was designed as an extension to Algol to support programming of simulation applications. As it turned out, this was useful for more than just simulation programming. In a sense, Dahl and Nygaard were saying that “Programming is simulation” since any software system could be seen as a set of cooperating objects.

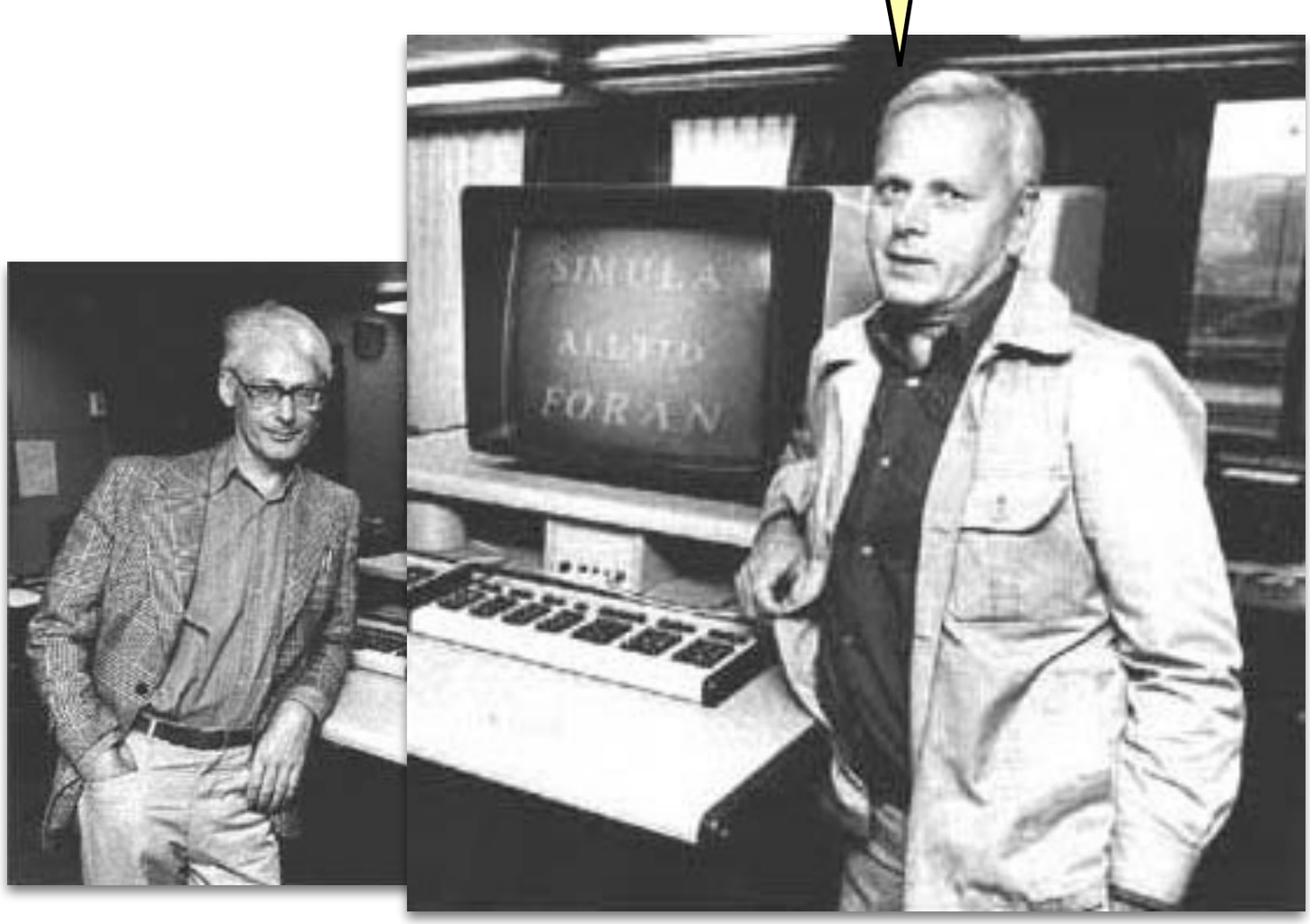
simula

Programming is modeling

Reading between the lines, we could also say that a simulation is a model, hence “Programming is modeling.”

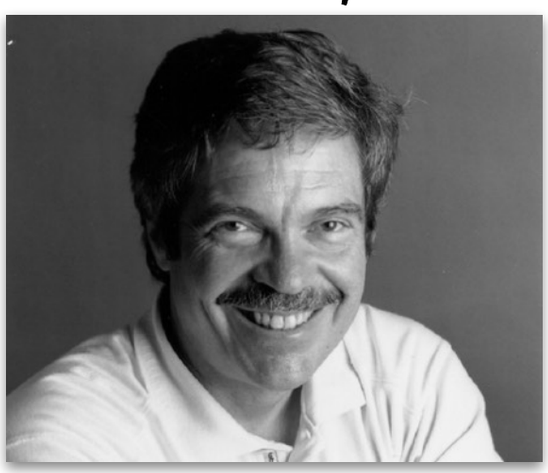
simula

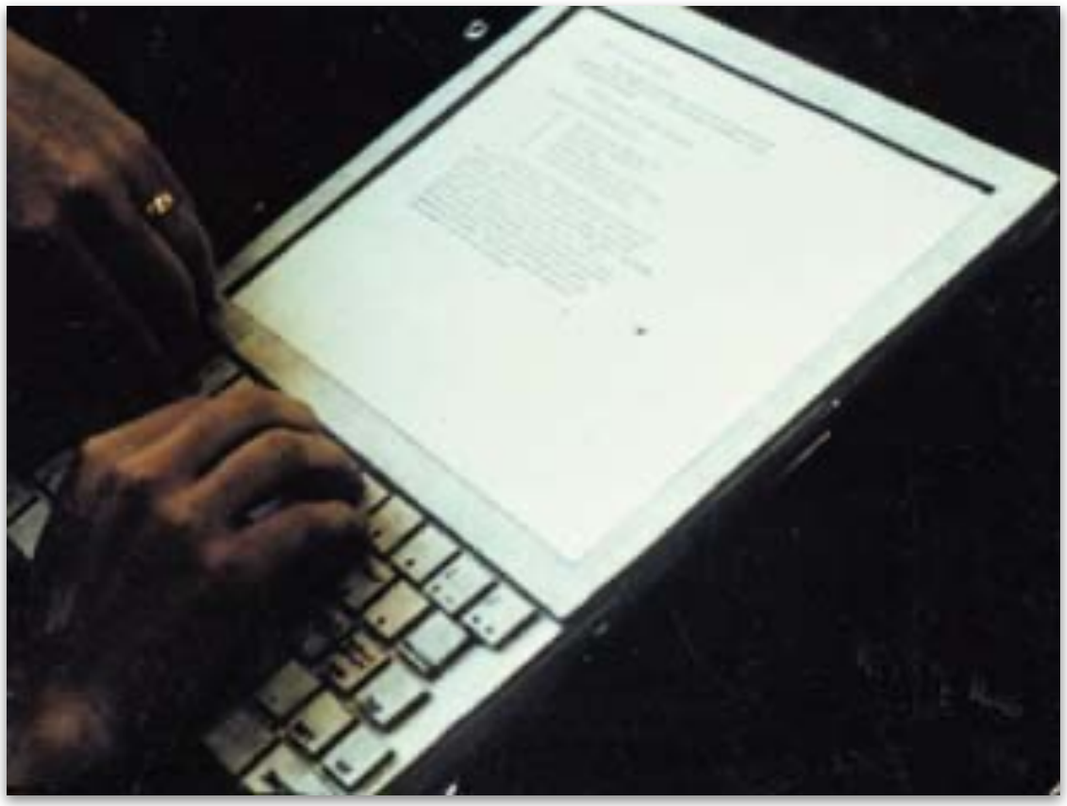
Programming is understanding



But what Kristen Nygaard is actually credited with saying is that “Programming is understanding,” which is arguably a more succinct way of expressing the same thing.

It’s objects all the way down





“Dynabook” mockup ca. 1970

Around this time Alan Kay came to the realization that increasing computing power and decreasing costs would soon lead to a new generation of “personal” computers. He envisioned a hand-held multimedia device that he code-named the “Dynabook”. He was convinced that in order to build such systems, we would need not just object-oriented languages, but systems that would consist of objects all the way down to the lowest levels.

When pressed on this, he is told to have explained, “Look, it’s all objects all the way down. Until you reach turtles.”

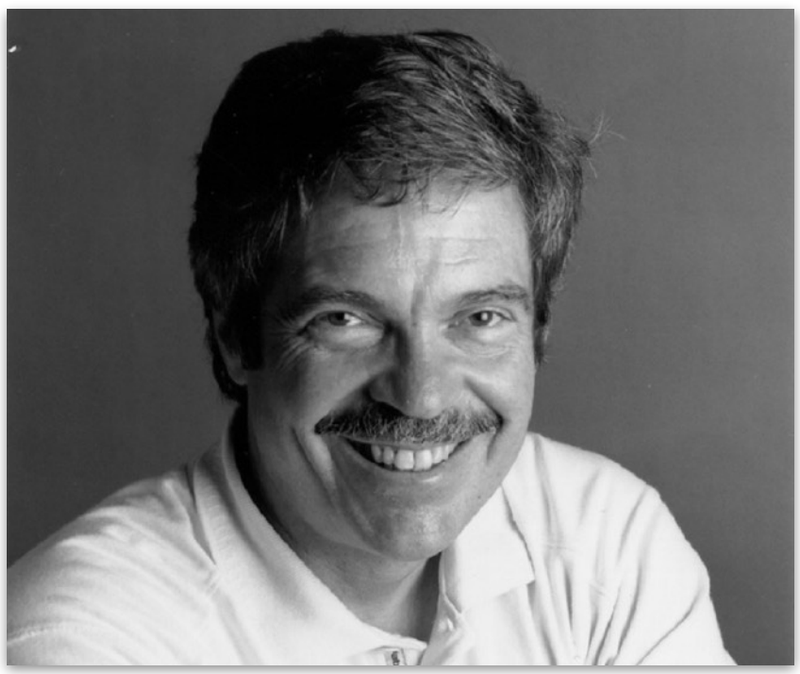
The Dynabook of Alan Kay

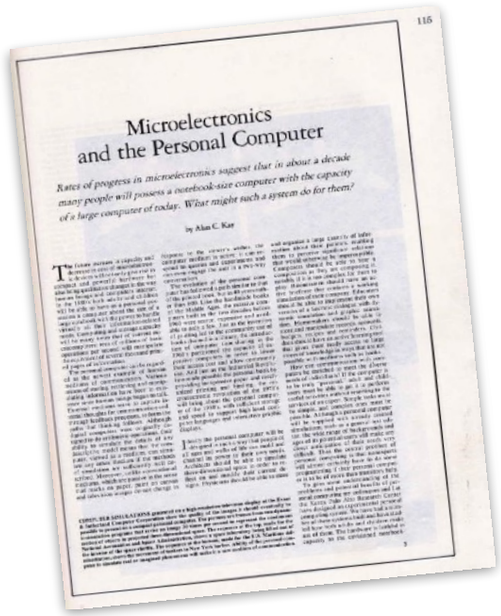
http://history-computer.com/ModernComputer/Personal/Dynabook.html

A Brief, Incomplete, and Mostly Wrong History of Programming Languages

http://www.cvaieee.org/html/humor/programming_history.html

Computation is simulation





Microelectronics and the Personal Computer

Rates of progress in microelectronics suggest that in about a decade many people will possess a notebook-size computer with the capacity of a large computer of today. What might such a system do for them?

by Alan C. Kay

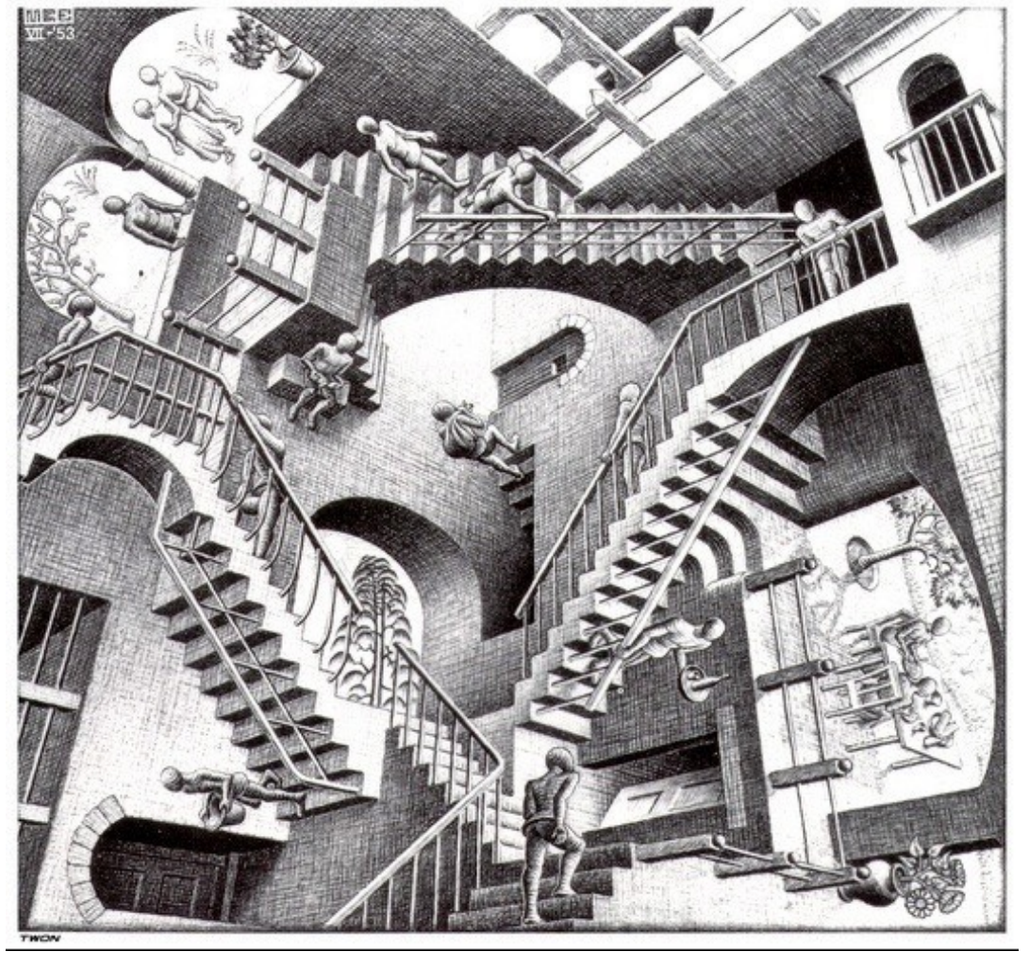
Microelectronics and the Personal Computer, 1977

Inspired by Simula, Kay was saying that not just programming, but “Computation is simulation.”

“The social impact of simulation — the central property of computing — must also be considered.” Alan Kay, 1977, “Microelectronics and the Personal Computer”

http://mnielsen.github.io/notes/kay/micro.pdf

Programming
is *objects talking to objects*



In Smalltalk,
everything happens
somewhere else

Kay assembled a team at Xerox PARC and over a period of ten years developed the Smalltalk system, which was not just a language, but also an operating system (virtual machine) and a development environment, including multimedia hardware.

Dan Ingalls, explaining the design principles behind Smalltalk, “Instead of a bit-grinding processor … plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires.”

Adele Goldberg interestingly is credited with saying that, “In Smalltalk, everything happens somewhere else.” On one hand, this expresses nicely the principle of delegation in good OO design, but it also points out some of the difficulties inherent in understanding complex OO systems.

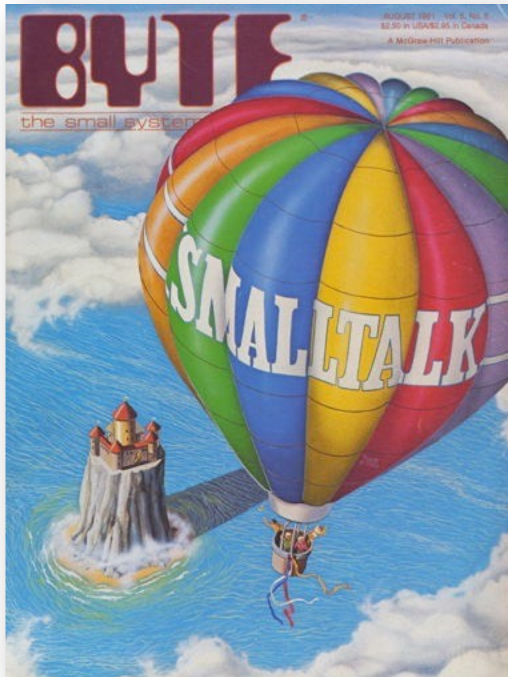
Countering this, Alan Knight advises: “One of the great leaps in OO is to be able to answer the question “How does this work?” with “I don’t care”.”

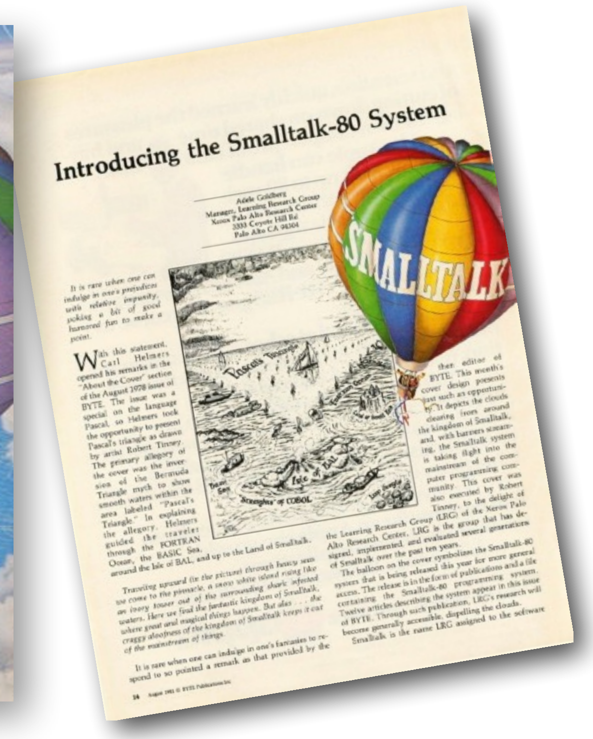
Design Principles Behind Smalltalk, Byte Magazine, August 1981.

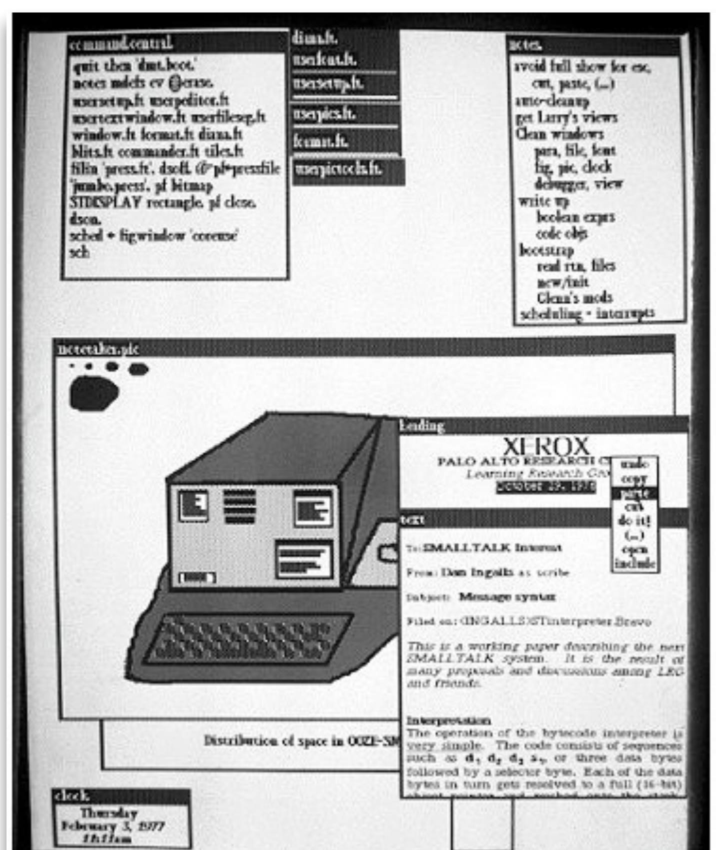
https://archive.org/details/byte-magazine-1981-08

Escher, Relativity, 1953

https://en.wikipedia.org/wiki/File:Escher%27s_Relativity.jpg







Uh, what's a "Dorado"?

Reading the August 1981 issue of Byte magazine, I was blown away by the description of the Smalltalk 80 system. I was convinced that this was what we needed for developing our advanced OIS prototypes. Unfortunately it only seemed to run on the experimental workstations, known as the “Dorado”, developed within Xerox PARC.

I spoke to my boss, Dennis, about it, and he said, “Why don’t you grab a couple of Masters students and build yourself an object-oriented system.” I started to do that, but that’s another story ...

Using Objects to Implement Office Procedures, Nierstrasz, Mooney, Twaites, 1983

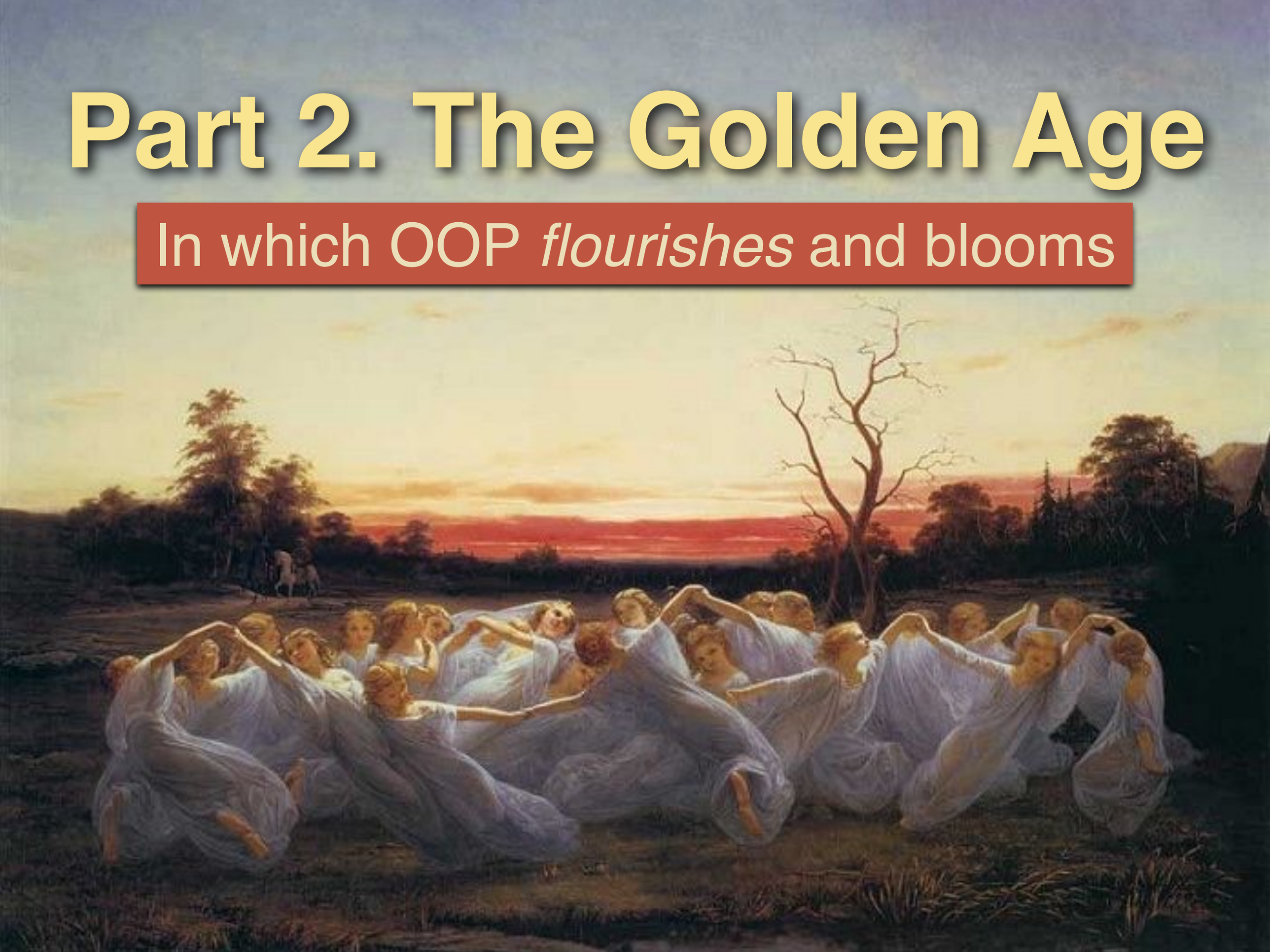
http://scg.unibe.ch/scgbib?query=Nier83b&display=abstract

Byte Magazine, August 1981.

https://archive.org/details/byte-magazine-1981-08

Part 2. The Golden Age

In which OOP *flourishes* and blooms



Ängsälvor (Swedish "Meadow Elves") by Nils Blommér (1850)

https://en.wikipedia.org/wiki/Elf

"*C with classes*" initially added classes and inheritance to C, just like Simula added them to Algol.

OOP is programming using inheritance.

Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance.

What is "Object-Oriented Programming?" ECOOP 1987, revised 1991

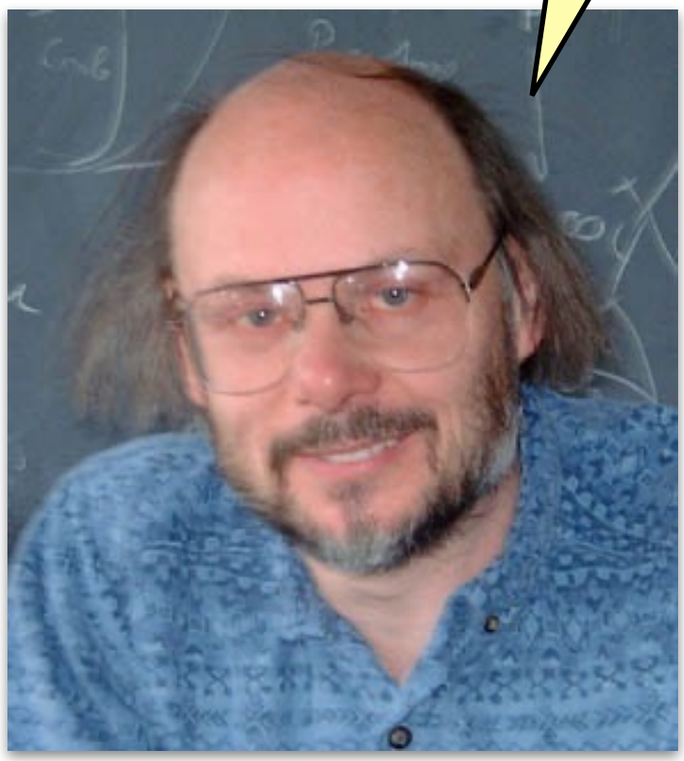
As legend goes, Bjarne Stroustrup, an experienced Simula programmer was tasked with developing some simulation programs while working at AT&T Labs. Not having a Simula compiler available, (and finding Simula too slow for his purposes), he decided to follow in the footsteps of Dahl and Nygaard and add object-oriented features to C, using C's macro facilities. "C with classes" gradually evolved into C++, a much more profound extension of C that fundamentally changed the way you program with the language.

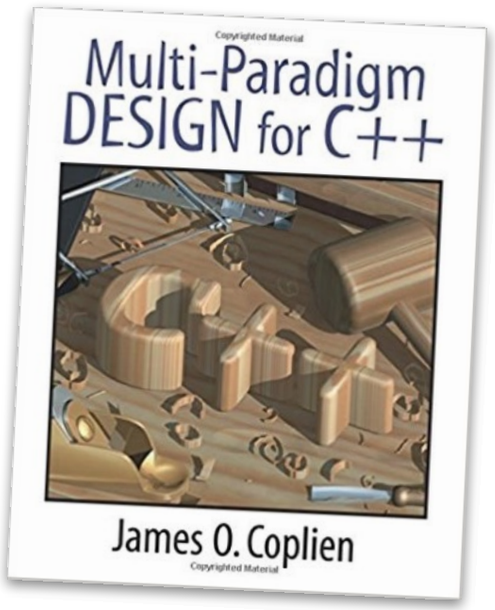
Stroustrup epitomized OOP as "programming with inheritance", that is, he saw sharing of features between classes as the most radical feature of OOP.

What is "Object-Oriented Programming?" ECOOP 1987, revised 1991

http://www.stroustrup.com/whatis.pdf

C++ is a *multi-paradigm* language





Gradually C++ evolved into more than just an extension of C to support simulation. Improvements in C++, such as a more robust type system, eventually led to changes in the C standard itself. Stroustrup did not see C++ as just an object-oriented extension of C, but rather as a “multi-paradigm” language that supported various programming styles. (This was epitomized in the 1998 book and 2000 PhD thesis by James Coplien.)

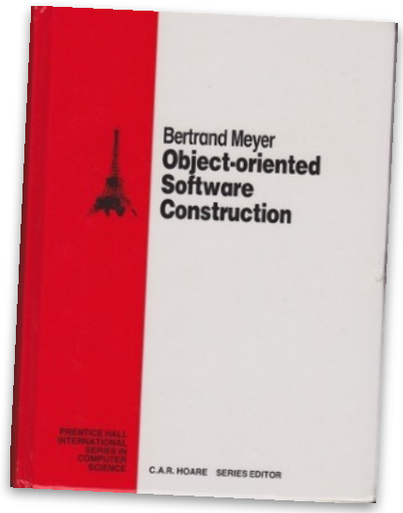
Presumably in response to criticisms of the complexity of C++, Stroustrup is quoted as saying “There are only two kinds of languages: the ones people complain about and the ones nobody uses.”

Multi-Paradigm Design, PhD thesis, James Coplien, 2000

http://tobeagile.com/wp-content/uploads/2011/12/CoplienThesis.pdf

Eiffel introduces “Design by Contract” as an OO language feature

OOSC is based on the objects manipulated rather than the functions performed

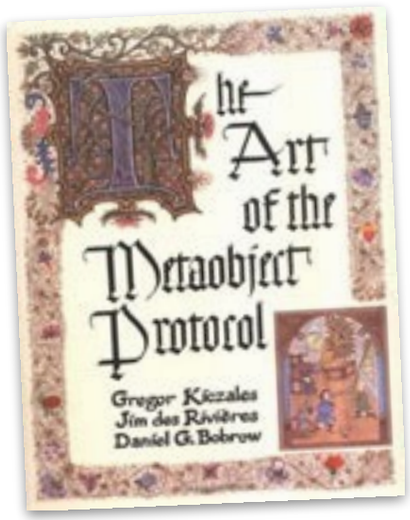


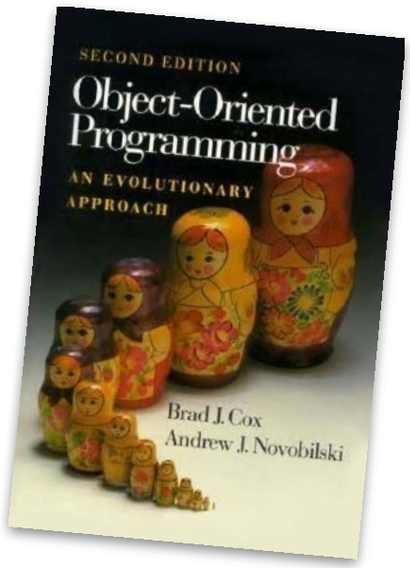
One of the most influential innovations in OOP was “Design by Contract”, which is both a methodology for designing classes that adhere to well-defined contracts for client/supplier relationships, as well as a set of programming language features to specify contracts as preconditions, postconditions and invariants in the code. While these features were originally introduced in Bertrand Meyer’s “Eiffel” language, variants have found their way into virtually every modern object-oriented language.

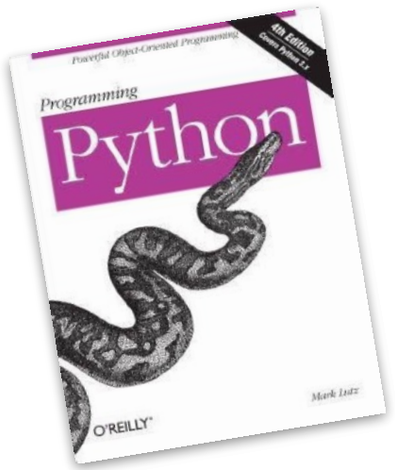
BM argued that OO design is fundamentally different since it focuses on the objects manipulated rather than the functions performed. As an application evolves, the function it performs may change, but the objects (domain concepts) tend to stay the same.

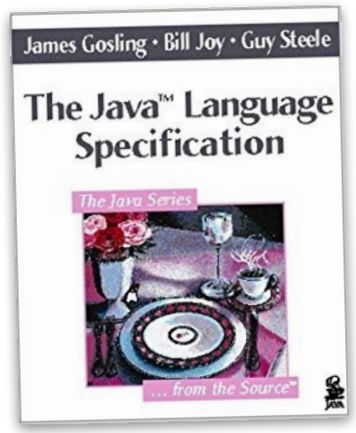
OOPLs proliferate

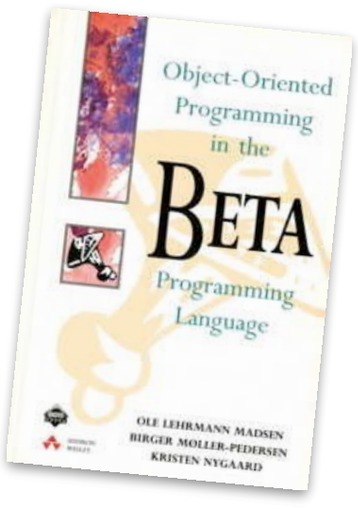
1980-…

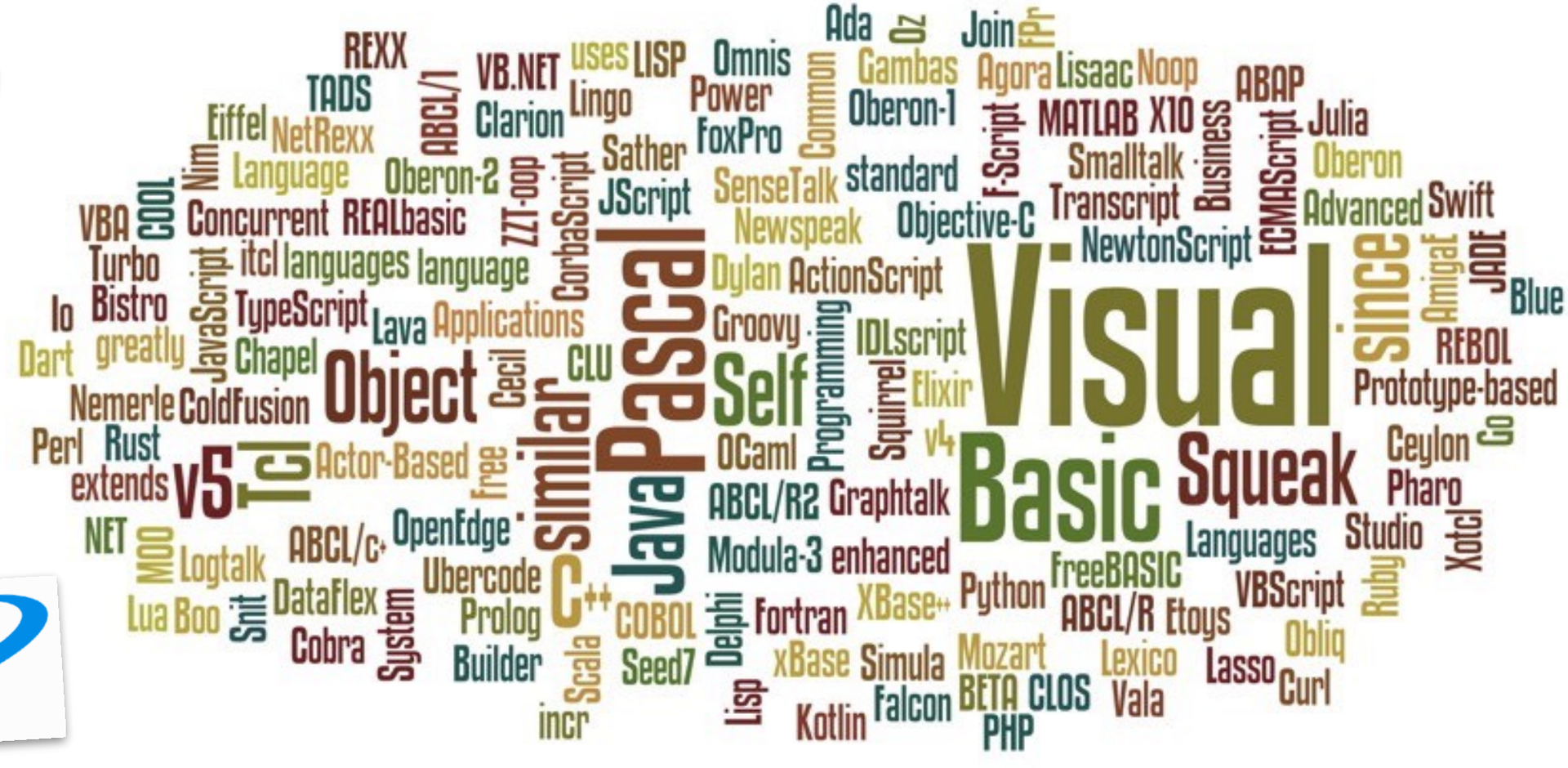












Dozens of new object-oriented languages were designed starting in the early 80s. Some, like CLOS and Objective C, added object-oriented features or layers to existing languages, while others were completely new. Python was conceived as OO scripting language. Beta reinvented Simula by reducing all language features to a single construct called a “pattern.” Self reinvented Smalltalk, replacing classes and inheritance by prototypes and delegation, leading to a much more dynamic language.

Dozens of research languages were also developed, particularly to experiment with different models of concurrency.

1987

So, what is “OOP” anyway?

Dimensions of Object-Based Language Design

Peter Wegner
Department of Computer Science
Brown University
Providence RI 02912

Abstract:

The design space of object-based languages is characterized in terms of objects, classes, inheritance, data abstraction, strong typing, concurrency, and persistence. Language classes (paradigms) associated with interesting subsets of these features are identified and language design issues for selected paradigms are examined. Orthogonal dimensions of object-based design space are related to non-orthogonal features of real languages. The influential application of object-oriented methodology to the development of object-based language paradigms is examined.

Delegation is defined as a generalization of inheritance and design alternatives such as semantics, multiple, and abstract inheritance are considered. Actors and prototypes are presented as examples of classless (delegation based) languages. Processes are classified by their degree of internal concurrency and distributed autonomy is discussed, suggesting that compromises between sharing and autonomy will be necessary in designing strongly typed object-oriented distributed database languages.

1. Design Space for Object-Based Languages

In order to examine design alternatives for object-based languages the following “dimensions” of language design are considered

objects
classes
inheritance
data abstraction
strong typing
concurrency
persistence

Those features span the design space of object-based languages. Each relates to a different aspect of computational behavior. Objects are autonomous entities that respond to messages or operations and share a state. Classes classify objects by their common operations. Inheritance serves to classify classes by their shared behavior. Data abstraction hides the representation of data and the implementation of operations. Strong typing imposes static constraints on the applicability of operations, both within and among objects. Concurrency allows objects to execute concurrently with other objects and to have internal concurrency. Persistence allows object identity to persist across applications and so be independent of values or keys used in object addressing.

Language classes worthy of special study are identified and the efficiency, simplicity, and methodology of the associated paradigms is examined. We first consider just objects, classes, and inheritance, then add data abstraction and strong typing, and finally consider concurrency and persistence. Along the way global properties of design dimensions such as consistency and orthogonality are introduced and related to non-orthogonal features that occur in real languages.

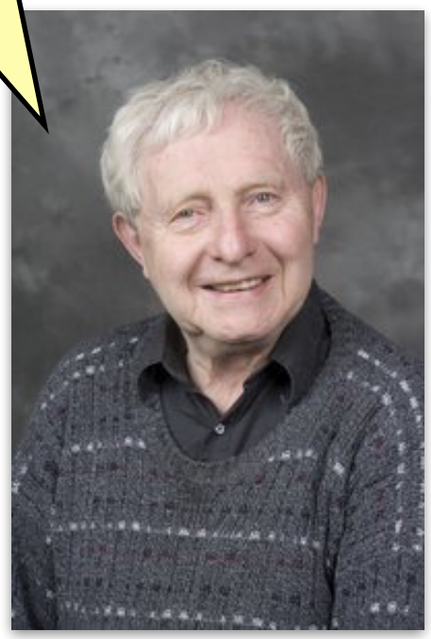
2. Objects, Classes, and Inheritance

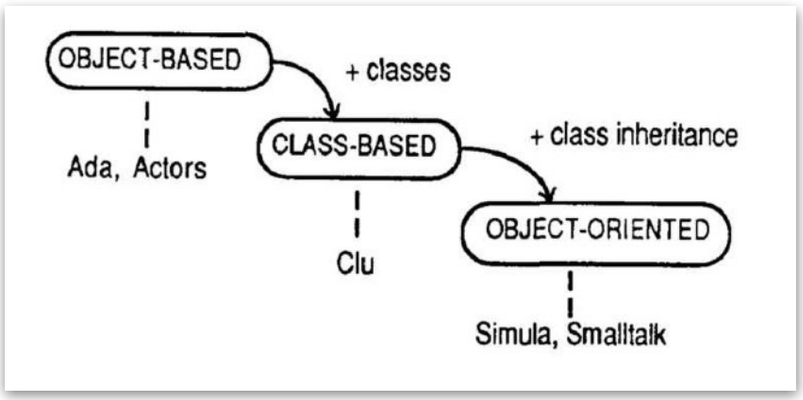
Objects have the following properties:

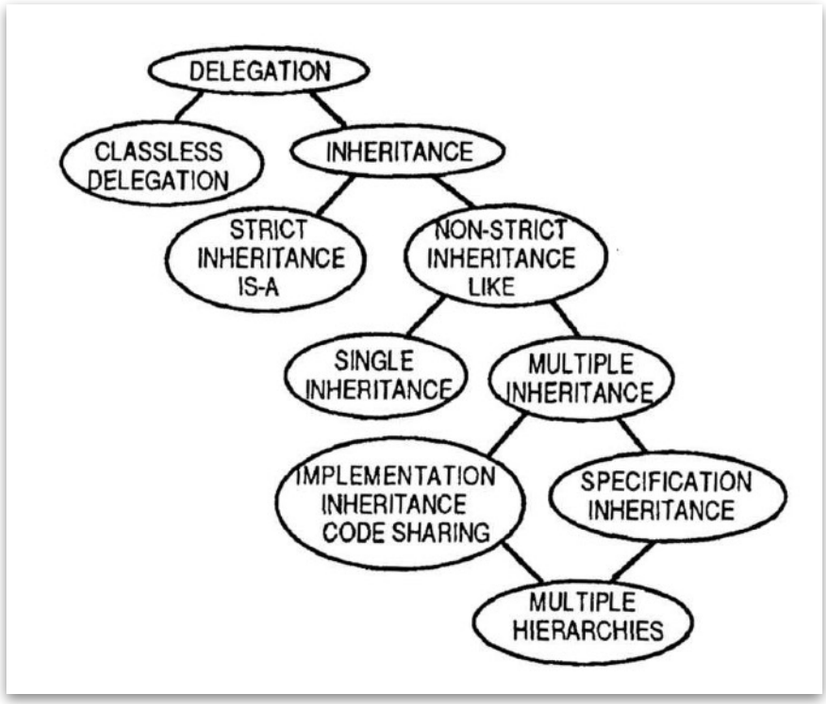
objects: An object has a set of “operations” and a “state” that remembers the effect of operations. Objects may be operated with functions, which have no memory. Function values are completely determined by their arguments, being genuinely the same for each invocation. In contrast, the value returned by an operation on an object may depend on its state as well as its arguments. An object may learn from experience, its reaction to an operation being determined by its invocation history.

The term “object-based language” may now be defined as follows:

OOP = Objects +
Classes + Inheritance







Given the increasing number of OO languages and the diverse interpretations OOP, Peter Wegner tackled the problem of trying to define OOP and classify OO languages. He drew a distinction between “object-based” languages, “class-based” ones, and fully “object-oriented” ones that support all three of objects, classes and inheritance. He also proposed a taxonomy of the different forms that inheritance found in OO languages.

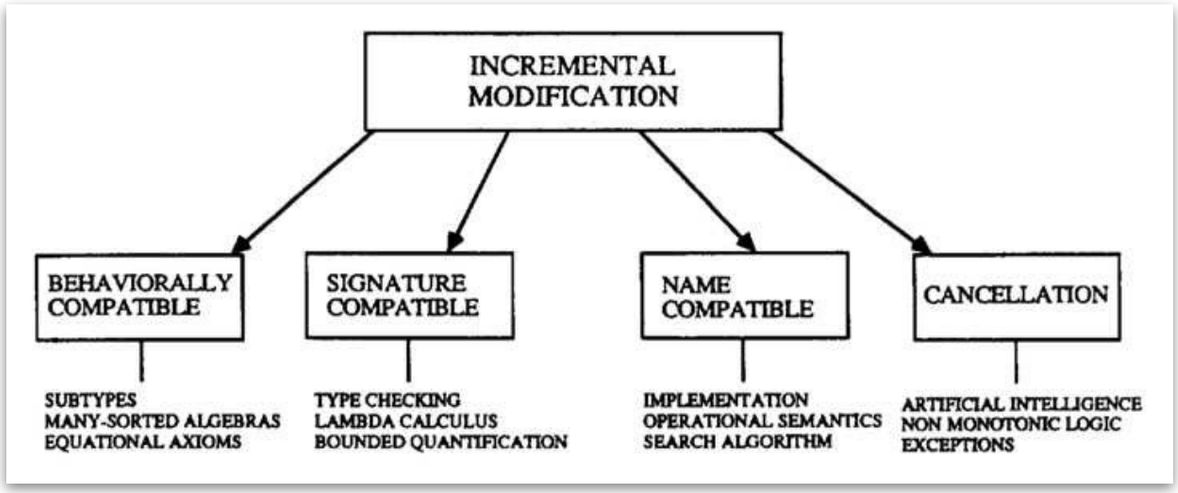
Dimensions of Object-Based Language Design, OOPSLA 1987

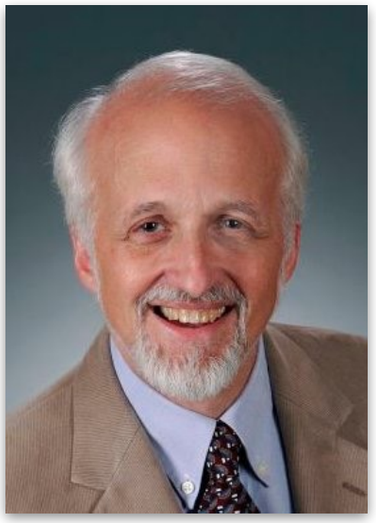
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.3742&rep=rep1&type=pdf

1988

Inheritance is an incremental modification mechanism


Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. ECOOP 1988





Principle of substitutability: An instance of a subtype can always be used in any context in which an instance of a supertype was expected.

In another influential paper, Peter Wegner and Stanley Zdonik surveyed the different forms of inheritance in OO languages and studied how they impact diverse notions of compatibility. Interestingly, they proposed a “principle of substitutability” several years before Barbara Liskov and Jeannette Wing formulated what is now known as the “Liskov substitution principle”.

Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. ECOOP 1988

https://www.researchgate.net/publication/221496346

There are **three views** of OOP:
the *Scandinavian* view, the *Mystical* view,
and the *Software Engineering* view:

Programming is *modeling*

Programming is *objects sending messages* to objects

Programming is *data abstraction + polymorphism + inheritance*

Ralph Johnson sees it like this:

> "I explain three views of OO programming. The *Scandinavian view* is that an OO system is one whose creators realise that *programming is modelling*. The *mystical view* is that an OO system is one that is built out of *objects that communicate by sending messages to each other*, and computation is the messages flying from object to object. The *software engineering view* is that an OO system is one that supports *data abstraction, polymorphism by late-binding of function calls, and inheritance*."

You are free to guess which programming languages are referred to here …

Attributed to Ralph Johnson in "The Myths of Object-Orientation", James Noble, ECOOP 2009

https://doi.org/10.1007/978-3-642-03013-0_29

OO Principles proliferate

1980s

Separate interface from implementation

Encapsulation, Abstraction and Information Hiding

Program to an *Interface*, not an *Implementation*

The open-closed *principle*

Single Responsibility Principle

Law of Demeter

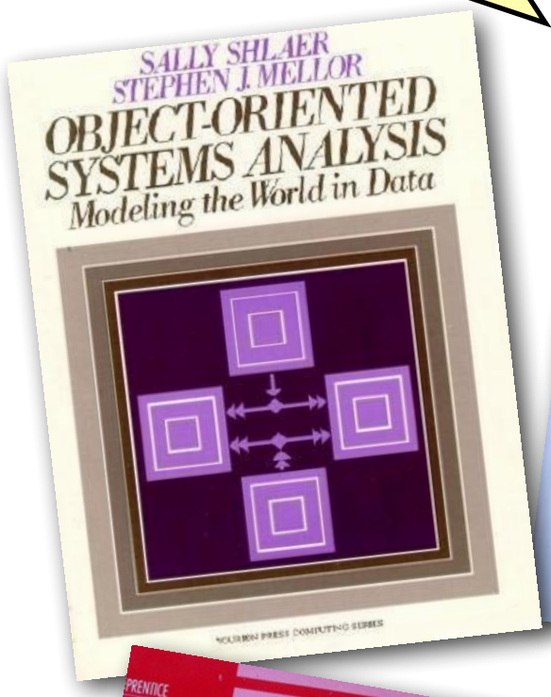
Many of these principles have been reformulated and repackaged over the years, but they all have their origins in the 80s.

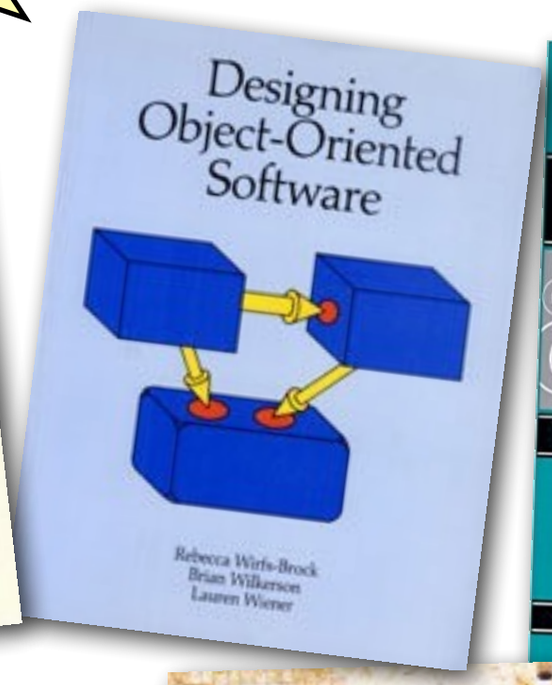
1988-1993

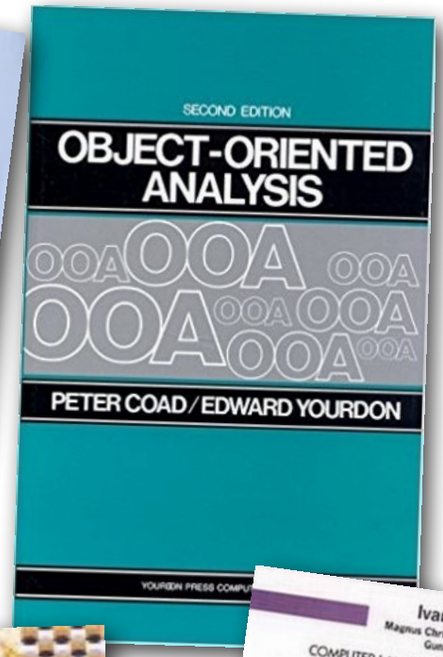
OO Methods proliferate

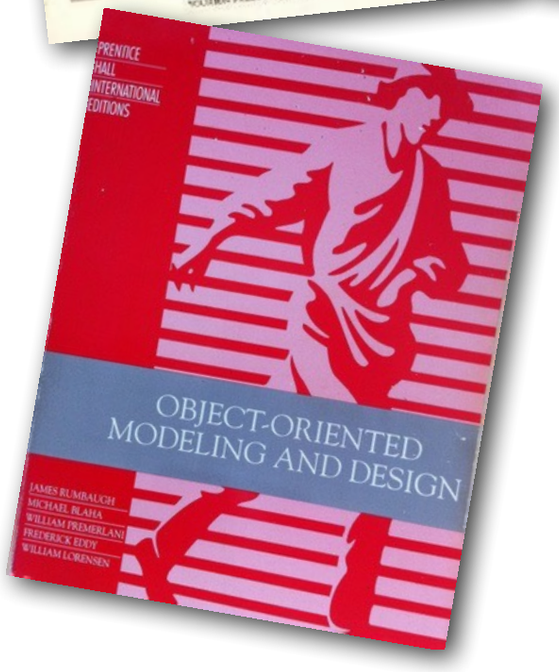
Modeling is programming

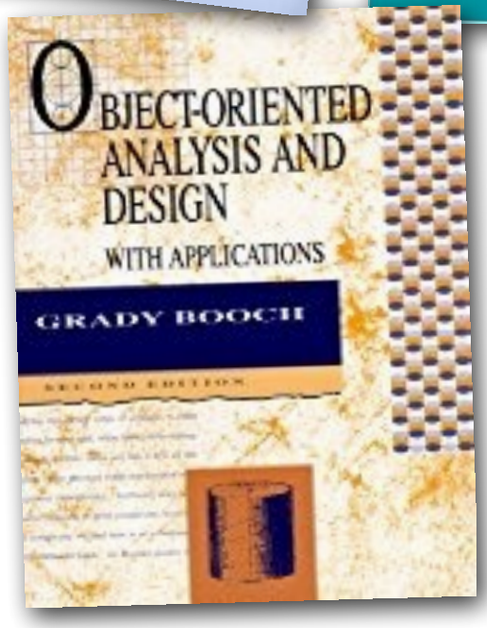
It's objects all the way down!

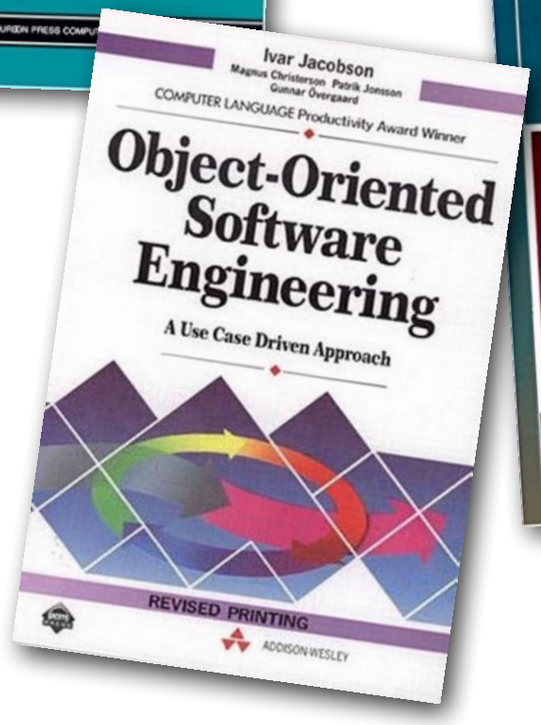










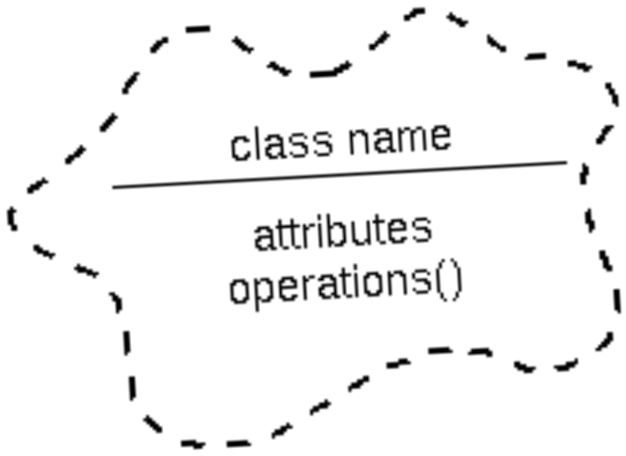


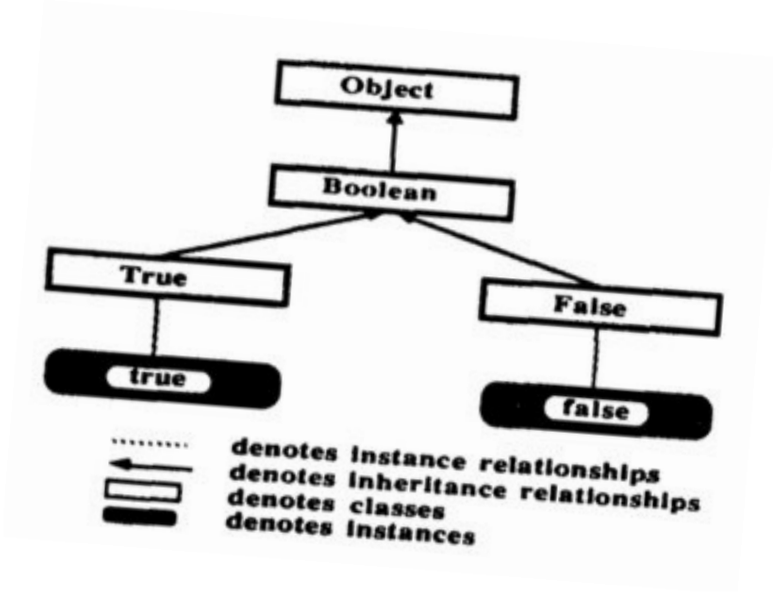
Starting in the late 80s, a number of very influential books attempted to crystallize methodologies for object-oriented analysis and design. Amongst the more famous of them, my favourite is “Designing Object-Oriented Software”, which lays down the principles of “responsibility-driven design.” I still use this in teaching today. My next favourite is “Object-Oriented Software Engineering” which explains the role of use cases in the OOSE process.

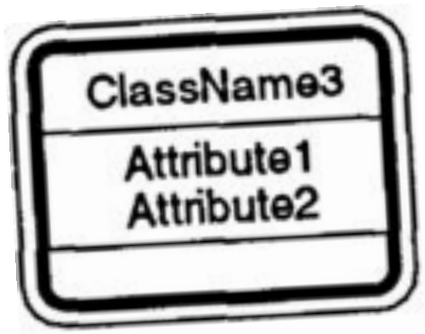
Taken as a whole, these books make clear that OO does not just mean programming, but that the act of *modeling* is fundamental to OOP. Furthermore, they send the message that it is “objects all the way down,” not in the sense that Alan Kay meant, but in the SE process from domain modeling and requirements specification down to implementation.

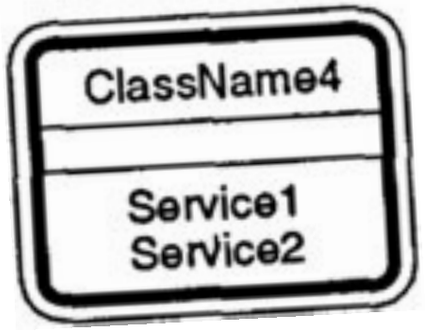
Object Oriented Systems Analysis: Modeling the World in Data, 1988; Designing Object-Oriented Software, 1990; Object-Oriented Modeling and Design, 1991; Object-Oriented Analysis and Design, 1991; Object Oriented Analysis, 1991; Object-Oriented Software Engineering: A Use Case Driven Approach, 1992; Object-Oriented Development: The Fusion Method, 1993

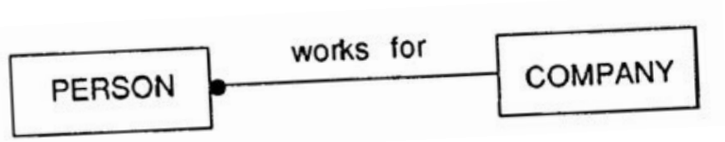
OO Diagrams proliferate

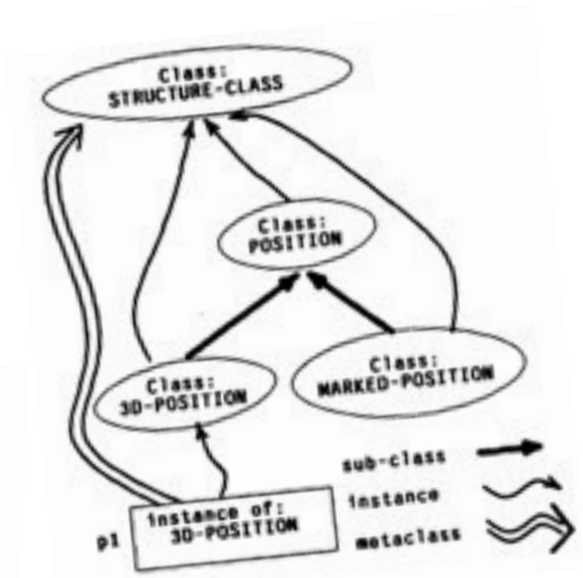












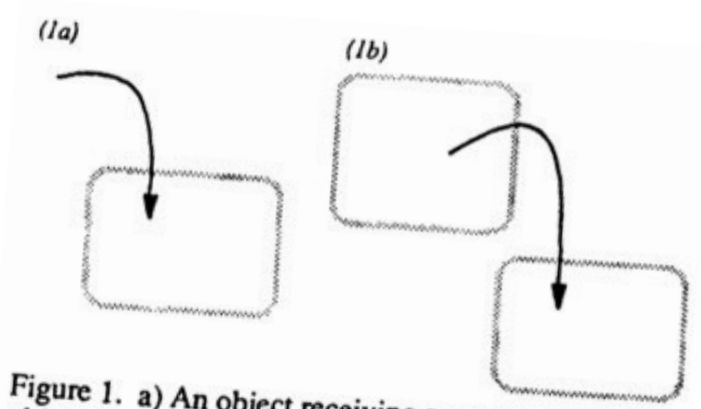
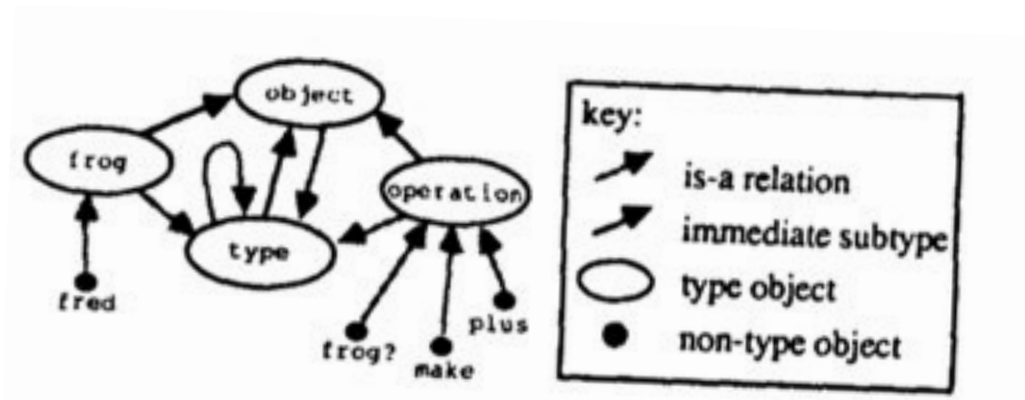


Figure 1. a) An object receiving a message. b) An object sending a message to another object.



In the mid to late 80s, dozens of different ways to represent class diagrams were invented. Classes were drawn as boxes, ellipses, clouds and even hexagons, with arrows going in all sorts of directions. By some counts, there were over 100 different styles of diagrams defined by 1992.

Bertrand Meyer said that at the time he was puzzled why people were so fascinated by diagrams when OO languages themselves worked perfectly well as modeling languages. (I.e., programming is modeling.)

One day when he was in the shower it hit him: “Bubbles and arrows don’t crash!”

[Figures are mostly drawn from various OOPSLA 1986 papers.]

Part 3. Rebellion

In which *battle lines* are drawn

late 80s, early 90s

Objects are not enough!

(Inheritance is not enough)

Programming is specializing *frameworks*

Programming is configuring *components*

Programming is instantiating *design patterns*

In the early days of OOP, there was a great deal of hype about how objects and especially inheritance would simplify development through reuse. Quickly people discovered that this was not so simple, and they started to look for more. Already in the mid to late 80s the idea of an “application framework” started to emerge.

Norman Meyrowitz (OOPSLA 86) and later Erich Gamma (OOPSLA 88) were among the first to show how this could be realized.

Then in the late 80s the idea of “software components” started to take hold. Although it was never clear exactly what a “component” was, everyone agreed that components had interfaces that could be plugged in to clients, without necessarily depending on inheritance.

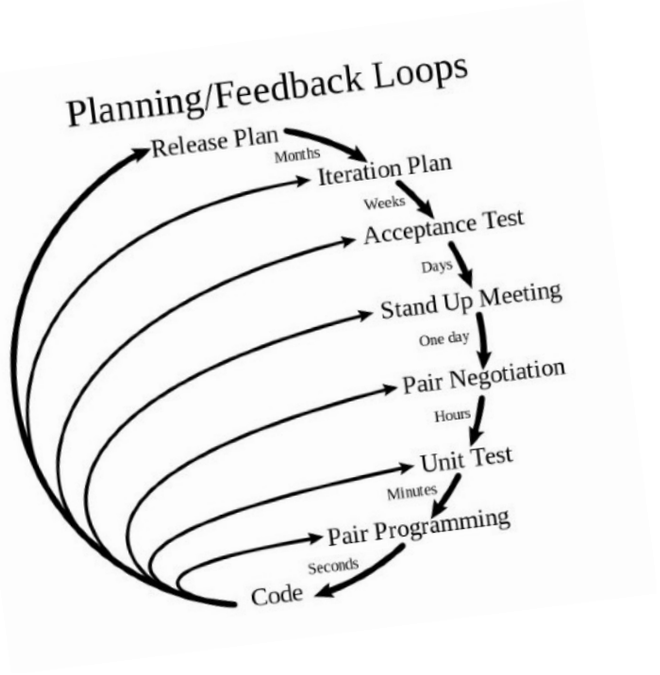
The design pattern community started to grow around this time, and emerged from these same ideas, as the first patterns nicely expressed the key ideas behind frameworks and components.

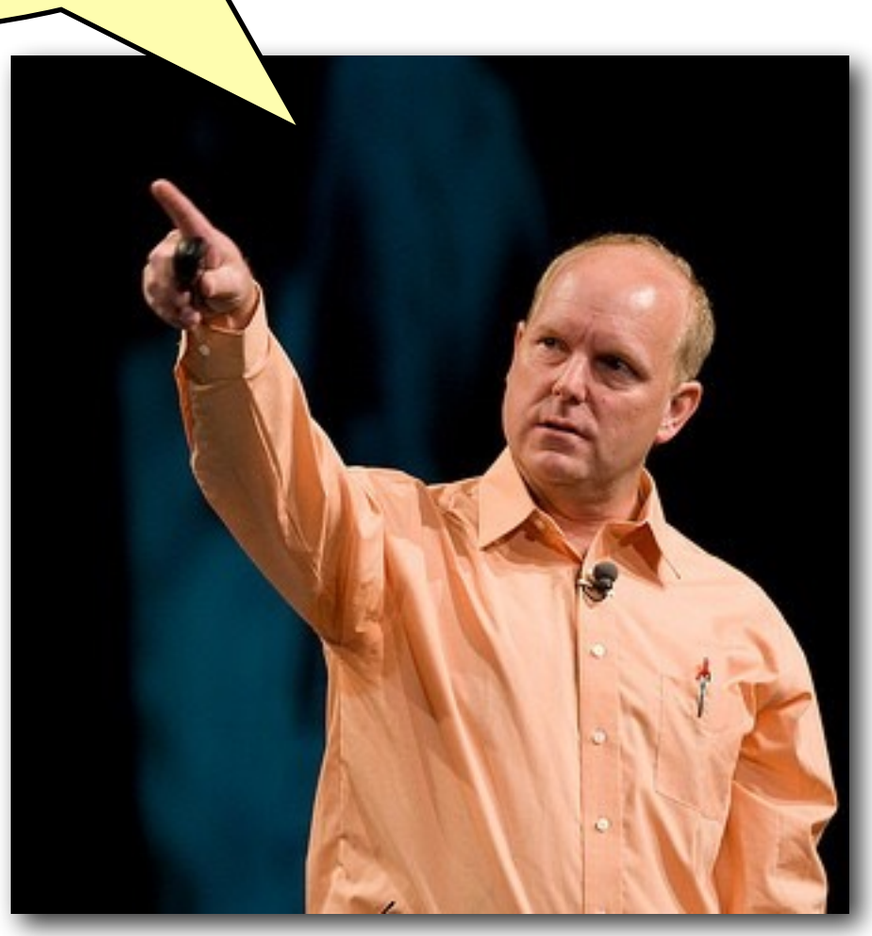
https://www.quora.com/How-important-are-design-patterns-in-software-development

You need to be “agile”

1999

Programming is testing, refactoring and pair programming





Kent Beck and others argued that we should pay more attention to the software practices in place. They identified a number of best practices that, they argued, would make software development more responsive to stakeholder needs. From unit testing to scrum, these practices have had a huge influence over the past twenty years on how object-oriented software is developed.

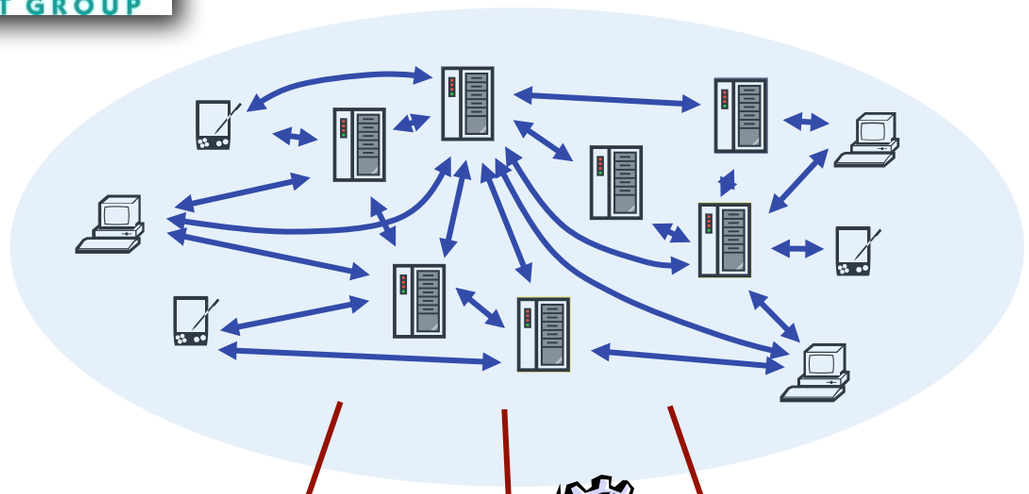
Model transformation takes off

2001

Modeling is programming

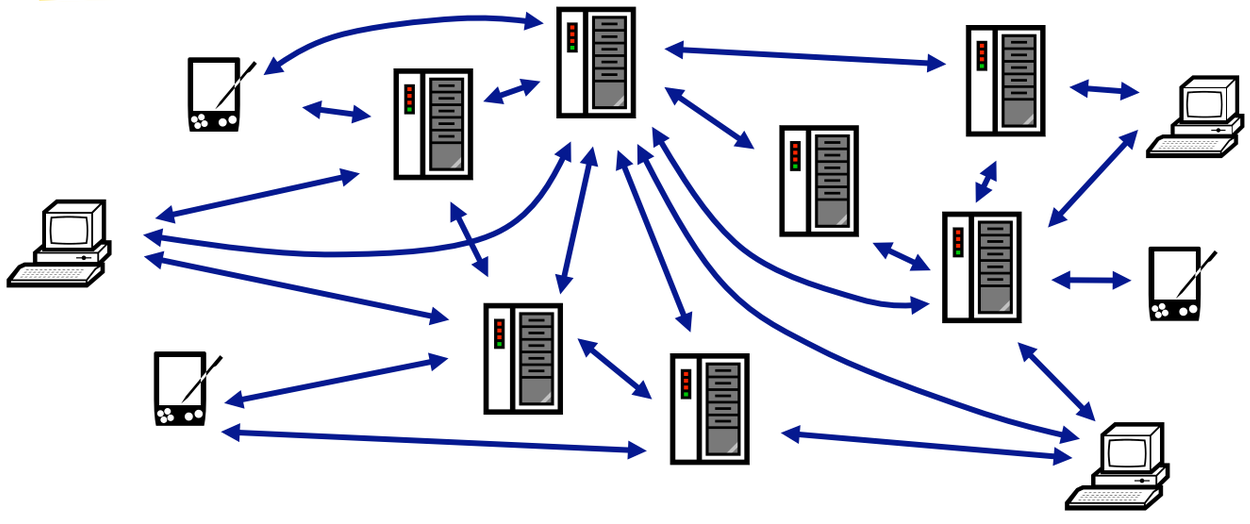
OBJECT MANAGEMENT GROUP

software developer



Platform Independent Model

automatic translation



UML, the Unified Modeling Language, was developed at Rational Software by the “three amigos” (Booch, Rumbaugh and Jacobsen) partly in response to the profileration of OO diagrams (but also as a way to market its own tools and views on “round-trip engineering”). UML was handed over to the Object Management Group for standardization.

The focus on UML as a modeling tool led to the idea that models could be transformed (or compiled) to running systems. (Actually an old idea followed by CASE tools in the 1980s.) In essence, the proponents of model-driven engineering were saying not that programming is modeling but that “modeling is programming”.

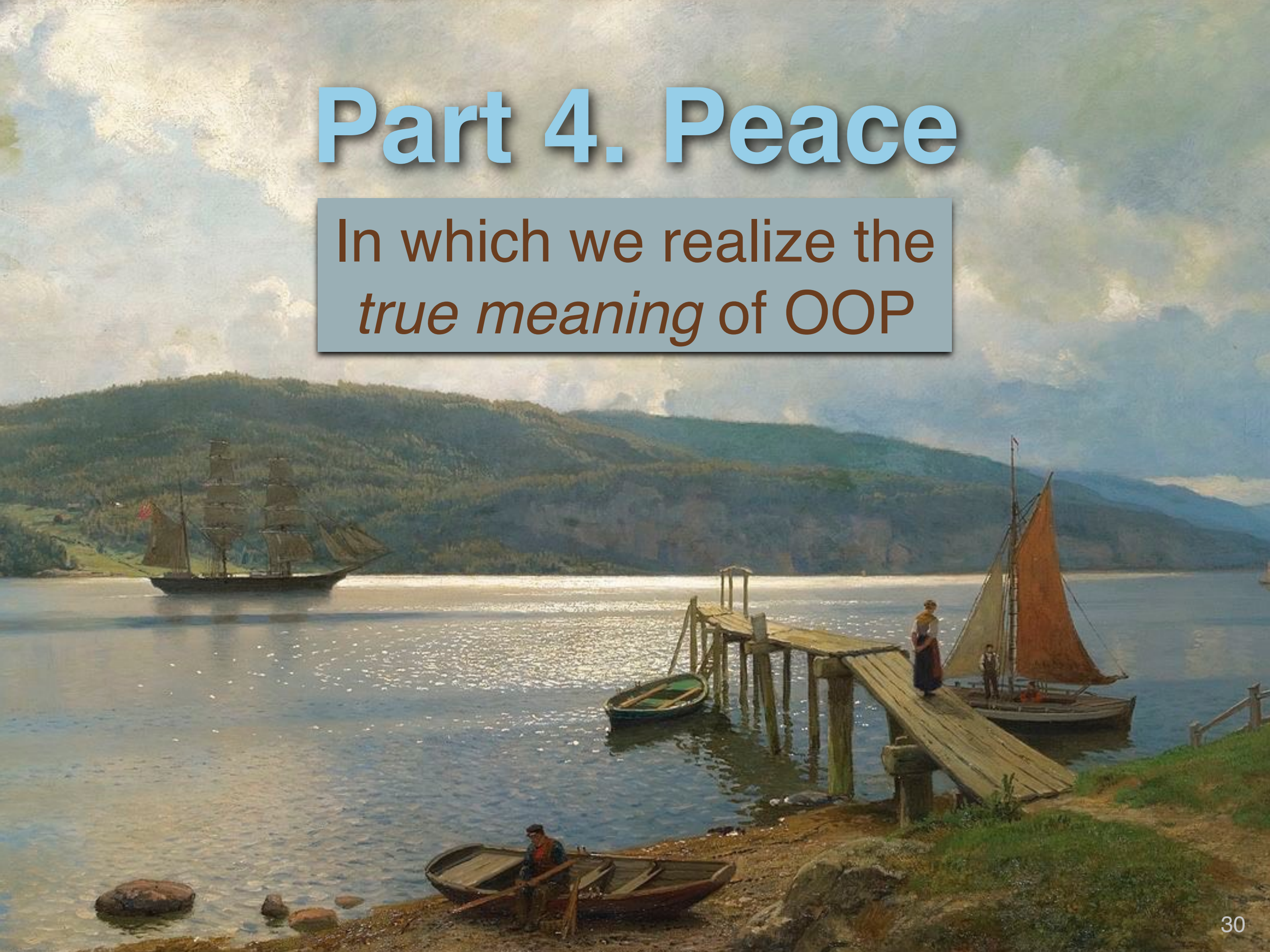
Programs must be statically type-checked!

We don’t need no stinkin’ types!

While all this was happening, new OOPLs and variants were being developed. Richer and more expressive type systems were being developed for statically typed languages at the same time as new dynamically typed languages were being invented and reinvented. Although the jury is still out on which approach allows programmers to be more productive, a lot of research is devoted to type inference for dynamically typed languages, whether it be for compiler optimization or to support program understanding.

Johannes Flintoe, Egill Skallagrímsson engaging in holmgang with Berg-Önundr

https://en.wikipedia.org/wiki/File:Johannes-flintoe-egil-skallarimsson.jpg



Part 4. Peace

In which we realize the *true meaning* of OOP

The jetty at Feste near Moss - Hans Gude - Kaien på Feste i nær Moss (1898)

https://commons.wikimedia.org/wiki/File:Hans_Gude_-_Kaien_på_Feste_i_nær_Moss_(1898).jpg

API = Metamodel = DSL

Configuration = Model = Script

If we step back and consider what all the different camps are trying to achieve, I would argue that the differences are more cosmetic than profound. At the “framework” level, an API or a metamodel or a language are really the same thing. An internal domain specific language is just a “fluent API”, and a meta model defines the language of its models.

At the instance level we speak of configurations of components, or platform specific models, or scripts.

What is remarkable about object-oriented programming is that it is so good at helping you define the framework level.

Programming is modeling

The lesson I draw from this is that object-oriented programming (and programming in general) is indeed modeling. Object-oriented languages are especially good at this because they allow you to define your own meta-model in terms of the classes of your system, their interfaces, and the relationships between them, while this is not the focus (or strength) of other programming paradigms.

Programming is understanding

But why do we care about modeling? I would say that the ability to model domain concepts in the code of object-oriented software systems helps us as software developers understand better the impact of changes in both the real world and in the code. In other words, as Kristen Nygaard put it: “Programming is understanding.”

Programming is understanding