Smalltalk

A Complete 8-Chapter History of the Birthplace of Object-Oriented Programming

Topics covered:

Simula 67 & the state of programming before objects

Alan Kay, the Dynabook, and Xerox PARC

Everything is an object, everything is a message

The live image & class browser · MVC & the GUI

The lineage through Objective-C and Ruby

Why Smalltalk lost the commercial market · Squeak & Pharo today

Format: A4 · Humanities-style, reflection questions throughout

A completeness/history pick — closer in spirit to the site's own History of AI courses

Table of Contents

CHAPTER 1 OF 8

Before Smalltalk — Programming Before Objects

Smalltalk

Chapter 1 · Before Smalltalk — Programming Before Objects

Object-oriented programming has a real origin point — a specific place, a specific research group, and a specific philosophy that most programmers today inherited without ever meeting the source directly. Before tracing that origin, it's worth spending one chapter on what programming looked like *before* objects existed as an idea at all — because the gap is easy to underestimate once you've never known anything else.

Programming in the Early 1970s — No Objects, Just Procedures and Data

The dominant model of the era — FORTRAN, COBOL, early BASIC, and the direct ancestors of C — built programs from three ingredients: variables holding data, procedures that operated on that data, and control flow tying it all together. Data and the code that operated on it were strictly separate, unrelated things. A customer record was just a data structure sitting somewhere in memory; the code that validated it, printed it, or updated it lived entirely elsewhere, with no formal, enforced connection between the two at all.

This is genuinely difficult to appreciate today, because almost no working programmer has ever *not* known objects. What's actually missing without them: no bundling of data together with the operations that belong to it, no enforced boundary around how a piece of data is allowed to be touched, no natural way to even say "a Customer knows how to validate itself" — that idea has no home in this model at all. Data is data; behavior is separate code that happens to be pointed at it.

Simula 67 — The Direct Precursor

Developed in Norway by Ole-Johan Dahl and Kristen Nygaard, Simula was originally built for exactly what its name says — *simula*ting real-world systems: queues, processes, resources being shared and contended for. Representing each entity in a simulation as a self-contained unit, carrying both its own data *and* its own behavior, turned out to be a genuinely natural fit for that specific domain.

Simula 67 introduced, for the first time in a real, working language, actual class-based objects, inheritance, and something recognizably close to virtual method dispatch. Said plainly, because it's often skipped over: Simula 67, not Smalltalk, was technically first to a working class-based object system.

SO WHY IS SMALLTALK REMEMBERED AS "THE BIRTHPLACE OF OOP"?

Simula's objects were a feature added to serve simulation modeling — genuinely useful, but built for one specific purpose. What comes next is a completely different starting motivation, one that would turn objects from a modeling convenience into an entire, ground-up philosophy of how computing itself should work — and where the actual term "object-oriented programming" would be coined for the first time. That's Chapter 2.

The Context at Xerox PARC's Learning Research Group

Xerox PARC — the Palo Alto Research Center, founded in 1970 — became an extraordinary concentration of researchers imagining what personal computing could eventually become, years before "personal computer" was anything close to a mainstream idea. Within PARC, the Learning Research Group, led by Alan Kay, was tasked with something distinctly different from business data processing or scientific simulation: exploring computing as a tool for learning and creativity.

That's a genuinely different starting motivation from Simula's own simulation-modeling roots — and it's exactly why what came out of Kay's group would end up philosophically different from Simula, even while borrowing Simula's own class concept directly as raw material. Kay wasn't trying to model queues and processes. He was trying to build an entirely new kind of computing environment — previewed in full next chapter.

WORTH SITTING WITH BEFORE CHAPTER 2

Every object-oriented language covered elsewhere on this site — Java, C#, C++, Kotlin, Ruby — inherited its object model from a lineage that runs through what's about to be described. None of them invented objects independently; all of them are, in a very real sense, still working in Kay's shadow.

Reflection Questions

QUESTION 1

This chapter describes pre-object programming as having "no formal, enforced connection" between a customer record and the code that validates it. Using a language you already know, describe concretely what problems that lack of connection could cause in a real program.

QUESTION 2

Simula 67 technically introduced class-based objects before Smalltalk existed. Why do you think "object-oriented programming" as a named philosophy is still associated with Smalltalk rather than Simula? What does that suggest about the difference between inventing a feature and inventing a philosophy?

QUESTION 3

Simula's objects grew out of a need to simulate real-world systems. The Learning Research Group's own motivation was learning and creativity, not simulation. Before reading Chapter 2, speculate: what kind of computing environment might that different motivation actually produce?

CHAPTER 1 KEY TAKEAWAYS

- Pre-object programming kept data and the behavior operating on it strictly, formally separate
- **Simula 67** (Dahl & Nygaard, Norway) technically introduced the first working class-based object system, built for simulation modeling
- The term "object-oriented programming" was not coined by Simula's own creators
- Xerox PARC's Learning Research Group, led by Alan Kay, was motivated by learning and creativity — a genuinely different starting point from simulation
- Every OOP language on this site traces its object model back through this lineage

CHAPTER 2 OF 8

Alan Kay & the Vision at Xerox PARC

Smalltalk

Chapter 2 · Alan Kay & the Vision at Xerox PARC

`smalltalk1-1` closed on a question: what kind of computing environment does a motivation of "learning and creativity," rather than simulation, actually produce? This chapter is the answer — and it starts somewhere unexpected for a programming language: biology.

Alan Kay's Background — Biology, Not Just Computer Science

Kay's own influences reached well beyond computer science. A documented, genuine influence was biology — specifically, the image of a cell as a self-contained unit, enclosed by its own membrane, communicating with other cells only by passing signals across that boundary, never by reaching directly inside another cell to manipulate its internals. That's not a loose metaphor tacked on after the fact — it's close to the literal conceptual seed of "objects sending messages to each other," as opposed to "programs calling functions that reach into shared data."

Two other influences matter here too: Ivan Sutherland's Sketchpad, an early interactive graphical program that showed computing could be direct and visual rather than batch-processed; and Seymour Papert's work with LOGO, exploring how children learn through direct, hands-on manipulation rather than abstract instruction.

The Dynabook — A Personal Computer for Children

Around 1972, Kay proposed the **Dynabook**: a portable, personal computer meant to be usable by children of all ages, for creative expression — not business computation. At a time when computers were still enormous, expensive, institutionally-shared machines, the idea that a single person — let alone a child — might have their own personal computer was genuinely far ahead of the hardware that actually existed.

The Dynabook was never built as physical hardware in that era — the hardware to truly realize it wouldn't exist for another decade or more. But it set the *design goal* Smalltalk itself was built to serve: the software environment a Dynabook would need, something a child could pick up, explore, and build things in directly, with no real gap between using the computer and programming it.

Smalltalk Built Specifically to Realize That Vision

This is the structural difference from Simula, worth stating plainly: Simula's objects were a feature serving simulation modeling. Smalltalk's objects were the foundation of an entire environment, designed from the ground up to be approachable, explorable, and directly buildable by a child. Every later chapter in this course — the live "image" (`smalltalk1-4`), the class browser as a live-editing tool — exists specifically because of this goal: a live, inspectable, directly-manipulable system, not a batch-compiled black box.

Worth being honest here too: "Smalltalk" was not one single design delivered fully formed. It went through several real iterations at PARC over most of a decade — Smalltalk-71, Smalltalk-72, Smalltalk-76 — culminating in Smalltalk-80, the version whose public documentation is what made the language widely known and historically influential.

> **NOT A SOLO PROJECT**
> Kay's own group at PARC included other significant, credited contributors — Dan Ingalls and Adele Goldberg most notably — whose own work was central to actually building and documenting what Smalltalk became. Treating Smalltalk as a single "great man" invention flattens a genuinely collaborative research effort into a simpler story than what actually happened.

Coining "Object-Oriented Programming"

Alan Kay is credited with coining the actual term "object-oriented programming," specifically to describe what Smalltalk was doing. Worth being honest about a real, documented tension: Kay has since said the term went on to become associated with things he didn't originally mean by it — a gap between his own original intent and how most working programmers understand "OOP" today.

> **A THREAD THIS COURSE RETURNS TO**
> This gap between Kay's original intent and today's common understanding of "OOP" is not a footnote — it's a genuine throughline this course comes back to directly, especially in `smalltalk1-3` (the full "everything is a message" philosophy Kay actually meant) and `smalltalk1-7` (why the language embodying that original, fuller vision lost out commercially to languages that adopted a narrower slice of it).

Why This Context Matters for Everything That Follows

`smalltalk1-1` closed by noting that every OOP language on this site inherited something from this lineage. This chapter sharpens that point: most of them inherited a narrower slice of Kay's original vision specifically — classes and inheritance as a code-organization tool — rather than the full, radical "everything is a message,

sent to a live, directly-manipulable object" philosophy Smalltalk was actually built around. `smalltalk1-3` goes fully into that fuller philosophy next.

Reflection Questions

QUESTION 1

Kay drew on a biological metaphor — cells communicating across a membrane rather than reaching into each other's internals. In a language you already know, where does this idea show up (or fail to show up) as an enforced rule, rather than just a convention programmers are expected to follow voluntarily?

QUESTION 2

The Dynabook was never built in Kay's own era, yet it still shaped Smalltalk's entire design. What does this suggest about the relationship between a concrete, even unrealized, design goal and the actual technical decisions a language ends up making?

QUESTION 3

This chapter claims most modern OOP languages inherited "a narrower slice" of Kay's vision. Before reading Chapter 3, speculate: what part of that vision do you think got left behind, and why might that specific part have been the hardest to carry into languages built for more conventional, file-based, compiled workflows?

CHAPTER 2 KEY TAKEAWAYS

- Kay's own influences included biology (cells communicating via messages, not internal access), Sketchpad, and Papert's work with children and LOGO
- The **Dynabook** — a personal computer for children, proposed c. 1972 — was never built physically but set Smalltalk's own design goal
- Smalltalk's objects were the foundation of an entire environment, not a feature bolted onto simulation modeling (unlike Simula 67)
- Smalltalk iterated substantially (71, 72, 76) before Smalltalk-80 made it widely known
- Dan Ingalls and Adele Goldberg were significant, credited co-creators — not a solo Kay project
- Kay coined "object-oriented programming" — and has since said the term outgrew his own original meaning

CHAPTER 3 OF 8

Everything Is an Object, Everything Is a Message

Smalltalk

Chapter 3 · Everything Is an Object, Everything Is a Message

`smalltalk1-2` left a question open: what part of Kay's original vision got left behind by most of Smalltalk's own descendants? This chapter is the full answer — the single idea most languages softened, split, or abandoned entirely: literally everything in Smalltalk is an object, and the only thing you can ever do to an object is send it a message.

The Core Claim — Literally Everything Is an Object

Every value in Smalltalk is an object — numbers, booleans, characters, strings, arrays, and even classes themselves (which are instances of their own "metaclass"). There is no primitive-versus-object split anywhere in the language — no `int` distinct from `Integer`, no primitive `boolean` sitting outside the object system the way Java famously keeps them separate (`java1-2`'s own primitive-vs-reference-type material), or the way C#'s own value-type structs (`csharp1-2`) are a genuine, deliberate middle ground but still not full objects receiving messages the Smalltalk way.

Concretely: `3 + 4` in Smalltalk is not "the `+` operator applied to two primitive integers." It is literally sending the message `+` to the object `3`, with the argument `4`. The number `3` is a real object that receives a message and responds with a result — another object, `7`.

Even Control Flow Is Message Sends

Here's the genuinely radical detail. `ifTrue:ifFalse:` is not built-in syntax the way `if` / `else` is in virtually every other language covered on this site — Java, C#, C++, Kotlin, Python, all of them. It's an ordinary message, sent to a Boolean object, carrying two block arguments (Smalltalk's own closures) representing "what to do if true" and "what to do if false." `True` and `False` are actually two separate classes, each independently implementing `ifTrue:ifFalse:` to do the obvious thing for its own case.

```
x > 5
  ifTrue: ['big']
  ifFalse: ['small']
```

This has an astonishing structural consequence: because `ifTrue:ifFalse:` is just an ordinary method on a class, nothing in principle stops a third boolean-like class from implementing its own version — something

structurally impossible in any language where `if` is baked directly into the compiler or interpreter's fixed grammar. Loops work identically: `whileTrue:` is a message sent to a block, not built-in loop syntax either.

Why This Matters — Uniformity as a Design Principle

The payoff of this radical uniformity: there is exactly one mechanism in the entire language — sending a message — and it applies everywhere, with no special cases. Compare this to virtually every other language on this site, each of which has at least two tiers: "the stuff that's really built into the language's own grammar" (`if`/`else`, loops, operators on primitives) and "the stuff that's just regular code" (method calls, user-defined types). Smalltalk collapses this into one tier, entirely.

Direct Contrast — Dispatch in Other Languages on This Site

LANGUAGE	DISPATCH MODEL	PRIMITIVES ARE OBJECTS?
Java (java1-5)	Every instance method virtual by default	No — int/boolean sit outside the object system
C# (csharp1-5)	Explicit virtual/override — opt-in dispatch	No — real value-type structs, not messages
C++ (cpp1-7)	Compiler-built vtable for virtual classes	No — primitives are raw machine types
Kotlin (kotlin1-4)	JVM-inherited class/primitive split	No — same split as Java underneath
Smalltalk	Live message lookup at send time, no exceptions	Yes — every value, no exceptions

The throughline across every one of those languages: each treats "the fixed set of things the language itself natively understands" as separate from "user-extensible object behavior." Smalltalk has no such separation anywhere — even arithmetic and control flow live entirely inside the same, single, uniform mechanism as every user-defined class's own methods.

A Worked Example — A Conditional as Pure Message Sends

Walking through `x > 5 ifTrue: ['big'] ifFalse: ['small']` one message at a time: first, `x > 5` sends the message `>` to the object `x`, with argument `5` — this returns a Boolean object, either `true` or `false`. That Boolean object then receives the message `ifTrue:ifFalse:`, carrying the two block arguments. The Boolean itself — not some special interpreter-level "if" logic — decides which block to actually evaluate and return, purely through its own class's own method implementation.

UNIFORMITY HAS A REAL COST TOO

This radical uniformity isn't a free lunch, and `smalltalk1-7` covers the honest consequences in full. Every single operation — even `3 + 4` — technically involves a full message dispatch and method lookup, which is genuinely

slower at a naive implementation level than a compiler emitting one raw machine-code ADD instruction for two primitive integers, the way C, C++, and Java's own primitive arithmetic works. Uniform elegance and raw performance are a real tradeoff here, not a solved problem.

A DIRECT ROOT OF SOMETHING YOU MAY ALREADY KNOW

This exact "everything is an object" model is the direct ancestor of Ruby's own famous claim to the same idea — `ruby1-2`'s own material on Ruby's object-identity focus traces straight back to what's described in this chapter. `smalltalk1-6` covers that lineage in full.

Reflection Questions

QUESTION 1

In a language you already know, try to imagine defining a third boolean-like value with its own custom "if" behavior. What specifically stops you from doing this in that language, that Smalltalk's own model doesn't stop you from doing?

QUESTION 2

This chapter argues that treating control flow as message sends is philosophically consistent but has a real performance cost. Do you think that tradeoff was a reasonable one for Kay's own Dynabook vision (smalltalk1-2) specifically? Why or why not?

QUESTION 3

Every language in this chapter's own compare-table keeps a "built into the grammar" tier separate from a "regular code" tier. Can you think of a real practical benefit that separation provides, that Smalltalk's own fully uniform model gives up?

CHAPTER 3 KEY TAKEAWAYS

- Every value in Smalltalk is an object — no primitive/object split anywhere, unlike Java, C#, C++, or Kotlin
- `3 + 4` is literally sending the message `+` to the object `3`
- `ifTrue:ifFalse:` and `whileTrue:` are ordinary messages, not built-in syntax — True and False are separate classes each implementing their own version

- Smalltalk has exactly one mechanism (message sending) with no special cases; every other language on this site has at least two tiers
- This uniformity carries a genuine performance cost — every operation is a full dispatch, not a raw machine instruction
- This is the direct ancestor of Ruby's own "everything is an object" claim (smalltalk1-6)

CHAPTER 4 OF 8

The Image — A Live, Persistent System

Smalltalk

Chapter 4 · The Image — A Live, Persistent System

`smalltalk1-3` covered the philosophy — everything is an object, everything is a message. This chapter covers the environment that philosophy actually lives inside, and it's just as unfamiliar to a modern programmer as the message-passing model itself: the Smalltalk "image."

Source Files Are Not the Program — The Image Is

In virtually every language on this site, "the program" is fundamentally a set of source files, compiled or interpreted fresh each run, producing a new process from scratch every single time — `c1-1`'s own compile-then-link model, `rust1-1`'s own `cargo build`, `py1-1`'s own script execution. The running process's own state disappears the instant it exits; the next run starts over from source, with nothing carried forward.

Smalltalk inverts this entirely. The "image" is a literal, saveable snapshot of an entire live system's objects and state — every class, every object instance, every open window, potentially even a half-finished thought mid-edit — all preserved together in one file. You don't "run the Smalltalk program" the way you run a compiled C binary. You load the image, and you're dropped directly back into the exact live system state you left it in.

What This Actually Means in Practice

There is no meaningful "compile, then run" distinction the way `c1-1` / `rust1-1` established. Changes made to a class apply to the live, running system immediately, while it's executing, with nothing restarted. Objects already running continue running — and, worth naming honestly, can genuinely continue using the *old* version of a method until something tells them to pick up the new one, meaning a class's own definition and an instance's own currently-running behavior can actually diverge mid-session. Powerful, but a real, documented source of confusion.

This is exactly why Smalltalk programmers didn't typically edit text files in an external editor the way C, Rust, or Python programmers do — the natural way to work *is* inside the live image itself.

The Class Browser — Live Editing, Not File Editing

The class browser is Smalltalk's own primary development tool. Rather than opening a `.java` or `.rs` file in a text editor, a Smalltalk programmer navigates a live, structured view of the actual class hierarchy currently loaded in the running image — browsing packages, classes, and individual methods directly as live objects, editing one method at a time, taking effect immediately upon being accepted. No separate save-then-compile-then-run steps exist at all.

This is a genuinely different mental model from `code1`'s own VS Code Essentials course — however powerful VS Code's own live debugging and hot-reload features are, they still fundamentally operate on files sitting in a folder on disk, edited by a general-purpose text editor. Smalltalk's class browser only makes sense because there are no files in the conventional sense to begin with — the "source" is a live structure inside the image itself.

Contrasted With the Edit-Compile-Run Cycle

	STATE BETWEEN RUNS	EDITING MODEL
C (c1-1)	None — compiled binary, exits, gone	External text editor, separate compile step
Rust (rust1-1)	None — cargo build/run, exits, gone	External text editor, separate build step
Python (py1-1)	None per script run, even with a REPL	External text editor or a REPL, both file-adjacent
Smalltalk	Fully preserved — the image itself is the state	Live class browser, editing the running system directly

Worth being precise here: a Python REPL (`py1-1`) and a Smalltalk image are *not* the same thing, even though both let you type expressions and see live results. A Python REPL session's own state disappears the moment the terminal closes, with no built-in mechanism to save "the whole running interpreter state" back to disk and reload it exactly as it was. A Smalltalk image genuinely can do exactly that — as a first-class, designed feature of the system, not an ad-hoc workaround.

Why This Serves Kay's Original Vision

This ties directly back to `smalltalk1-2`'s own Dynabook material: a live, directly-manipulable, always-running environment is exactly what a child exploring and building things needs — no gap between poking at the system to see what happens and actually programming it. The image *is* the environment being explored, not a separate artifact produced by a separate build process standing between the person and the system.

A REAL, GENUINE FRICTION — PREVIEWING CHAPTER 7

The image-based model creates real friction with things every other course on this site takes for granted. `git1`'s own file-based diff/commit model has no natural analog for "diff this live object graph." Something as simple as sharing code with someone else becomes genuinely harder when "the code" is entangled with a live, potentially

large, stateful snapshot rather than a clean set of text files. This is a direct, honest preview of `smalltalk1-7`'s own account of why Smalltalk lost out commercially.

A DISTANT, MUCH NARROWER ECHO TODAY

The general idea of "kept state between executions" echoes faintly in things like a Jupyter notebook's own kernel preserving variables between cell runs — but even that is a far more limited, narrower echo than a full Smalltalk image, which preserves the entire live system, not just a handful of variables in one notebook session.

Reflection Questions

QUESTION 1

This chapter describes a scenario where a running object can keep using an old method version after the class itself has been edited. Is this closer to a feature or a bug, in your own view? What would you want to know before deciding?

QUESTION 2

Using a language you already know, describe what "losing your work" typically means when a program crashes. Would that same kind of loss even be possible in a properly-used Smalltalk image? Why or why not?

QUESTION 3

The chapter argues the image model fits Kay's Dynabook vision well but creates real friction with version control. Can you imagine a modern tool or workflow that tries to get some of the image's own benefits without giving up file-based version control entirely? How complete a solution do you think it would actually be?

CHAPTER 4 KEY TAKEAWAYS

- A Smalltalk **image** is a saveable, live snapshot of an entire running system's objects and state — not source files
- No compile-then-run cycle — edits apply to the live, running system immediately
- Running objects can genuinely diverge from a freshly-edited class definition mid-session
- The **class browser** edits the live system directly — there are no conventional files to open
- Unlike a Python REPL, an image's entire state is designed to be saved and reloaded exactly as left

- The image model directly serves Kay's Dynabook vision, but creates real, honest friction with version control (previewed here, covered in full in smalltalk1-7)

CHAPTER 5 OF 8

Smalltalk's Real Legacy — MVC and the GUI

Smalltalk

Chapter 5 · Smalltalk's Real Legacy — MVC and the GUI

`smalltalk1-4` covered the image and the class browser — genuinely unfamiliar ideas to most programmers today. This chapter covers the opposite kind of legacy: two ideas from the exact same era and place that became so completely normal, they're now almost invisible.

Inventing MVC — Model-View-Controller

Trygve Reenskaug, working within the Smalltalk-80 environment at Xerox PARC as a visiting researcher, designed the Model-View-Controller pattern to solve a real, concrete problem: how should a live, directly-manipulable Smalltalk object (`smalltalk1-4`'s own image model) be displayed on screen *and* respond to user input, without tangling the object's own actual data and logic together with the code drawing it or handling clicks?

Model — the actual data, logic, and state, a live Smalltalk object. **View** — how that model gets displayed on screen. **Controller** — handles user input and translates it into changes on the Model. Worth dwelling on given `smalltalk1-3`'s own uniform-object philosophy: even this separation is achieved entirely through Smalltalk's own ordinary object and message mechanism — not a special framework bolted on top of the language, but the same single mechanism doing everything else in the system too.

MVC's Afterlife — Still Named Everywhere

`react1`, `angular1`, `vue1`, and `svelte1` all still use "MVC" or a direct descendant term — MVVM, or "component-based" architectures that still separate state, view, and logic — as inherited vocabulary. Most web developers using these terms today have never heard of Reenskaug or Smalltalk-80 at all, working with a fifty-year-old pattern's own name without knowing where it came from.

Worth being precise rather than sweeping: modern frontend frameworks aren't literally implementing Reenskaug's original Smalltalk-80 MVC exactly as designed — `django1`'s, `rails1`'s, and `laravel1`'s own MVC framing are structurally closer to the original server-side pattern than most client-side JavaScript frameworks are. But the underlying *conceptual* split — separate the data, the display, and the input-handling — is the direct, genuinely traceable inheritance, regardless of how loosely or tightly any given modern framework maps onto Reenskaug's own original three-part division.

Xerox PARC and the Graphical User Interface Itself

Beyond MVC as a software pattern, PARC's own research — much of it happening in the same building, same era, and same Learning Research Group culture that produced Smalltalk — pioneered the graphical user interface itself: the mouse, overlapping windows, icons, menus. The entire visual vocabulary of "using a computer" that's now so completely normal it's effectively invisible.

> **THE 1979 DEMO, TOLD CAREFULLY**
>
> In 1979, Apple engineers — including Steve Jobs — were given access to see PARC's own GUI work, as part of a real, documented licensing arrangement. This directly and demonstrably shaped the Lisa and later the Macintosh's own graphical interface — a genuine, well-documented historical fact, not an urban legend. But it's also worth being precise rather than repeating the flattened pop-history version of this story: Apple's own engineers made real, substantial contributions and innovations well beyond what they saw at PARC. This is a story of real influence and real licensed access, not simple theft.

Why the GUI and Smalltalk Are the Same Story

This ties directly back to `smalltalk1-2`'s own Dynabook vision: a computer meant to be directly, visually manipulable by a child needs a visual interface, not a command line. The GUI wasn't a separate PARC project that happened to exist alongside Smalltalk — it was a genuinely necessary part of realizing the same underlying vision, and Smalltalk's own live image (`smalltalk1-4`) was frequently the actual environment being displayed and manipulated through these new GUI concepts in the first place.

MVC PART	REENSKAUG'S ORIGINAL ROLE	ROUGH MODERN EQUIVALENT
Model	A live Smalltalk object holding real data/logic	Application state, a database record, a component's own data
View	How the object is rendered on screen	A rendered template, a React component's own JSX output
Controller	Translates user input into Model changes	Event handlers, a server-side controller (django1, rails1, laravel1)

> **TWO RELATED BUT SEPARATE THREADS**
>
> This chapter traced the design/UI lineage — MVC and the GUI itself. `smalltalk1-6` traces a genuinely separate thread: the direct programming-language lineage, through Objective-C and Ruby. Both threads flow from the same PARC-era work, but they're worth keeping distinct rather than blurring together.

Reflection Questions

QUESTION 1

Pick a modern web framework you're familiar with (or one covered elsewhere on this site). Where do you see Reenskaug's original Model/View/Controller split showing up, even loosely? Where does it break down or not map cleanly?

QUESTION 2

This chapter is deliberately careful to avoid the "Steve Jobs stole the GUI" pop-history version of the Xerox PARC story. Why do you think the more accurate, nuanced version matters, beyond just being more factually correct?

QUESTION 3

The chapter argues the GUI and Smalltalk are "the same story" rather than two coincidentally related projects. Do you find that framing convincing? What evidence in this chapter supports it, and what might complicate it?

CHAPTER 5 KEY TAKEAWAYS

- Trygve Reenskaug designed MVC inside the Smalltalk-80 environment to separate data, display, and input-handling
- MVC itself is implemented using Smalltalk's own ordinary object/message mechanism — no special framework
- react1/angular1/vue1/svelte1 and django1/rails1/laravel1 all inherit MVC's own vocabulary, loosely or tightly
- Xerox PARC pioneered the mouse, windows, icons, and menus — the GUI's entire basic visual vocabulary
- The 1979 Apple/PARC connection is real and documented, but the "theft" framing oversimplifies real, licensed access and real Apple innovation
- The GUI and Smalltalk share the same underlying Dynabook motivation from smalltalk1-2, not a coincidence

CHAPTER 6 OF 8

The Direct Lineage — Objective-C, Ruby, and Beyond

Smalltalk

Chapter 6 · The Direct Lineage — Objective-C, Ruby, and Beyond

`smalltalk1-5` traced Smalltalk's design and UI influence — MVC, the GUI. This chapter traces a genuinely separate thread: languages that borrowed Smalltalk's own actual syntax and semantics directly, not just its ideas about screens and windows.

Objective-C — Smalltalk's Syntax, Bolted Onto C

Brad Cox and Tom Love created Objective-C in the early 1980s by adding Smalltalk-style messaging directly on top of C — literally grafting Smalltalk's own message-send syntax onto C's existing structure. `[receiver message]` and `[receiver message: argument]` are direct syntactic descendants of Smalltalk's own keyword-message form.

This is a real, direct lineage, tied concretely back to `smalltalk1-3`'s own core claim: in Objective-C, `[myArray count]` is genuinely, literally sending the message `count` to the object `myArray` — the same message-passing model as Smalltalk, syntactically dressed to sit inside C's own surrounding structure. Not "OOP inspired by OOP in general" — the actual message-send concept survived the trip into Objective-C intact, not just classes and inheritance.

Objective-C went on to become the primary language for NeXTSTEP and, for decades, macOS and iOS development, right up until Swift. For a genuinely long stretch, Smalltalk's own message-passing model was running on hundreds of millions of Apple devices — mostly invisibly, to developers who may never have heard the word "Smalltalk" at all.

Ruby — Smalltalk's Philosophy, With Friendlier Syntax

Yukihiro "Matz" Matsumoto, Ruby's creator, has explicitly and repeatedly cited Smalltalk as one of Ruby's own major influences — a stated, acknowledged influence, not speculation. Ruby's own "everything is an object" claim, already flagged in `smalltalk1-3`'s own note-box, is inherited close to directly: `3.times { ... }` is a real method call on the integer object `3`, not special syntax, and `nil` itself is an object of class `NilClass`, not a special non-object null value the way many other languages treat null.

Worth revisiting `ruby1-6`'s own material directly: Ruby's own **open classes** — the ability to reopen and add methods to an existing class, even a built-in one like `Integer` or `String`, at any time — are a genuinely direct descendant of the same live-editability philosophy covered in `smalltalk1-4`'s own image and class-browser material. The idea that a class definition isn't a sealed, fixed artifact, but something you can keep live-editing at any point, is inherited close to intact.

Beyond Objective-C and Ruby — A Wider, Fainter Trail

Briefly, and without overclaiming: many other languages absorbed some Smalltalk influence more indirectly or partially. Python's own object model is less directly Smalltalk-influenced than Ruby's, but still object-uniform in spirit for many built-in types. JavaScript's own prototype-based object model is worth flagging as a genuinely *separate* lineage — despite a superficial "everything is dynamic and objecty" resemblance, it did not descend from Smalltalk directly, and lumping every dynamically-typed, object-heavy language together as "basically Smalltalk" would be inaccurate.

Self and Squeak are worth naming here too, as more direct — if less mainstream — descendants; `smalltalk1-8` covers Squeak specifically in full as this course's own closing chapter.

How Much of "Modern OOP" Genuinely Traces Back Here

A CORRECTION WORTH MAKING HONESTLY

`smalltalk1-1`'s own closing note claimed that every OOP language covered on this site "inherited its object model from a lineage that runs through" Kay's work. That's worth refining here, not repeating uncritically: there are actually **two** distinct OOP lineages. A **Simula-rooted** branch (C++, Java, C#) — emphasizing static types and compile-time class structure, tracing back to Simula 67's own earlier, independent root from `smalltalk1-1` — and a **Smalltalk-rooted** branch (Objective-C, Ruby, Python to a degree) — emphasizing dynamic typing, live message-passing, and duck typing. Most OOP languages inherit from one of these two roots, sometimes elements of both, rather than everything tracing to Smalltalk specifically.

BRANCH	LANGUAGES	TYPING	DISPATCH
Simula-rooted	C++, Java, C#	Static, compile-time	Compiler-resolved (vtables, opt-in virtual)
Smalltalk-rooted	Objective-C, Ruby, Python (partially)	Dynamic	Live message lookup at send time

A GENUINELY SEPARATE LINEAGE: JAVASCRIPT

JavaScript's own prototype-based object model didn't descend from Smalltalk — it's a distinct design with its own separate history. It's easy to lump every "dynamic and objecty" language together as one big Smalltalk family; this

chapter deliberately resists that, since accuracy about which influence actually flowed where matters more than a tidy, oversimplified story.

Reflection Questions

QUESTION 1

This chapter distinguishes "borrowing syntax/semantics directly" (Objective-C) from "borrowing UI/design ideas" (smalltalk1-5's own MVC material). Why might that distinction matter when tracing a language's real influences, rather than just saying "everything comes from Smalltalk"?

QUESTION 2

If you know Ruby, Python, or JavaScript, pick one and identify a specific feature that either does or doesn't fit this chapter's own Smalltalk-rooted vs. Simula-rooted distinction. Where would you place it, and why?

QUESTION 3

This chapter openly corrects an overclaim made back in smalltalk1-1. Why do you think it's worth the course revising its own earlier claim explicitly, rather than quietly using the more accurate framing going forward without mentioning the earlier one?

CHAPTER 6 KEY TAKEAWAYS

- Objective-C's `[receiver message]` syntax is a direct descendant of Smalltalk's own message-send form — same underlying concept, not just similar-looking OOP
- Objective-C carried Smalltalk's message-passing model onto hundreds of millions of Apple devices for decades
- Matz has explicitly cited Smalltalk as a major Ruby influence — Ruby's "everything is an object" and open classes both trace directly back
- JavaScript's prototype-based model is a genuinely separate lineage, not a Smalltalk descendant
- OOP genuinely splits into two lineages: Simula-rooted (C++, Java, C#, static/compiler-dispatched) and Smalltalk-rooted (Objective-C, Ruby, dynamic/message-dispatched)
- Not all OOP traces to Smalltalk specifically — a deliberate correction to smalltalk1-1's own earlier, looser claim

CHAPTER 7 OF 8

Why Smalltalk Didn't Win

Smalltalk

Chapter 7 · Why Smalltalk Didn't Win

Six chapters have made a consistent case: Smalltalk was first, philosophically coherent, and directly responsible for ideas — message passing, MVC, the live environment — that shaped decades of software that came after it. So the obvious question, asked directly and honestly in this chapter: why isn't it the dominant language today? Why do languages that borrowed only part of its ideas win out commercially, while the original lost mainstream relevance?

Commercial and Licensing Factors

Smalltalk-80 was commercialized through a specific, restrictive early licensing model — Xerox licensed it to a small number of vendors, including ParcPlace Systems, itself spun out specifically to commercialize Smalltalk. This happened at a time when competing languages, C and later C++, were comparatively cheap or free and widely available across virtually every platform. Cost and availability mattered enormously to who actually adopted a language at scale — a real, concrete, entirely non-philosophical factor.

Smalltalk environments were also historically demanding of hardware resources — the full live image, the GUI environment, all running together — at a time when hardware itself was expensive and limited. A real, practical adoption barrier, separate from any of the language's own philosophical merits.

C++ and Java — OOP Bolted Onto Familiar Syntax

C++ (Bjarne Stroustrup, early 1980s) took a genuinely different strategy: rather than building OOP as a from-scratch, radical new environment, it added class-based OOP features directly onto C. Existing C programmers, and the enormous existing base of C code, tooling, and compilers, could adopt OOP incrementally — without learning an entirely new paradigm, environment, and syntax all at once. This lowered the adoption barrier dramatically compared to Smalltalk's own "learn a completely new environment" cost.

Java (1995) went further in some ways — garbage collection, a more disciplined object model — but still kept C-family syntax: curly braces, semicolons, familiar control flow, explicitly designed to feel approachable to the enormous existing population of C/C++ programmers, per `java1-1`'s own material on that exact syntax choice. Sun's own marketing, its "write once, run anywhere" platform strategy, and a genuinely different,

more open licensing and distribution approach than Xerox's own Smalltalk model all mattered enormously here too — a real business-strategy contrast, not just a technical one.

The throughline: familiarity and incremental adoption cost, not technical or philosophical superiority, is what won. A language that let programmers keep 90% of what they already knew while gaining OOP incrementally beat a language that asked them to learn an entirely new environment and philosophy from scratch — even though the from-scratch language was arguably more coherent and consistent.

The Image-Based Model's Real Friction

This directly delivers on the honest preview from `smalltalk1-4`: version control. `git1`'s own file-based diff/commit model — and every version control system before Git, CVS and Subversion included — fundamentally assumes the artifact under version control is a set of discrete, diffable text files. A live Smalltalk image doesn't decompose into that shape naturally. Tools were built over the years to extract text-based representations of image state specifically to work around this — change sets, "fileOut" mechanisms — but these were always somewhat bolted-on workarounds, unlike a C, Java, or Rust codebase, which simply *is* files from the start.

This mattered enormously as software development matured into a genuinely team-based, collaborative discipline through the 1990s and 2000s. The entire ecosystem of code review, diffing, CI/CD (`pipelines1`'s own material), and distributed team collaboration assumes files-on-disk as the fundamental unit — and Smalltalk's own image-first model was structurally a worse fit for that emerging ecosystem, regardless of its own philosophical elegance.

Deployment and Distribution

A related, concrete point: shipping a compiled C, C++, or Java program to a customer is straightforward — a binary, or a JAR, self-contained and relatively predictable. Shipping "a snapshot of an entire live image" as a deployable product was a genuinely harder, less standard problem historically — the image contains far more than just the application itself, including development tools, debugging infrastructure, and potentially unrelated experimental code left in the live system. Extracting a clean, minimal, deployable subset was its own real engineering problem Smalltalk vendors had to solve — one C and Java's own compile-to-a-single-artifact model never faced in the first place.

An Honest Synthesis — Not a Story of Technical Failure

Worth restating directly, since a chapter like this can easily imply otherwise: none of the above means Smalltalk was technically inferior, or that its ideas were wrong. `smalltalk1-3` through `smalltalk1-6`

already demonstrated the opposite — its ideas were coherent, genuinely influential, and in many ways more elegant than what replaced it commercially.

THE ACTUAL LESSON

This is a story about adoption economics, tooling-ecosystem fit, and timing — not a story about a "better" language beating a "worse" one. It's a very similar honest lesson to what this site's own `c3` and `cpp3` advanced courses already established when contrasting C's trust-the-programmer model against Rust's compiler-enforced guarantees: technical merit and commercial success are frequently not the same axis at all, and conflating the two is a real, recurring mistake worth naming explicitly rather than glossing over.

IT DIDN'T DISAPPEAR

Despite losing the commercial mainstream, Smalltalk didn't vanish. It has real, actively-maintained modern descendants — Squeak and Pharo — which is exactly where this course closes next, in `smalltalk1-8`.

Reflection Questions

QUESTION 1

This chapter argues C++ and Java won largely through familiarity and incremental adoption cost, not technical superiority. Can you think of a more recent example of a technology winning for similar non-technical reasons?

QUESTION 2

The chapter identifies version control and deployment as two concrete, practical frictions — not abstract philosophical weaknesses. Why might it matter, when evaluating why a technology lost out, to distinguish concrete practical frictions from abstract design critiques?

QUESTION 3

This chapter draws a direct parallel to the site's own C-vs-Rust honest lesson about trust vs. compiler enforcement. Do you find that parallel convincing, or does Smalltalk's own story feel different in some important way the comparison misses?

CHAPTER 7 KEY TAKEAWAYS

- Xerox's early restrictive licensing, and the cost/scarcity of hardware capable of running a full image, both genuinely limited adoption
- C++ and Java won largely by offering OOP incrementally, on top of familiar C-family syntax, not through technical superiority
- Sun's own open licensing and "write once, run anywhere" platform strategy for Java was a real business-strategy advantage over Xerox's own approach
- The image model's poor fit with file-based version control became a serious liability as team-based, collaborative development matured
- Deploying a clean, minimal artifact from a live image was a genuinely harder problem than shipping a compiled binary or JAR
- This is a story about adoption economics and timing, not technical inferiority — echoing the site's own C-vs-Rust honest lesson
- Smalltalk didn't disappear — Squeak and Pharo carry it forward, covered next in smalltalk1-8

CHAPTER 8 OF 8

Smalltalk Today — Squeak, Pharo, and What Survives

Smalltalk

Chapter 8 · Smalltalk Today — Squeak, Pharo, and What Survives

`smalltalk1-7` closed with a promise: Smalltalk didn't disappear. This final chapter is where it actually lives today — and the place to step back and see the full shape of what this course has been tracing across seven prior chapters.

Squeak — Carrying the Dynabook Forward

Squeak, created in 1996 by a team that included Alan Kay himself, alongside Dan Ingalls and others — the same names from `smalltalk1-2` — is an open-source Smalltalk implementation built specifically to continue exploring the *original* Dynabook vision, not just to be "a modern Smalltalk" in the abstract. It's a direct continuation of the specific educational mission Kay started with in 1972.

Etoys, Squeak's own educational environment, uses direct manipulation and a simplified, visual, tile-based scripting approach rather than typed message-send syntax — genuinely closer in spirit to Scratch, the much later and far more widely known educational programming environment, than to a typical professional programming setup. Worth naming honestly: Scratch itself was directly influenced by Squeak and Etoys' own work — one more thread of invisible legacy, in the same spirit as `smalltalk1-5`'s own MVC material.

Squeak has been used in real, deployed educational programs, including the One Laptop Per Child project — a genuine, direct, modern-day echo of the original Dynabook mission: cheap, accessible computers built specifically for children's learning, arriving decades after Kay's own 1972 proposal.

Pharo — A Modern, Professional Open-Source Smalltalk

Pharo, forked from Squeak in 2008, is explicitly aimed at professional software development rather than education specifically — an actively maintained, modern open-source Smalltalk with a real, if niche, community and genuine production use cases, not purely a museum piece or teaching toy.

Pharo still carries forward the live image and class-browser model from `smalltalk1-4` largely unchanged in spirit, while building real modern tooling — package management, testing frameworks — attempting to bridge some of `smalltalk1-7`'s own honest gaps.

STILL AN ONGOING TENSION, NOT A SOLVED PROBLEM

Worth being honest rather than triumphant here: Pharo hasn't "solved" the version-control friction described in `smalltalk1-7` so much as built reasonable, still-somewhat-bolted-on tooling around it — similar in spirit to what Smalltalk-80's own vendors already attempted decades earlier. The underlying structural mismatch between a live image and a file-based diff/commit model remains real.

What Survives, Restated

A brief, explicit walk back through this course's own full arc: `smalltalk1-1` set the "before" state — pre-object programming, and Simula 67 as the genuine technical precursor. `smalltalk1-2` gave that precursor a motivating *reason* to become something more — the Dynabook vision. `smalltalk1-3` delivered the radical philosophy that vision actually demanded: everything is an object, everything is a message, with no exceptions anywhere, not even for control flow. `smalltalk1-4` showed the environment that philosophy required — a live, persistent image, not compiled files. `smalltalk1-5` traced the visible, still-everywhere-in-use legacy — MVC and the GUI itself. `smalltalk1-6` traced the direct language lineage — Objective-C's syntax, Ruby's philosophy — while honestly correcting an earlier overclaim about how much of OOP really traces back here. `smalltalk1-7` gave the honest account of why the commercial market didn't reward the most philosophically coherent option. This chapter closes the loop.

The Real Closing Point

Object-oriented programming as most working programmers understand and use it today is not an independent invention by Java, or C++, or Ruby, or any of its more commercially successful descendants. It's a direct, traceable inheritance from a specific research effort, motivated by a specific, still only partially realized vision — truly democratized, child-accessible personal computing — conducted by a specific, identifiable group of people at a specific place and time.

Most programmers writing classes, calling methods, and relying on inheritance every single day have never heard any of this history. In a strange way, that's the most complete measure of how deeply this work actually won — even while the specific language that carried it lost the commercial battle described in `smalltalk1-7`.

IF YOU'RE CURIOUS TO GO FURTHER

This course, in keeping with its own humanities-style format, never asked you to write a line of actual Smalltalk. If any of this made you curious, both Squeak and Pharo are real, freely available, still-running systems — genuinely worth opening once, if only to see the live image and class browser described back in `smalltalk1-4` for yourself.

Reflection Questions

QUESTION 1

This chapter claims the deepest measure of Smalltalk's influence is that most programmers using its ideas today have never heard of it. Do you find that a satisfying kind of "winning," or does it feel more like a loss dressed up as a win? Explain your reasoning.

QUESTION 2

Looking back across all eight chapters, which single idea from this course changed your own understanding of object-oriented programming the most — the message-passing philosophy (Ch.3), the live image (Ch.4), the MVC/GUI legacy (Ch.5), or something else? Why that one specifically?

QUESTION 3

One Laptop Per Child is described as a "modern-day echo" of the 1972 Dynabook vision, decades later. Do you think that vision — a computer genuinely built around a child's own learning and creativity, rather than adapted from an adult business tool — has actually been achieved anywhere today? What would fully achieving it even look like?

COURSE SUMMARY — THE FULL ARC

- Ch.1: Pre-object programming, and Simula 67 as the genuine technical precursor to class-based objects
- Ch.2: The Dynabook vision — a personal computer for children — as Smalltalk's real motivating goal
- Ch.3: Everything is an object, everything is a message — the radical, uniform core philosophy
- Ch.4: The live image and class browser — the environment the philosophy required
- Ch.5: MVC and the GUI — Smalltalk's most invisible, most successful legacy
- Ch.6: The direct language lineage (Objective-C, Ruby) and the honest two-branch OOP correction
- Ch.7: Why the commercially winning languages weren't the most philosophically coherent ones
- Ch.8: Squeak, Pharo, and the closing point — modern OOP is a traceable inheritance, not an independent invention