Software Processes & Incremental Change

Oscar Chaparro

Agenda

1. Essential properties of software
2. Software engineering paradigms & anomalies
3. A brief history of software engineering
4. Software lifespan & the staged model
5. Incremental change
6. Concept location & impact analysis

Facts about software

Society depends on software more every year	…and it is still frequently unreliable and of poor quality.
Software is treated as an engineering product	…but it is more complex than many things humans build.
Maintenance is treated as low-status work	…but 75–95% of what software costs is spent there.
Software changes for business and technical reasons	…and a system that has stopped changing is effectively dead.
AI is changing how software gets built — a new paradigm	…and we don’t yet know the best ways to use it.

Essential properties of software

Why software is different from other products

Essential vs. accidental difficulties

Essential difficulties

Intrinsic to software — they determine its nature and do not change

Complexity

Conformity

Changeability

Invisibility

Discontinuity

Essential vs. accidental difficulties

Essential difficulties

Intrinsic to software — they determine its nature and do not change

Complexity

Conformity

Changeability

Invisibility

Discontinuity

Accidental difficulties

Result of the implementation process and tech advancements — they change from time to time

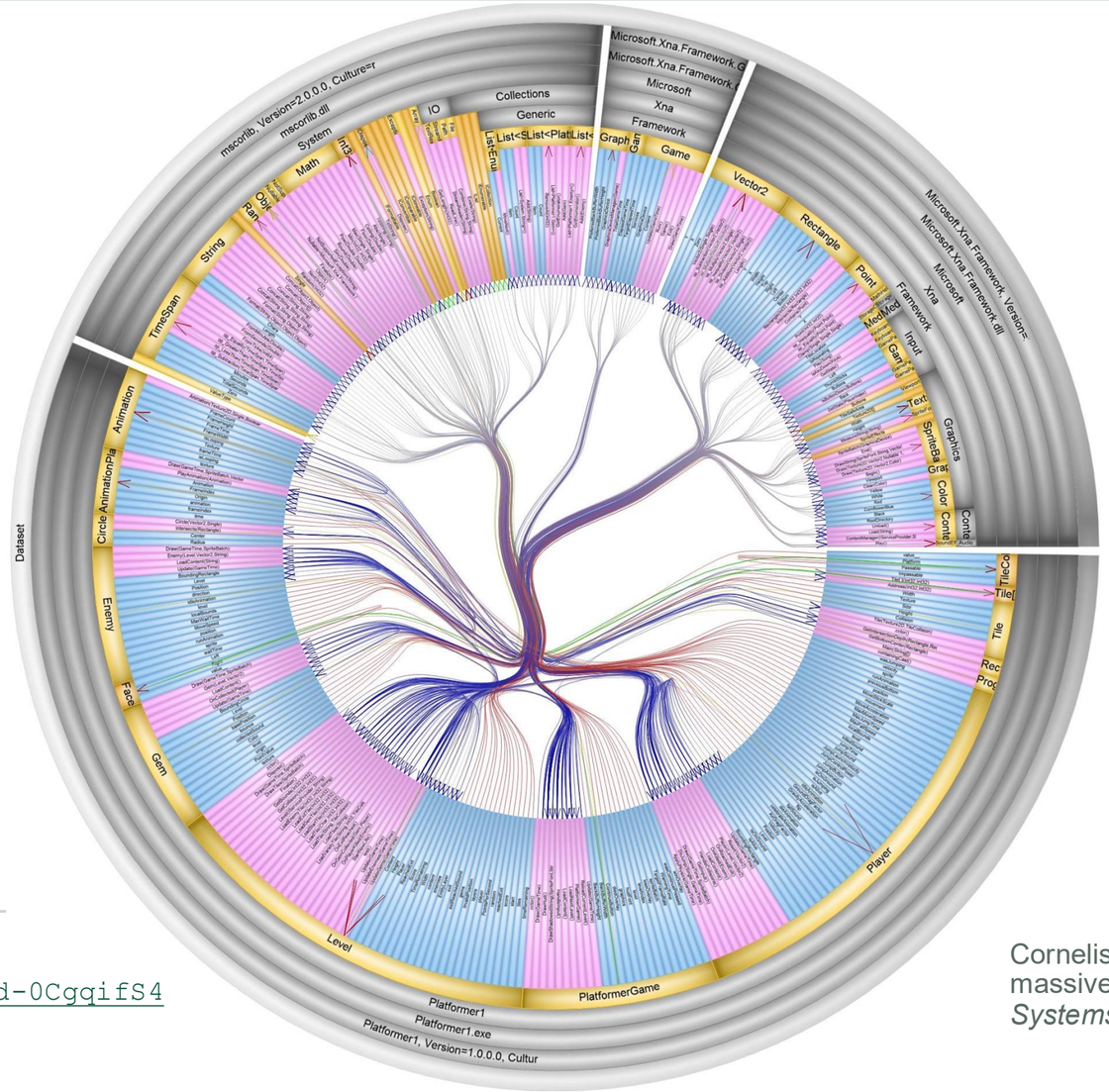
Hardware speed, memory size

Quirks of operating systems, languages, compilers, processes

Programming paradigm (functional, object oriented, …)

Role of the software (critical, noncritical)

Complexity



https://www.youtube.com/watch?v=Rd-0CgqifS4

Cornelissen, Bas, et al. “Execution trace analysis through massive sequence and circular bundle views.” *Journal of Systems and Software* 81, no. 12 (2008): 2252–2268.

Complexity

Software is among the most complex systems ever created.

Programs may be written in convoluted ways.

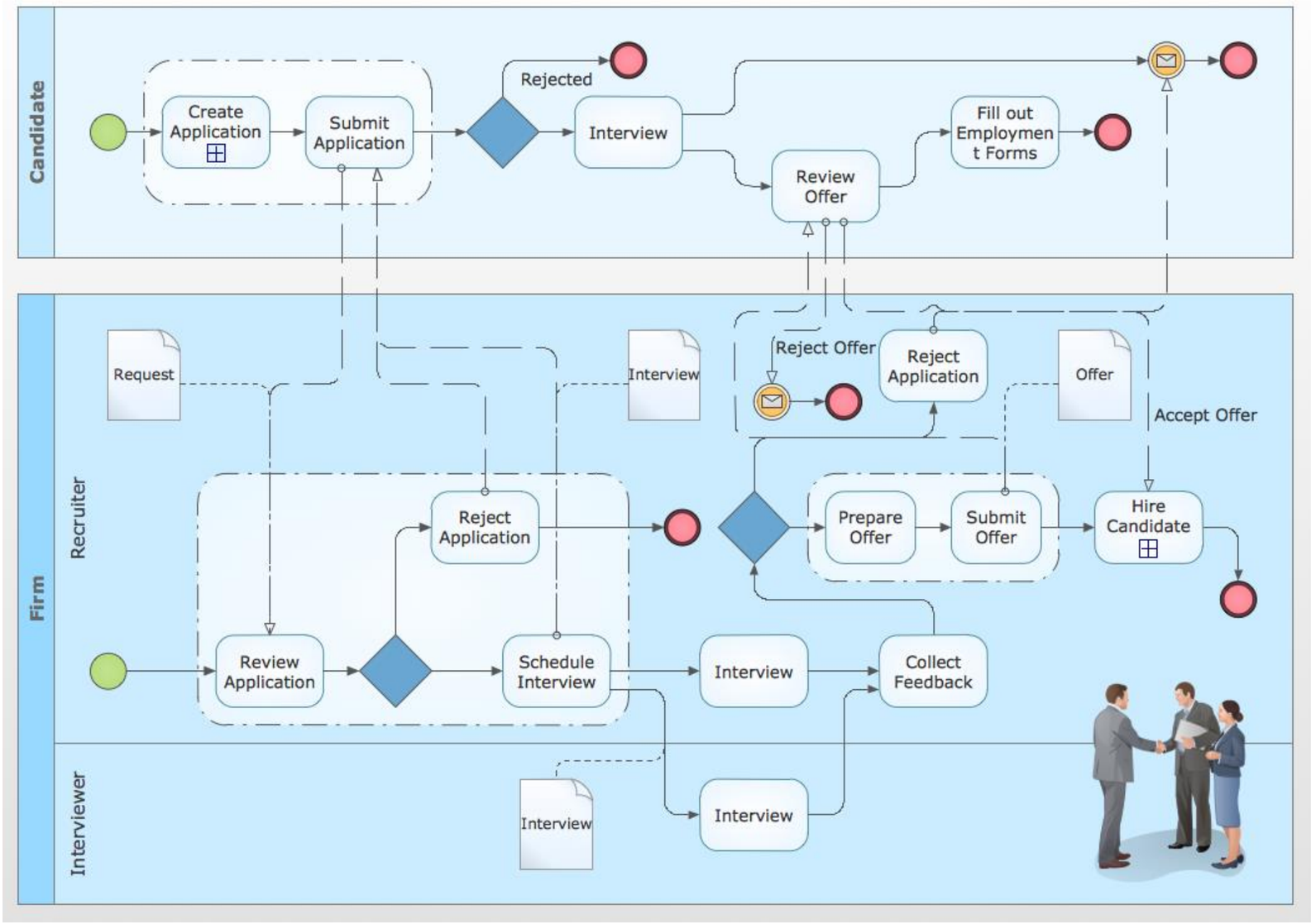
Software is composed of numerous interacting structures and dependencies.

Our short-term memory can accommodate only about seven items.

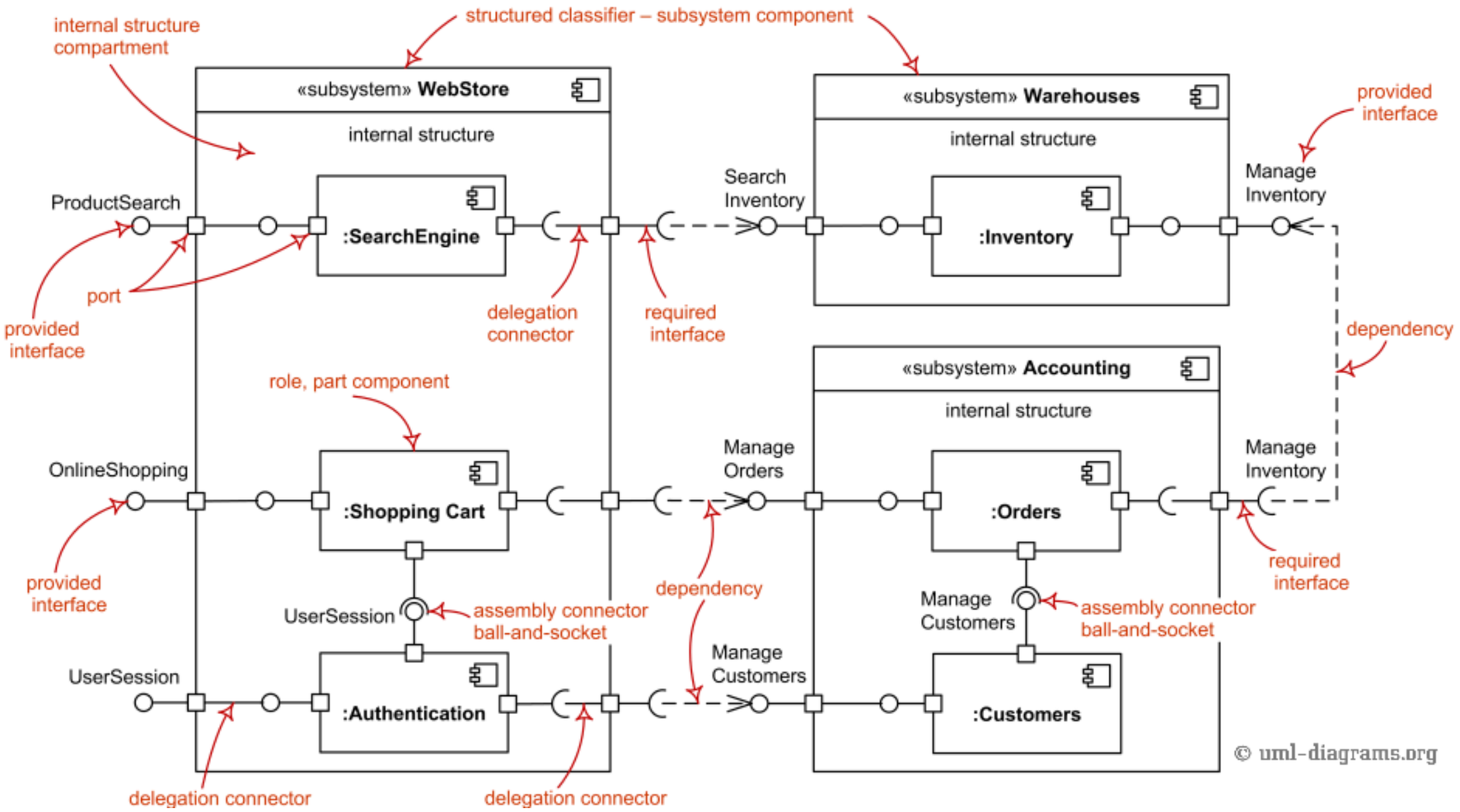
Complexity is the cause of most software problems.

> **Strategies:** modeling, decomposition, as-needed approach, abstraction

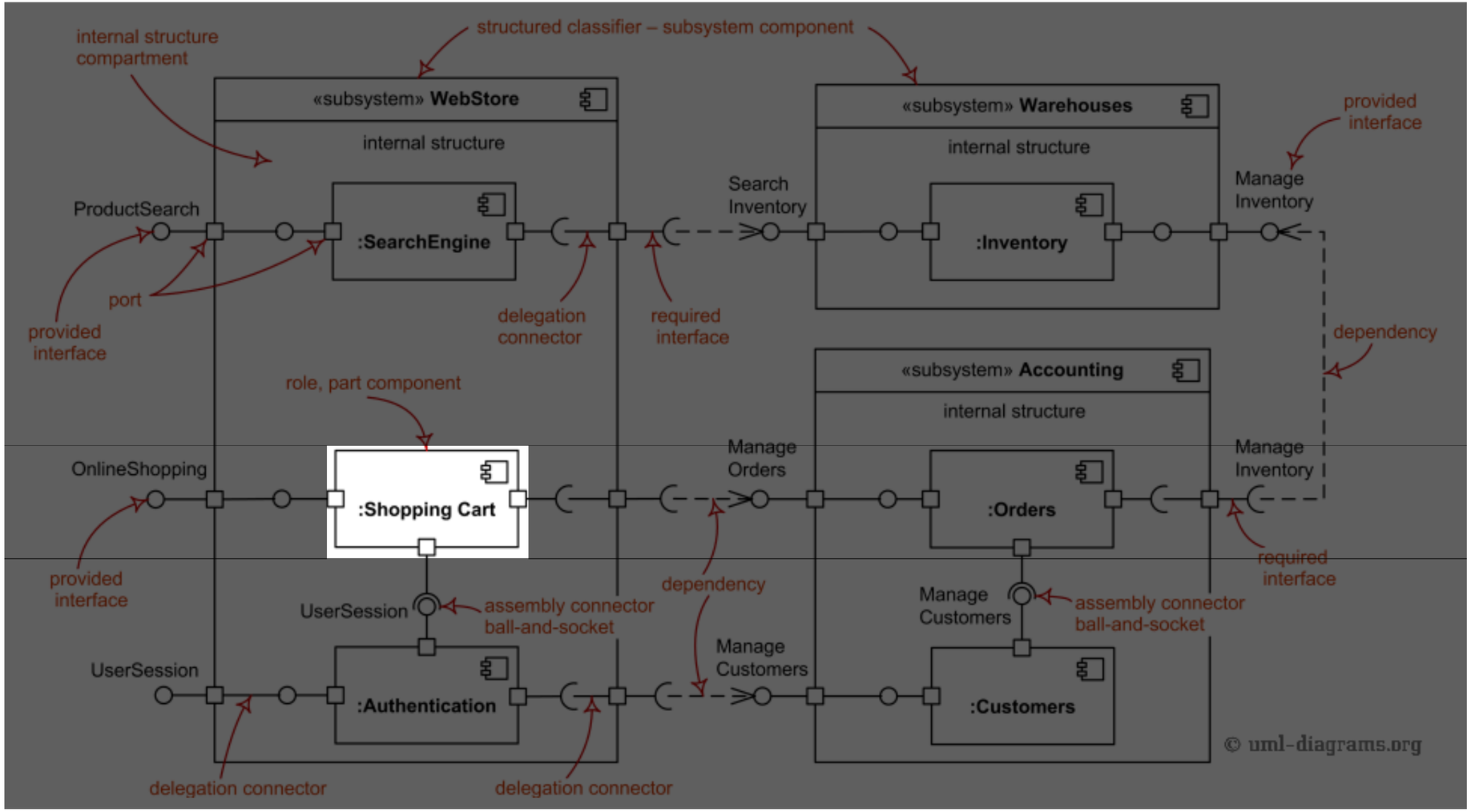
Models


Candidate
Create Application
Submit Application
Rejected
Interview
Review Offer
Fill out Employment Forms
Firm
Recruiter
Request
Interview
Reject Offer
Reject Application
Offer
Accept Offer
Reject Application
Prepare Offer
Submit Offer
Hire Candidate
Review Application
Schedule Interview
Interview
Collect Feedback
Interviewer
Interview
Interview

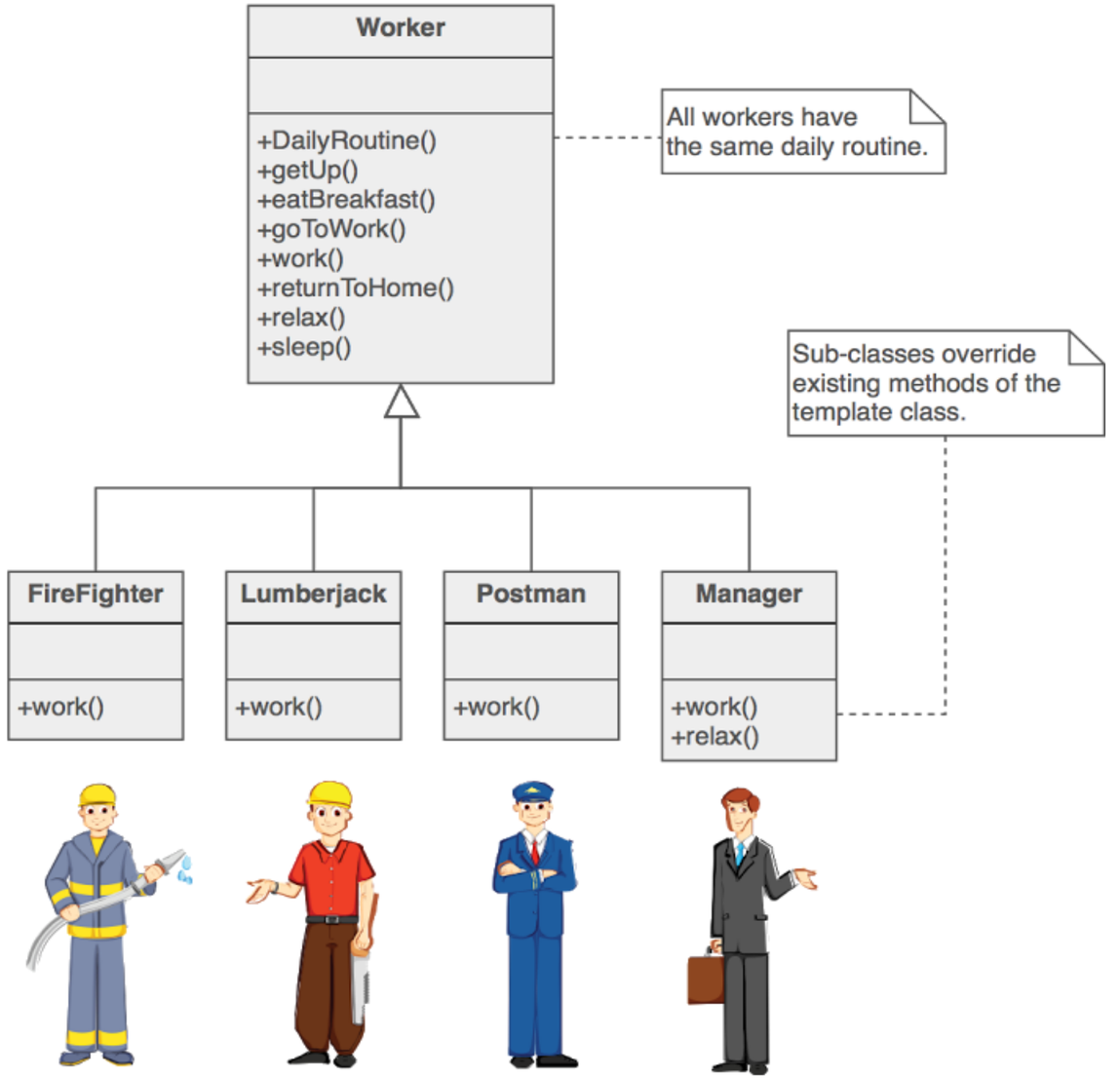
Decomposition



As-needed approach



Abstraction



Conformity

Software is part of a larger system: the domain.

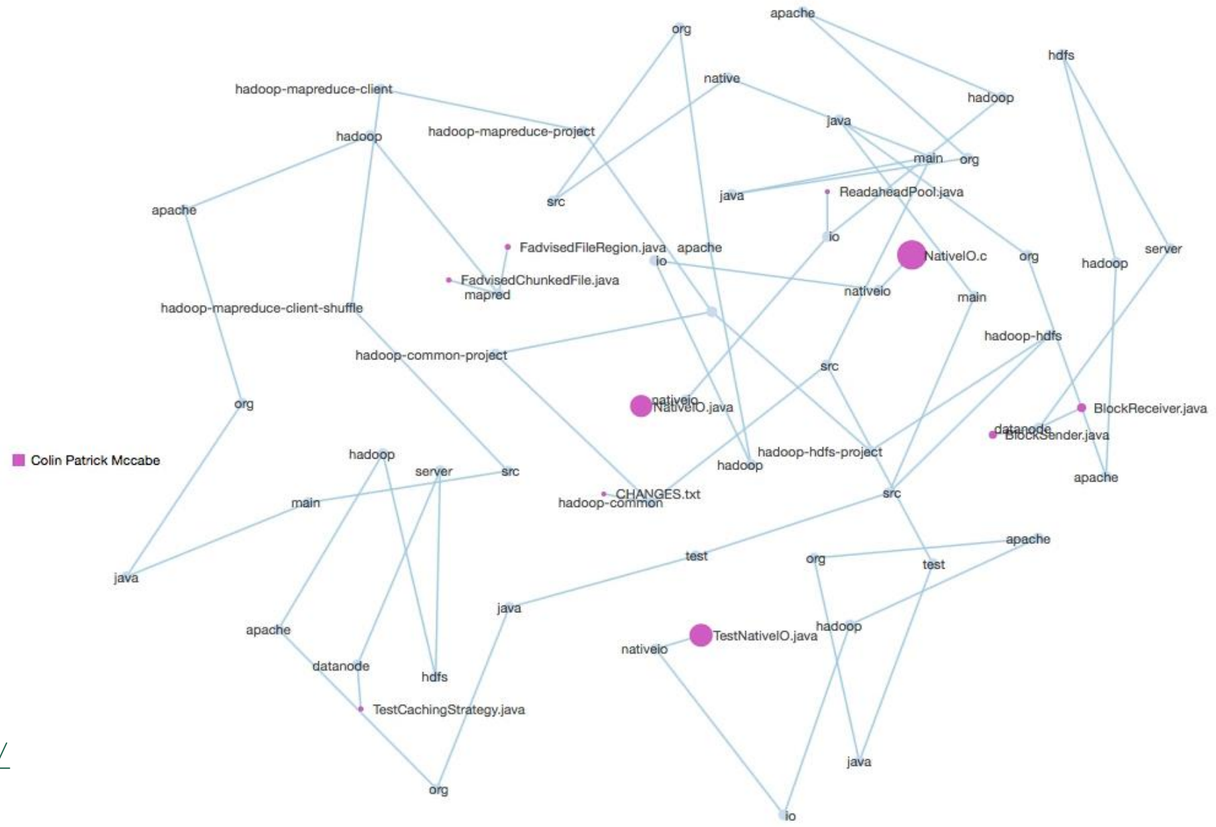
The software glues it all together, adding even more complexity.

Changes in the domain force software change.

> **Strategies:** domain knowledge

Changeability

commitMag: HADOOP-7824. NativeIO.java flags and identifiers must be set correctly for each platform, not hardcoded to their Linux values (Martin Walsh via Colin P. McCabe)(cherry picked from commit 21d10ccc6e463cf250414264c7



http://jamuhl.github.io/octoviz/

Changeability

Software is constantly changing — at least, successful software is.

Software is easy to change: only keystrokes are needed

It is the foundation of iterative approaches

Changing (complex) software correctly is much harder

Bad changes can increase complexity

Yesterday’s comprehension of the code may already be obsolete.

Invisibility

Senses cannot be easily used in comprehension.

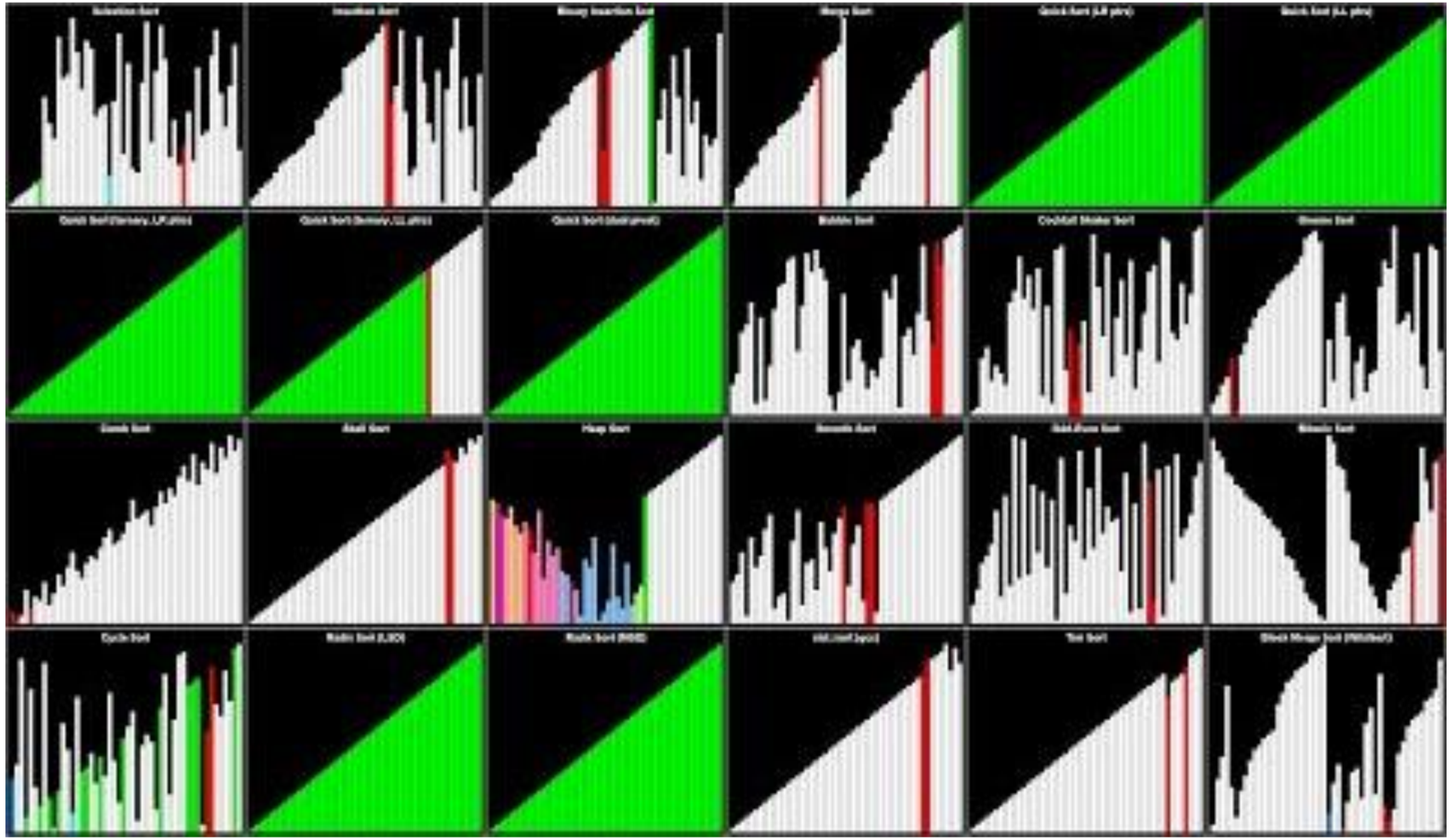
Software is an abstract construct/artifact (like math)

We cannot smell, touch, hear, or taste software

Sight is limited: GUI, code, models

> **Strategies**: visualization, sonification

Software visualization: algorithms



https://www.youtube.com/watch?v=BeoCbJPuvSE

Software visualization: code structure

https://youtu.be/AaHJRVQ3Z1E

Discontinuity

A small change in the input can produce a huge change in the output.

Unlike the continuous, semi-linear systems other engineers work with.

Which essential property is this?

Complexity	Conformity	Changeability	Invisibility	Discontinuity
Many interacting parts; memory holds ~7 things	Part of a larger domain; domain forces change	Software changes constantly; changing it correctly is hard	Software is invisible. Senses are hard to use in comprehension	Small input change, huge output change

A

A client sent new payment API requirements 2 weeks before launch. The entire checkout module had to be redesigned.

B

A developer spent 3 days drawing diagrams just to explain what one module does to a new teammate.

C

Fixing a one-line bug in the sorting function unexpectedly broke the inventory report.

Which essential property is this?

Complexity	Conformity	Changeability	Invisibility	Discontinuity
Many interacting parts; memory holds ~7 things	Part of a larger domain; domain forces change	Software changes constantly; changing it correctly is hard	Software is invisible. Senses are hard to use in comprehension	Small input change, huge output change

A

A client sent new payment API requirements 2 weeks before launch. The entire checkout module had to be redesigned.

Conformity / Changeability

B

A developer spent 3 days drawing diagrams just to explain what one module does to a new teammate.

Invisibility

C

Fixing a one-line bug in the sorting function unexpectedly broke the inventory report.

Discontinuity / Complexity

Software engineering paradigms

How the field learned to handle software's essential properties

Paradigms & anomalies

Paradigm |ˈperəˌdīm|

engineering A coherent tradition of engineering research and practice.

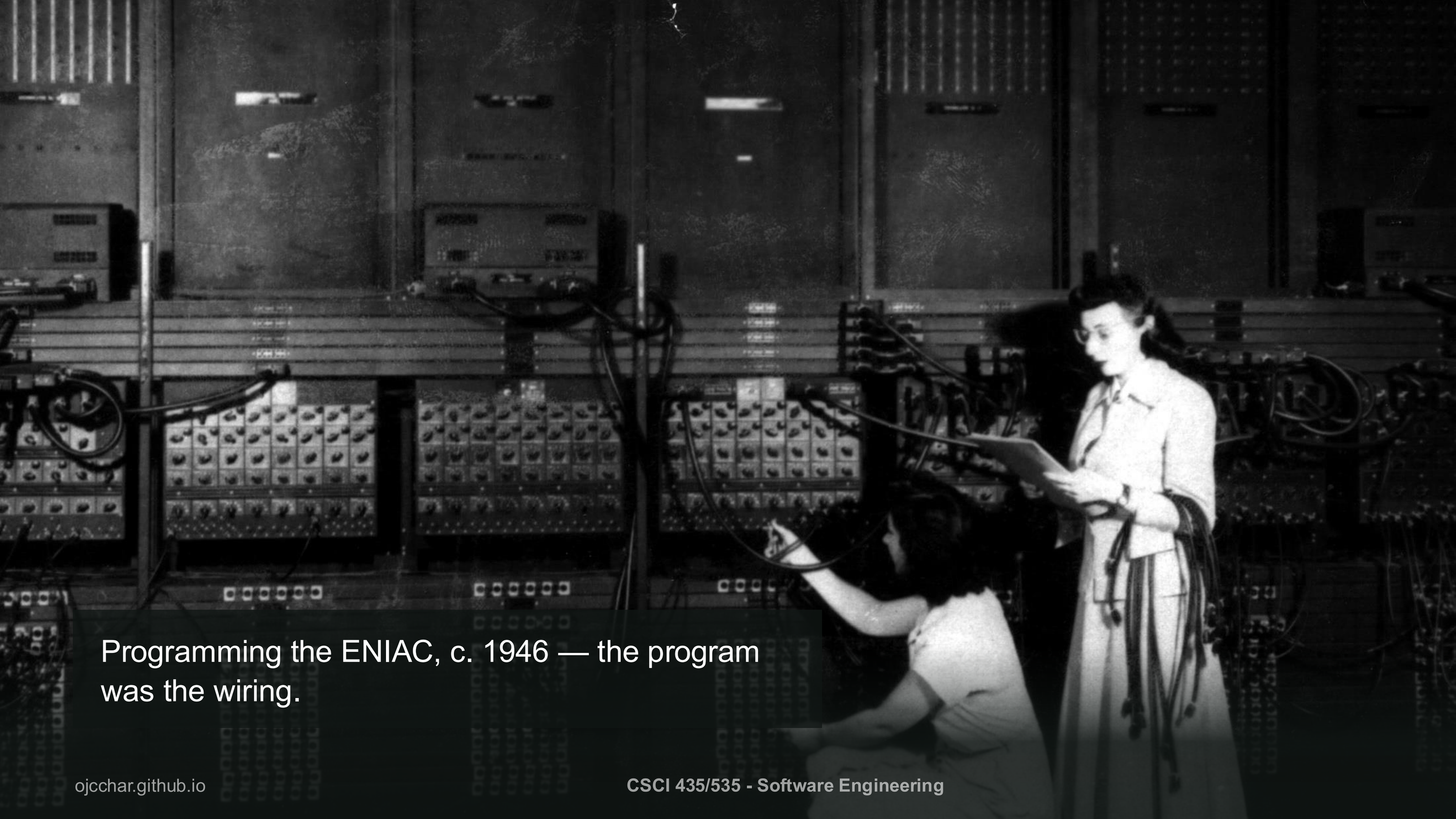
Rajlich, after Kuhn (1962)

Anomaly |əˈnäməlē|

engineering An important fact that directly contradicts a paradigm.

Kuhn

Paradigm shifts occur when anomalies accumulate and the existing model can no longer explain them.



Programming the ENIAC, c. 1946 — the program was the wiring.

The beginnings of software

The first paradigm: programming as craft

Software separated from hardware in the 1950s:

- Emerged as a distinct technology · became an independent product

Original programmers: electrical engineers and mathematicians

- Used ad-hoc techniques from their former fields · no established discipline, no shared methods

This worked — until it didn't.

The anomaly: complexity

Software complexity.

Previous (ad hoc) techniques did not scale up.

Unknown methods to manage software projects.

> *“Demands of the new operating system OS/360 taxed the limits of the programmers, project managers, and the resources of the IBM corporation.”*
>
> Brooks, *The Mythical Man-Month* (1975)

The response: software engineering

Paradigm change established software engineering as a discipline.

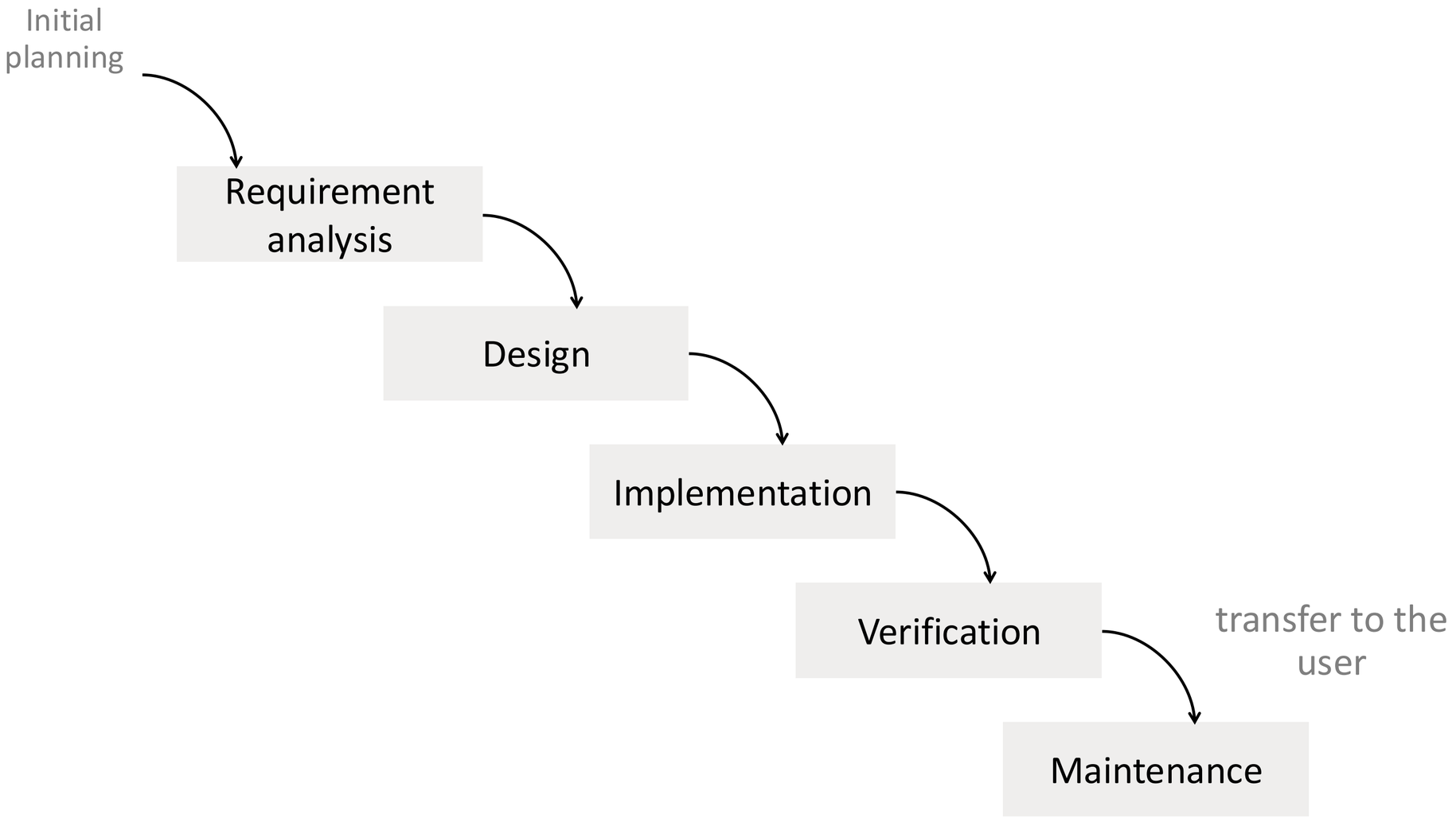
Software requires people with specialized skills.

Software design established as an important consideration.

A well-defined process is needed.

> The major change was the introduction of the **waterfall metaphor** — borrowed from traditional management and engineering of other products.

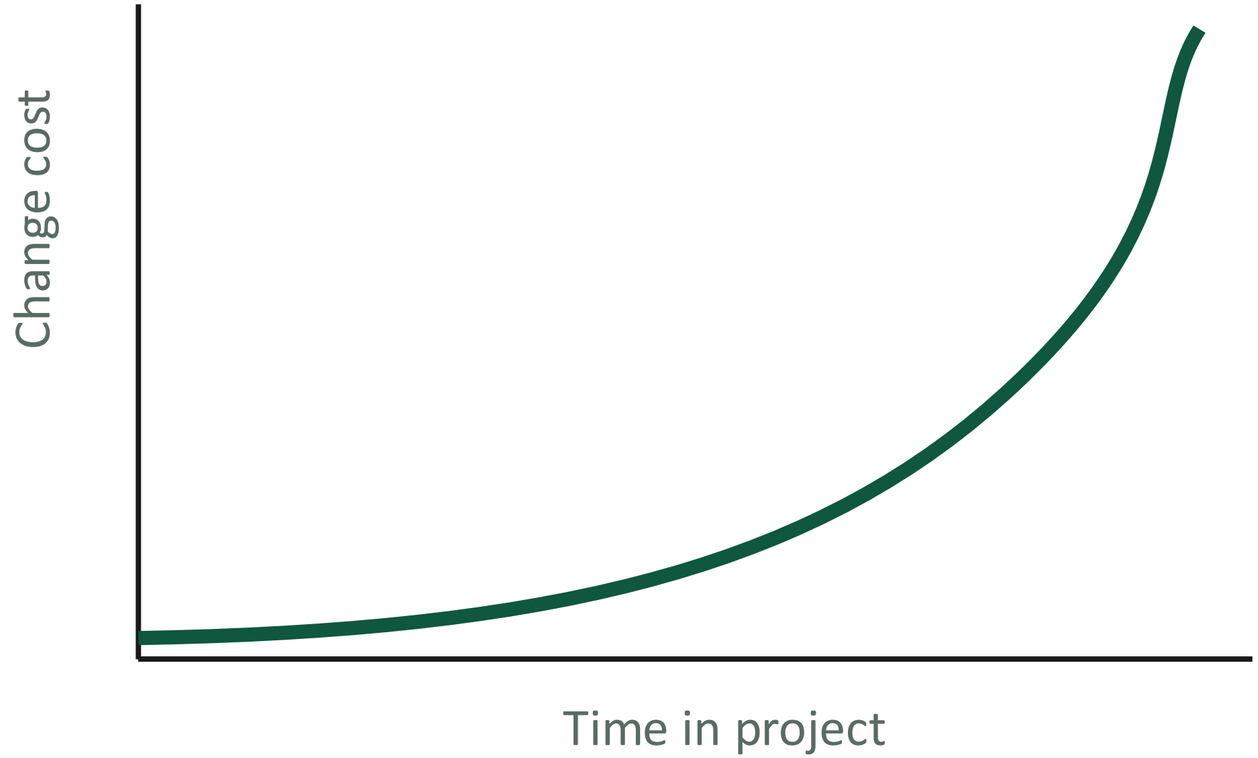
The waterfall model



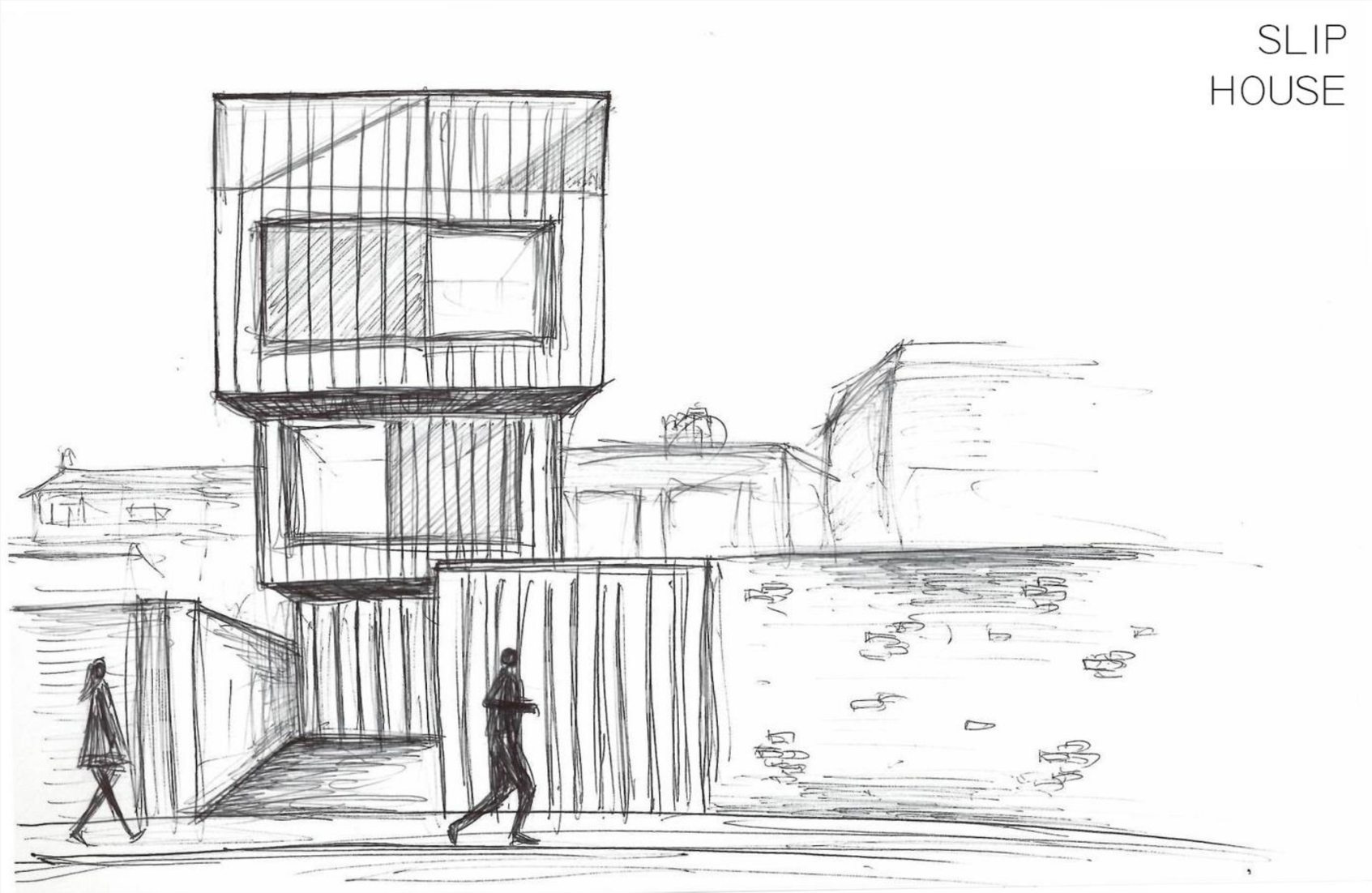
The waterfall model promise

Avoid unnecessary and expensive late changes.

How: invest heavily in requirements and design upfront, and freeze scope scope before implementation.



The waterfall model in construction



Waterfall works for other products

There are no major changes in requirements.

There is extreme detail when designing products (bridges, buildings, etc.).

The design can be frozen at a certain point and allows to move on to construction.

> Having said that, today's businesses — which are enabled via software — are more dynamic and demand more flexibility.

Let's check if waterfall worked

The Standish Group (1994) surveyed 365 executive managers across 8,380 software projects in banking, manufacturing, healthcare, retail, and more.

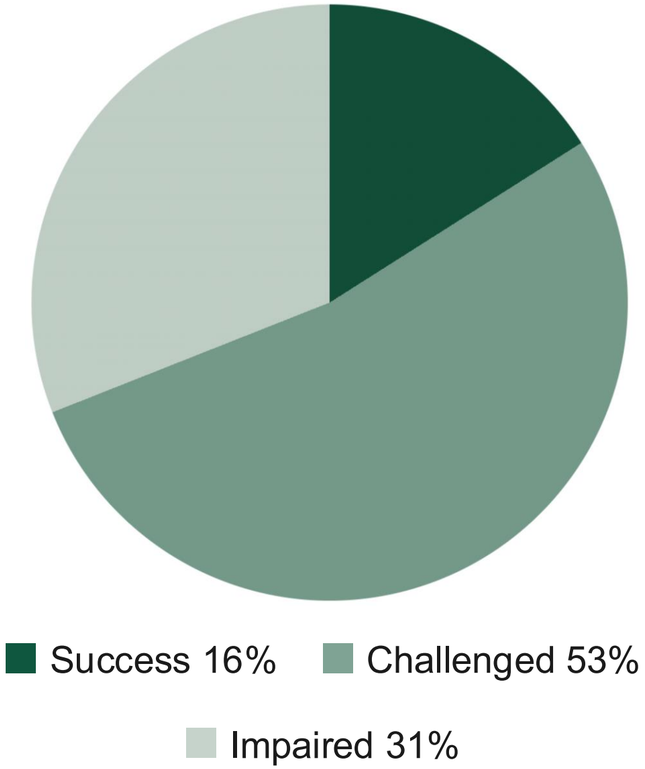
What % of projects were fully successful?

A 16%

B 42%

C 61%

The Standish Group report (1994)



31% of all software projects were cancelled before completion

53% were “challenged”: late, over budget, or substantially reduced in scope — cost reached 189% of estimates

Only **16%** could be called successful

The waterfall paradigm did not solve the problems of software development.

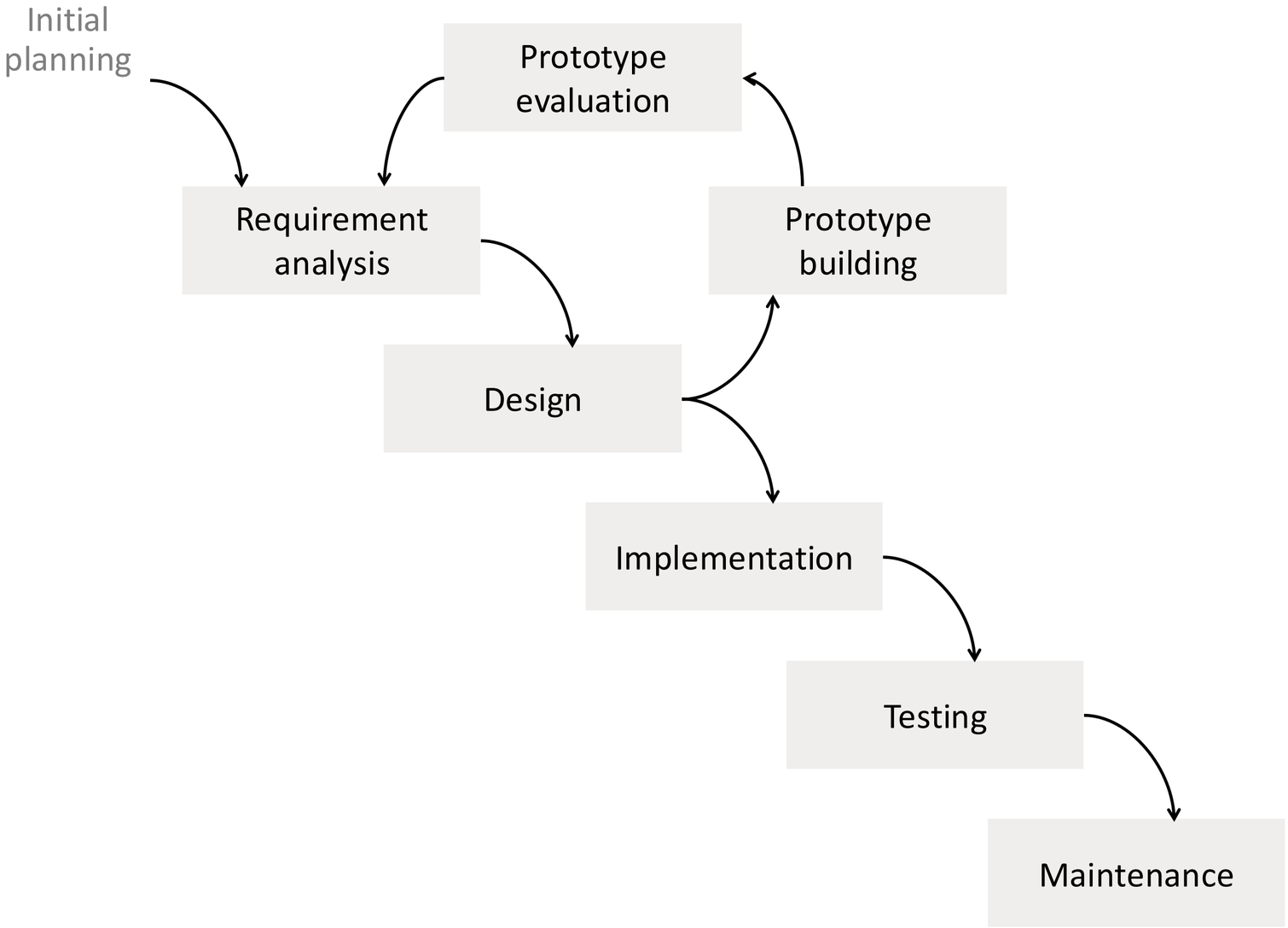
Waterfall is not enough: band-aids

Anticipation of changes

If changes can be anticipated at design time, they can be controlled — parameterization, encapsulation.

Problem: many changes are not anticipated by the original designers, and business opportunities are lost.

Band-aid: prototyping



Waterfall is not enough: band-aids

Prototyping

Build a very preliminary version of the software to refine requirements.

Problem: volatility continues after the prototype is completed; it is only useful for initial initial requirements elicitation.

The anomaly: volatility

Volatility of requirements, technologies, and knowledge. The waterfall model cannot accommodate it.

30%

or more of requirements change during development

Cusumano & Selby

2–3%

per month — the rate at which IT requirements change

Caper-Jones

Today's businesses, enabled via software, are more dynamic and demand more flexibility.

Design a better process

If you had to design a process that handles changing requirements, which would you choose?

A

Deliver everything at the end, but hold quarterly requirements review meetings to update the plan.

B

Break the work into short cycles. Deliver something working at the end of each cycle. Review and adjust priorities after each delivery.

C

Hire a dedicated requirements analyst to fully freeze scope before any coding begins. Never change it.

Design a better process

If you had to design a process that handles changing requirements, which would you choose?

A

Deliver everything at the end, but hold quarterly requirements review meetings to update the plan.

B

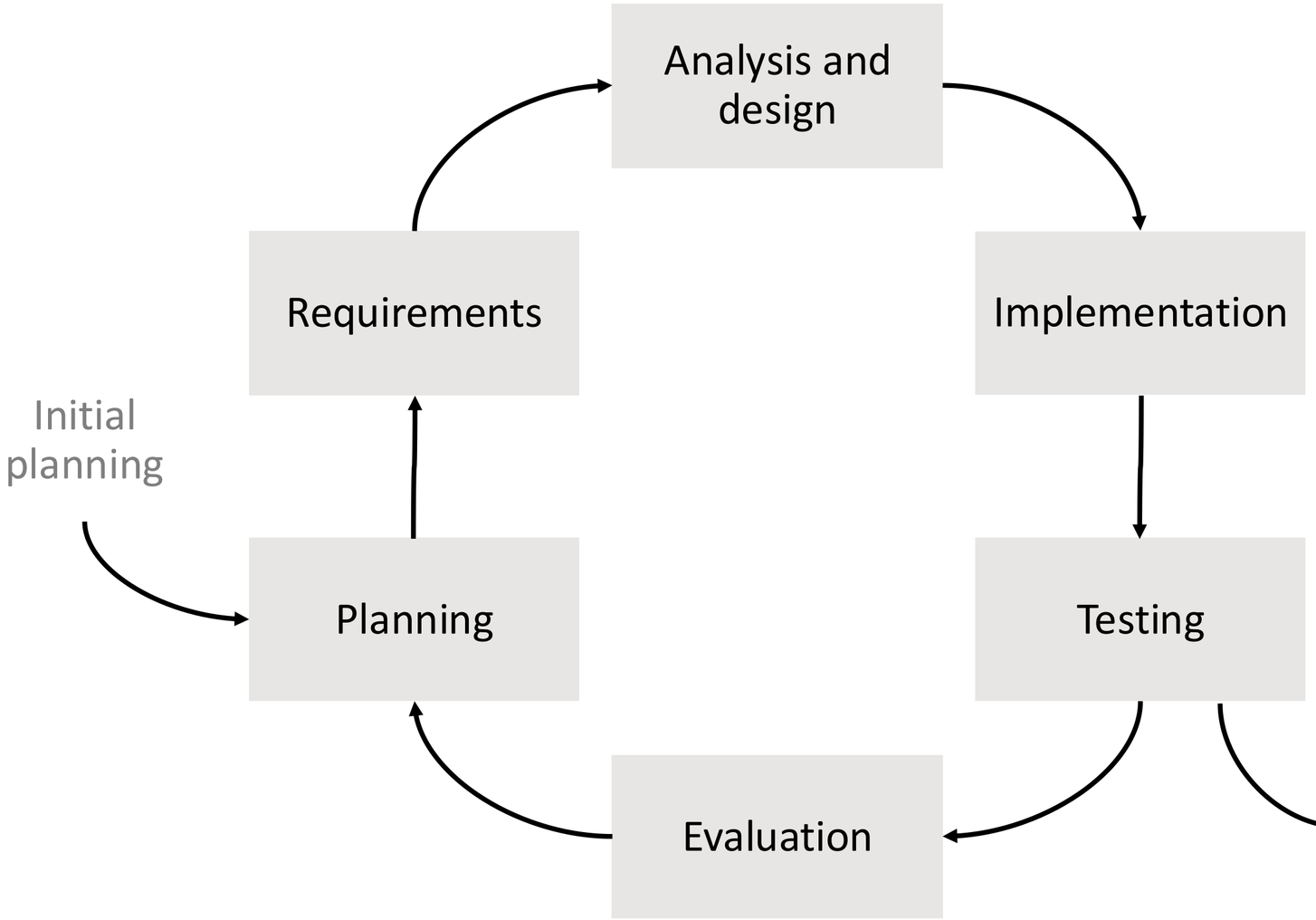
Break the work into short cycles. Deliver something working at the end of each cycle. Review and adjust priorities after each delivery.

The iterative approach

C

Hire a dedicated requirements analyst to fully freeze scope before any coding begins. Never change it.

Iterative software development



The iterative / evolutionary approach

Each iteration delivers working software and allows re-prioritization of the next cycle.

The paradigm change of the 2000s

Iterative / evolutionary development

Unified Process (UP, RUP, OpenUP) Rapid Application Development (RAD)

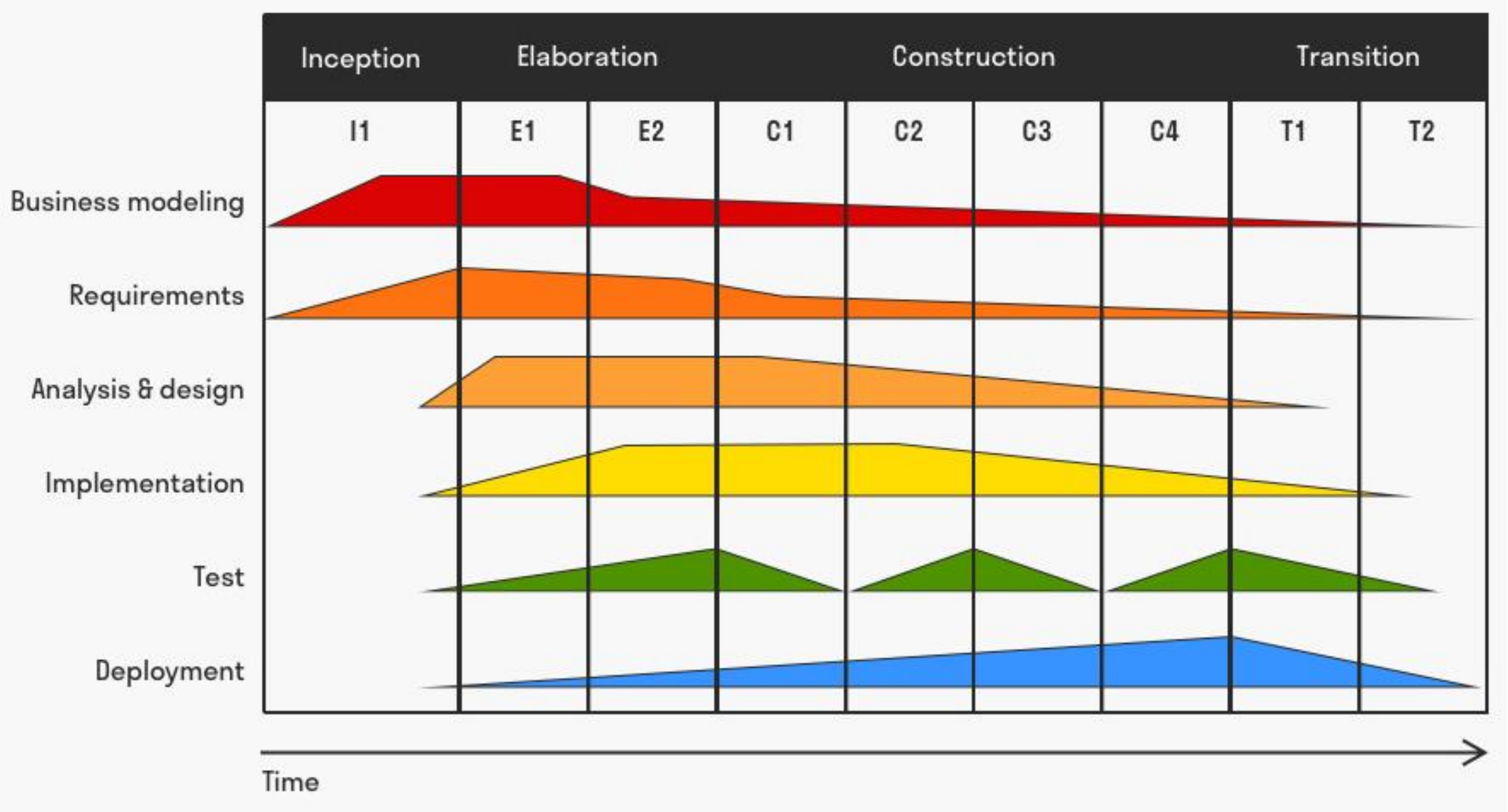
New life-span models emphasize evolution

Staged model of software lifespan (Rajlich) — next section

Agile development

Scrum Extreme Programming (XP) Agile Unified Process

The Unified Process



The Agile Manifesto (2001)

Created by 17 software developers (Kent Beck, Martin Fowler, et al.)

Individuals and interactions over processes and tools

Working software over comprehensive documentation

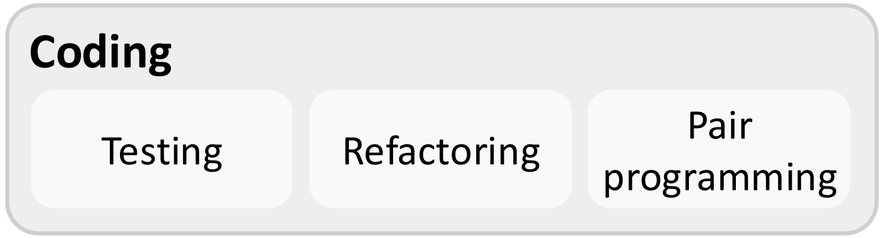
Customer collaboration over contract negotiation

Responding to change over following a plan

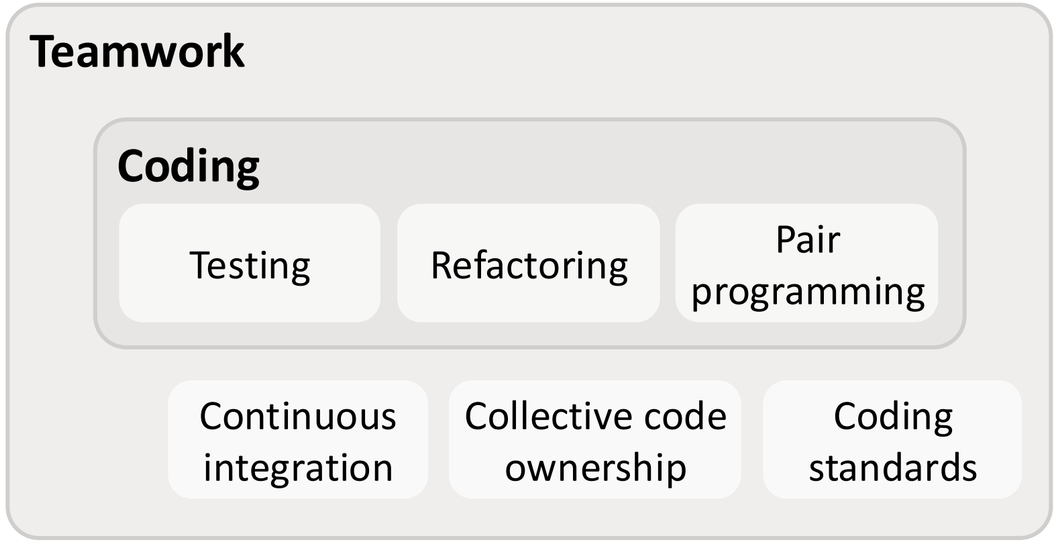
“That is, while there is value in the items on the right, we value the items on the left more.”

agilemanifesto.org

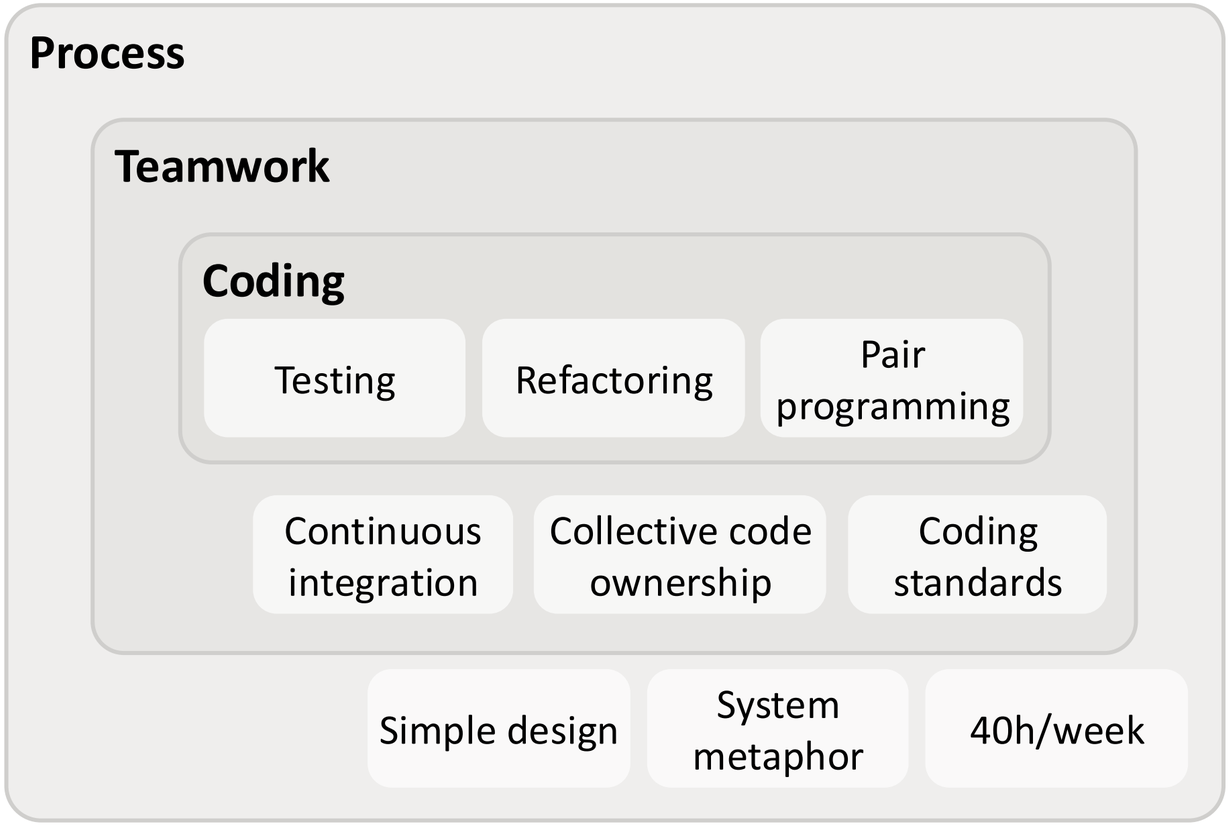
Extreme programming core practices



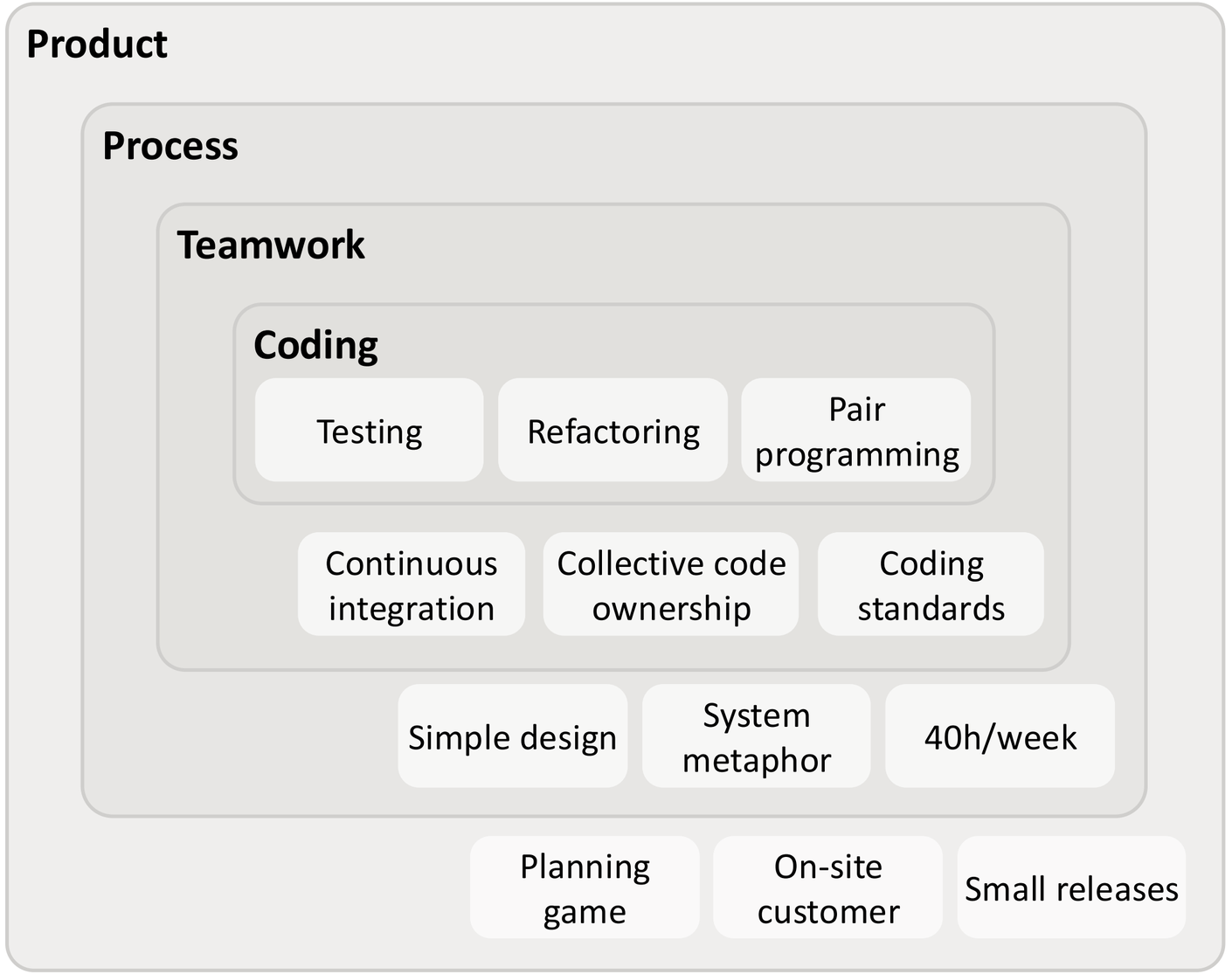
Extreme programming core practices



Extreme programming core practices



Extreme programming core practices



Scrum

An agile framework for managing work in short, fixed-length cycles called called **sprints** — typically 1–4 weeks.

> Rather than planning everything upfront, you commit to a small slice of work, deliver it, get it, get feedback, and repeat.
>
> **Each sprint produces something potentially shippable.**

A lightweight structure for iterative delivery — without prescribing specific engineering practices.

Scrum: roles

Product Owner	Owns the backlog; prioritizes what gets built and why
Scrum Master	Facilitates the process, removes blockers, protects the team
Development team	Self-organizing, cross-functional, does the work

Scrum: artifacts

Product backlog	The prioritized list of all work — features, bugs, tasks
Sprint backlog	The subset the team commits to in one sprint
Increment	The working software produced at the end of the sprint

Scrum: ceremonies

Sprint planning	Team selects work and breaks it into tasks (before start of sprint)
Daily standup	Brief (10-15 min) daily sync — what I did, what I will do, blockers
Sprint review	Demo the increment to stakeholders, get feedback
Retrospective	Reflect on the process, improve for next sprint

What Scrum is not

It says nothing about how to write code, test, integrate, or deploy. That is left to the team.

XP fills those gaps with practices like pair programming and continuous integration.

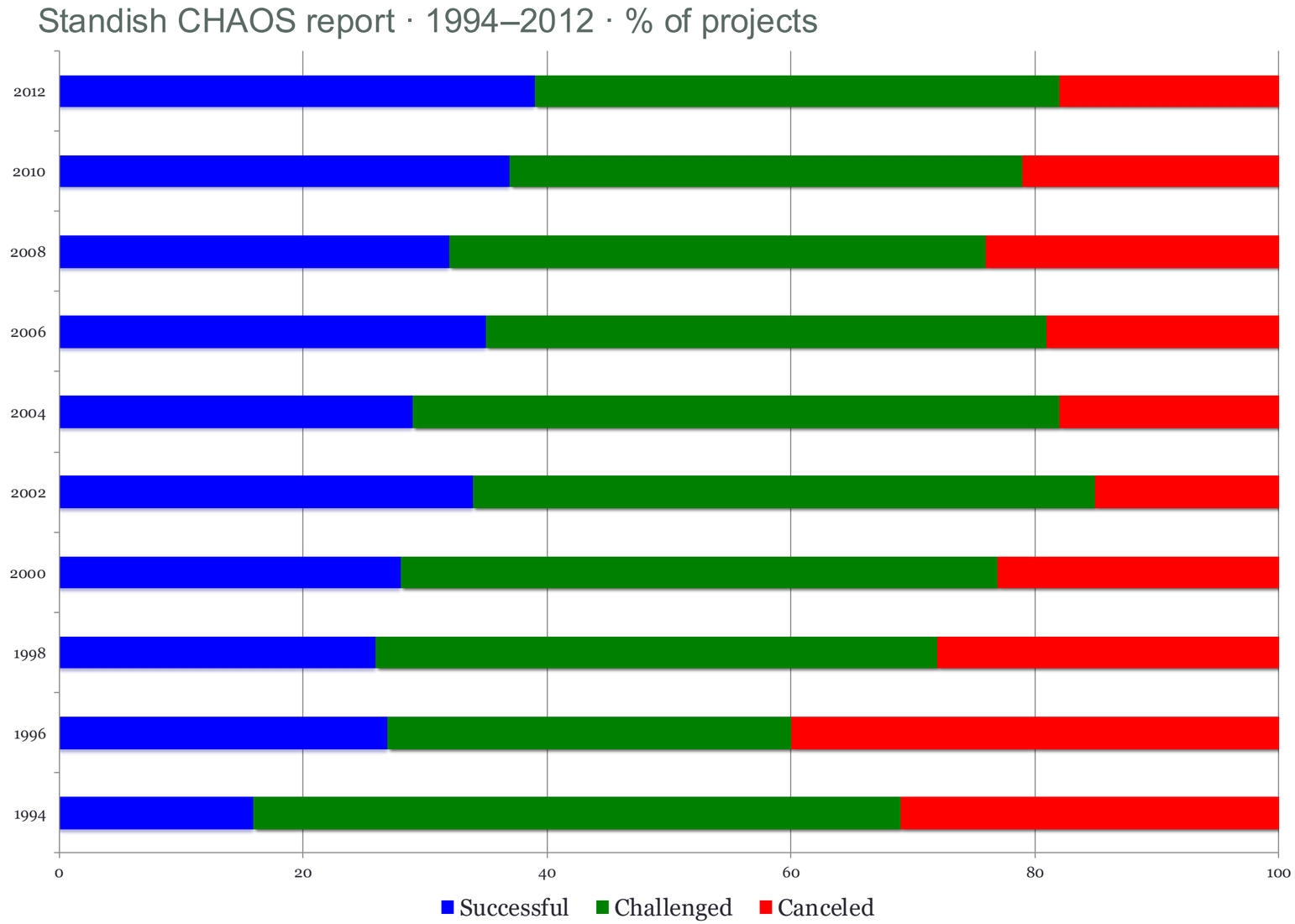
Many teams run a hybrid — sometimes called “Scrumban” or “XP + Scrum.”

XP and Scrum

	Extreme Programming	Scrum
What it is	A set of engineering practices	A framework for managing the work
Focus	How the code gets written	How the team organizes and delivers
Prescribes	Testing, refactoring, pair programming, CI	Roles, sprints, backlog, ceremonies
Says about code	A great deal	Nothing
Iteration	1–2 weeks	2–4 week sprints
Roles	Coach, customer, programmer, tester	Product owner, Scrum master, team

Most teams combine them: Scrum for the process, XP practices for the engineering.

The paradigm shift improved outcomes



SUCCESS RATE

16% → ~39%

CANCELLED RATE

~31% → ~18%

Progress — but still far from ideal.

All paradigms currently coexist

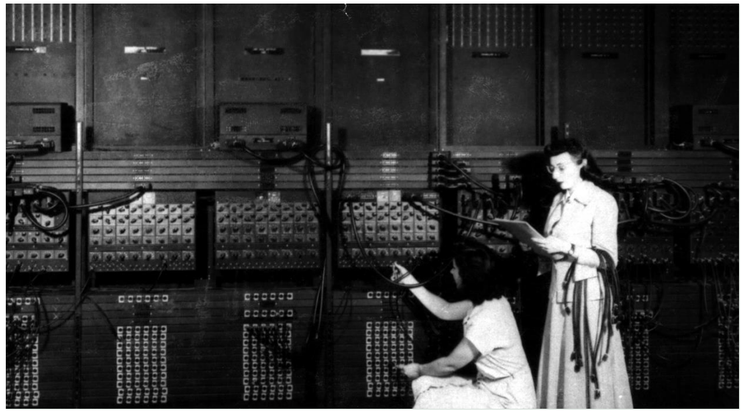
Ad hoc	Still used by some single programmers — small games, mobile apps, research prototypes. Programming as craft rather than engineering.
Waterfall	Still used where requirements are stable — small or short-lived projects, unusually stable environments. Some managers still insist on it.
Iterative / agile	The new mainstream. Handles volatility. Evolution is the key process.

This course uses iterative/agile (sprints). You are working in the third paradigm.

A brief history of software engineering

To understand where we are, we need to know where we came from

How SE evolved · 1950s–1970s



1950s

Hardware/software separation. Programming as craft, borrowed from EE and math.

1960s

Software crisis. NATO conference (1968). Programming becomes a discipline: SE established, with principles rather than personal habit.

1970s

Waterfall model. Design principles take hold: structured programming, modularity, information hiding, early object-oriented ideas.

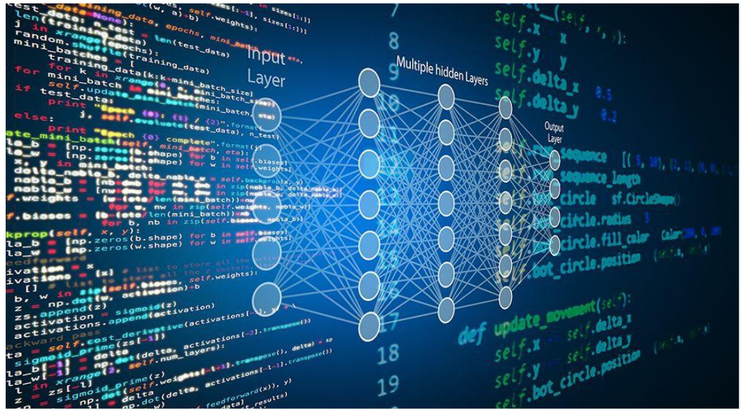
How SE evolved · 1980s–today

1980s

OOP (Smalltalk, C++). Personal computing. Internet foundations. Brooks becomes canon: *The Mythical Man-Month* , *No Silver Bullet* (1986).

1990s–2000s

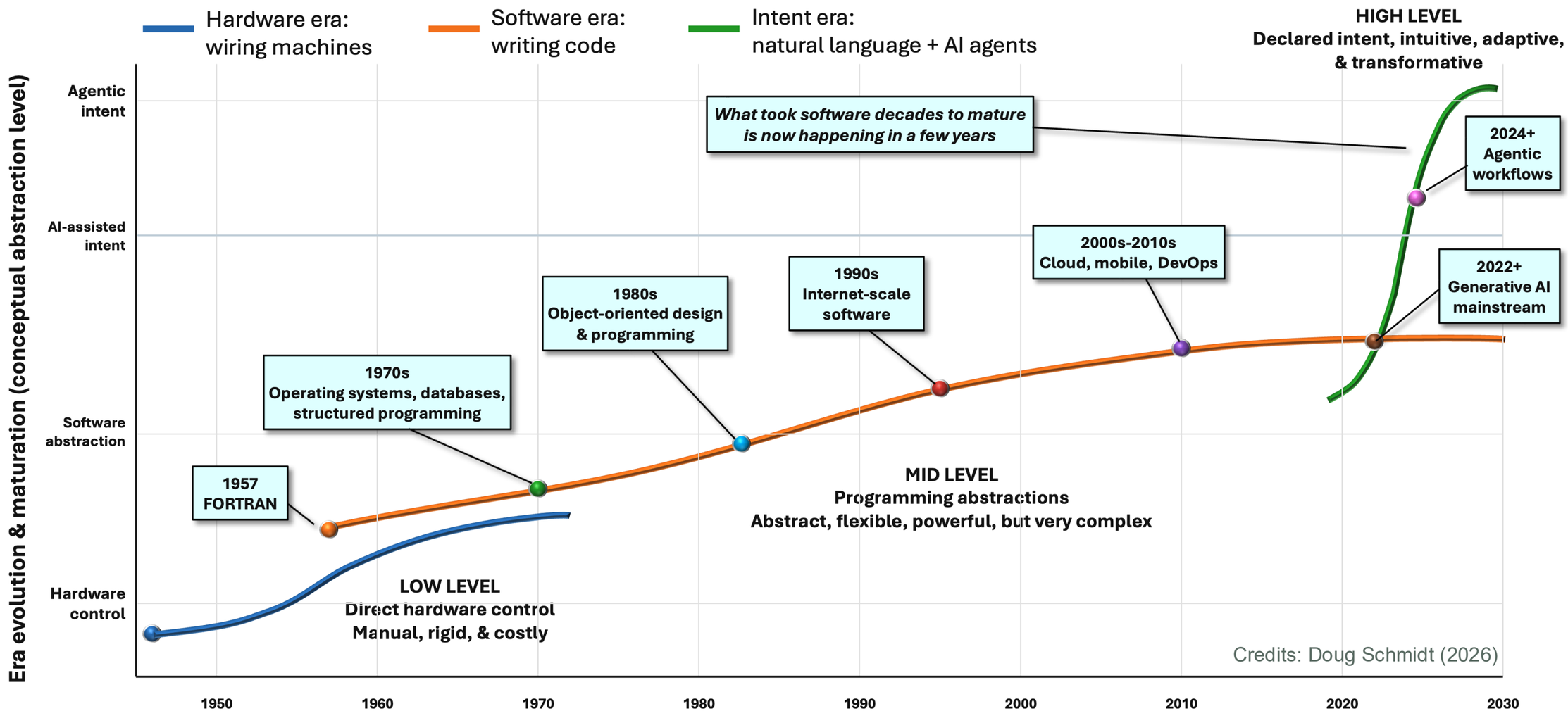
Internet & open source. Iterative/agile paradigms. Staged model. Unified Process. Agile Manifesto (2001).



2010s–now

Cloud & ecosystems. IoT. ML/AI. Agentic software engineering.

CS/SE evolution: from wires to intent



Conceptual graphic: y-axis visualizes increasing abstraction & maturity, not a measured index.

More on SE history...

Wirth, N., 2008. A brief history of software engineering. *IEEE Annals of the History of Computing* , 1(3), pp.32–39.

del Águila, I.M., Palma, J. and Túnez, S., 2014. Milestones in software engineering and knowledge engineering history: a comparative review. *The Scientific World Journal*, 2014.

Fiadeiro, J.L., 2004. Software services: scientific challenge or industrial hype? In *International Colloquium on Theoretical Aspects of Computing* , pp.1–13. Springer.

Rajlich, V., 2016. History of software engineering (Chapter 1). In *Software Engineering: the Current Practice* . Chapman and Hall/CRC.

Each paradigm shift responded to an anomaly.

Complexity → Waterfall. Volatility → Iterative / Agile.

Today's anomaly: the gap between what software must do and how fast humans can build and maintain it.

Agentic AI is the emerging response.

This is what we will cover in the next lectures

Software lifespan & the staged model

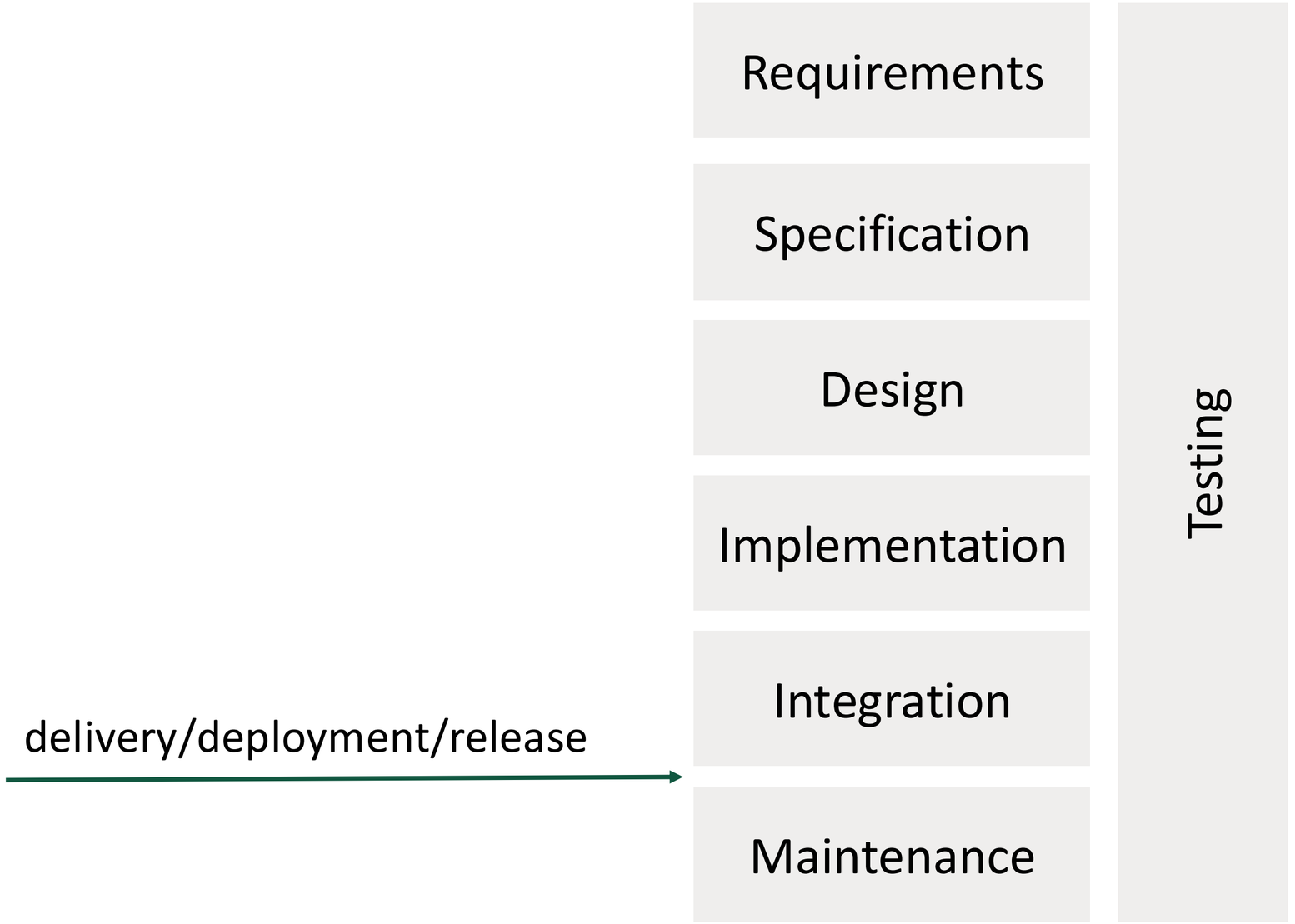
What happens to software after the first release?

Where does the effort actually go?

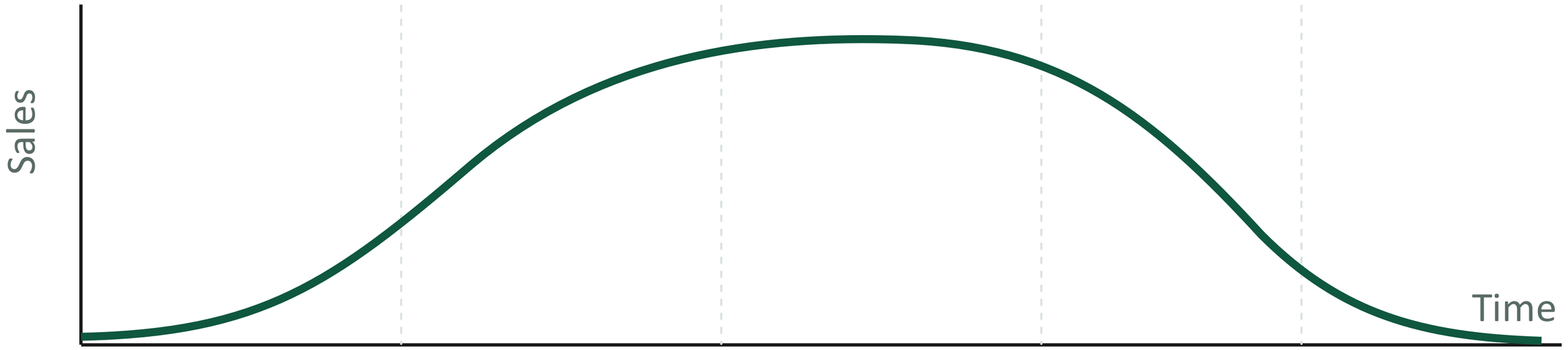
Building it is only the beginning.

Maintaining and evolving it is where most of the work lives.

Traditional software life span models

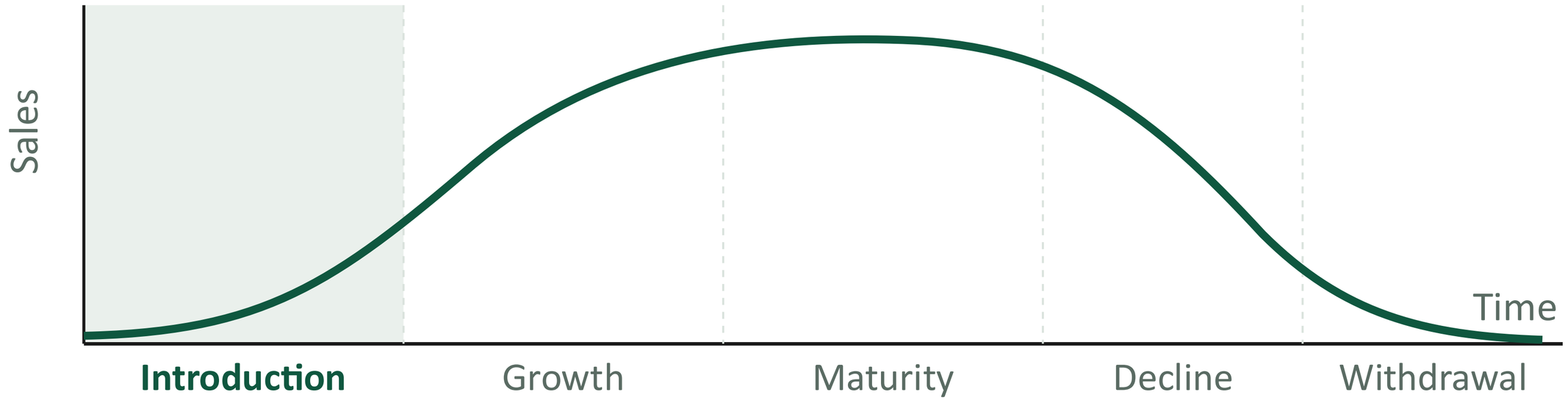


The product life cycle



Introduction	Growth	Maturity	Decline	Withdrawal
Low adoption, high investment	Rapid user adoption, adoption, many feature requests	Stable user base, incremental improvements	Usage falls, competition or obsolescence	Product discontinued discontinued

Product life cycle: Introduction



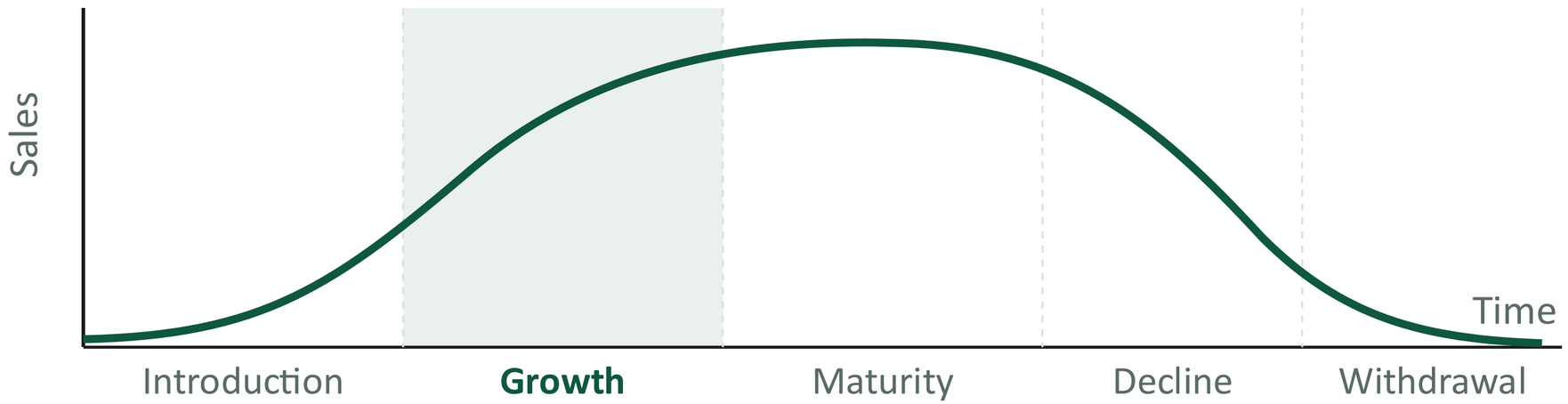
The product is conceptualized and initially developed.

The product is promoted to create awareness.

Limited numbers of the product.

Patents and trademarks are obtained.

Product life cycle: Growth



Market share tends to be stable.

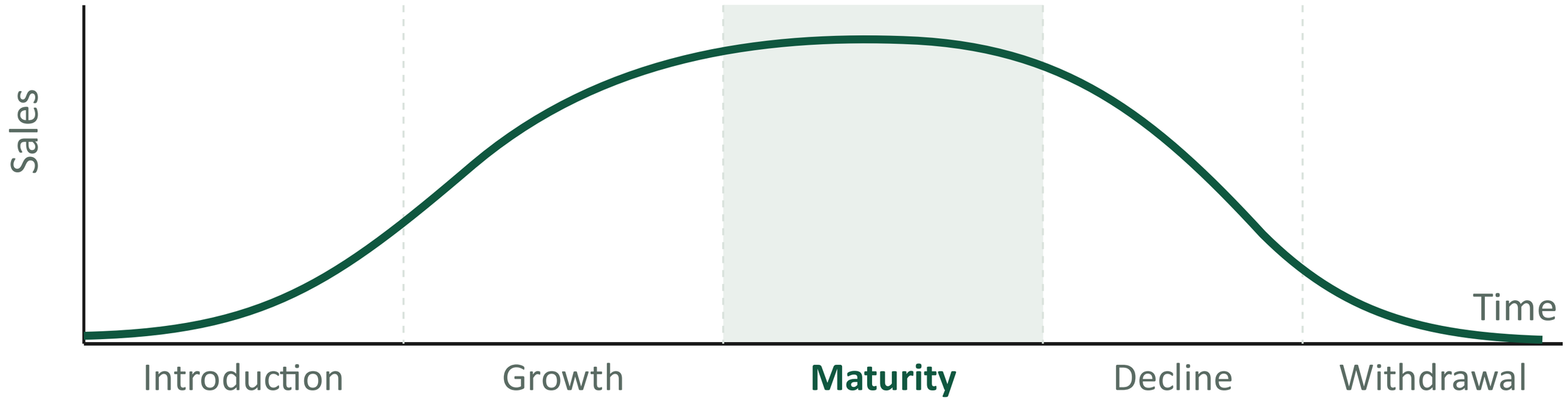
Product becomes more profitable.

Quality is maintained and improved.

Additional features may be added.

Competition might appear.

Product life cycle: Maturity



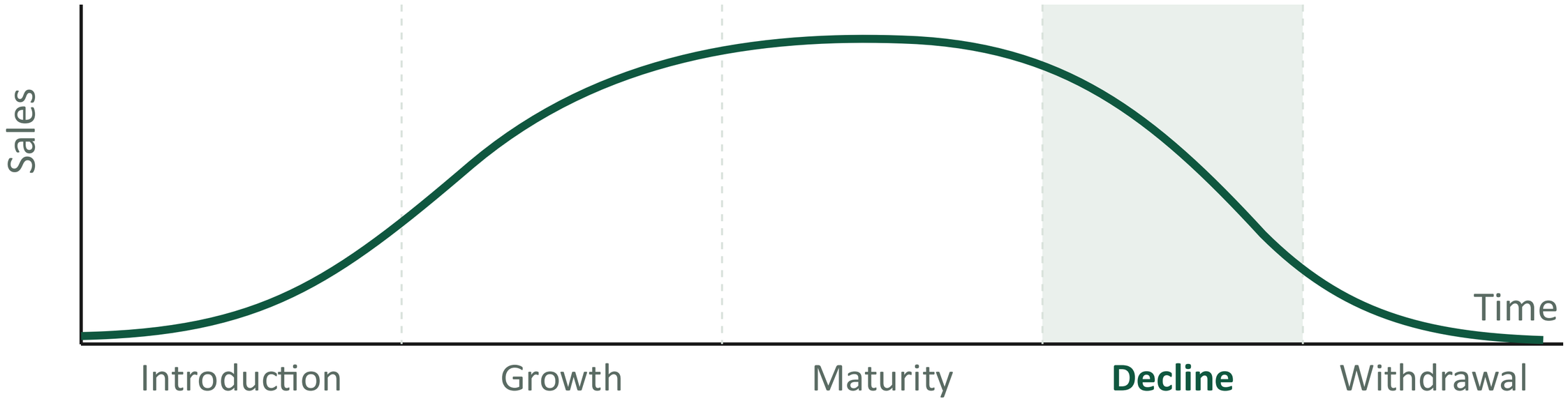
Sales grow at a decreasing rate and then stabilize.

Market saturation.

Product features may be enhanced to differentiate from competitors.

Promotion widespread.

Product life cycle: Decline

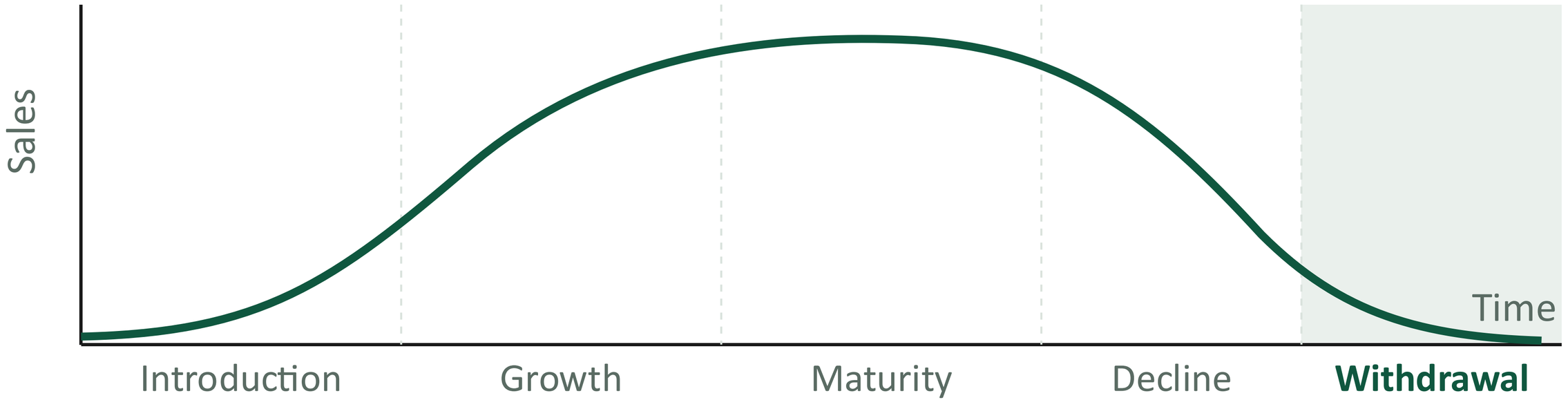


Market downturn.

Many more innovative products, or consumer tastes have changed.

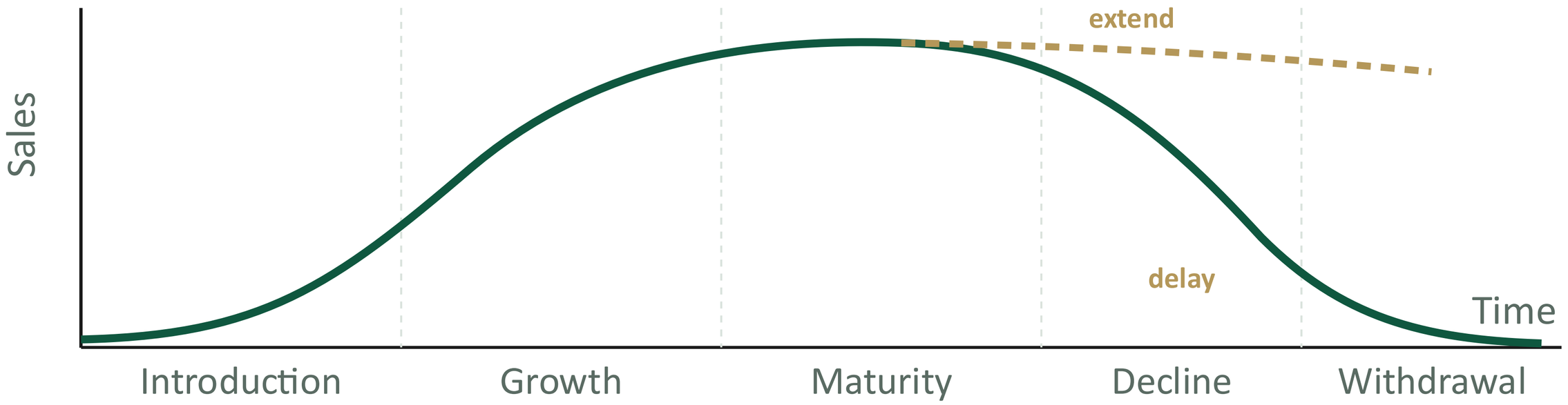
Management options: maintain the product, possibly by adding new features and finding new uses; reduce costs (e.g. marketing) and continue to offer it; sell the product to another company.

Product life cycle: Withdrawal



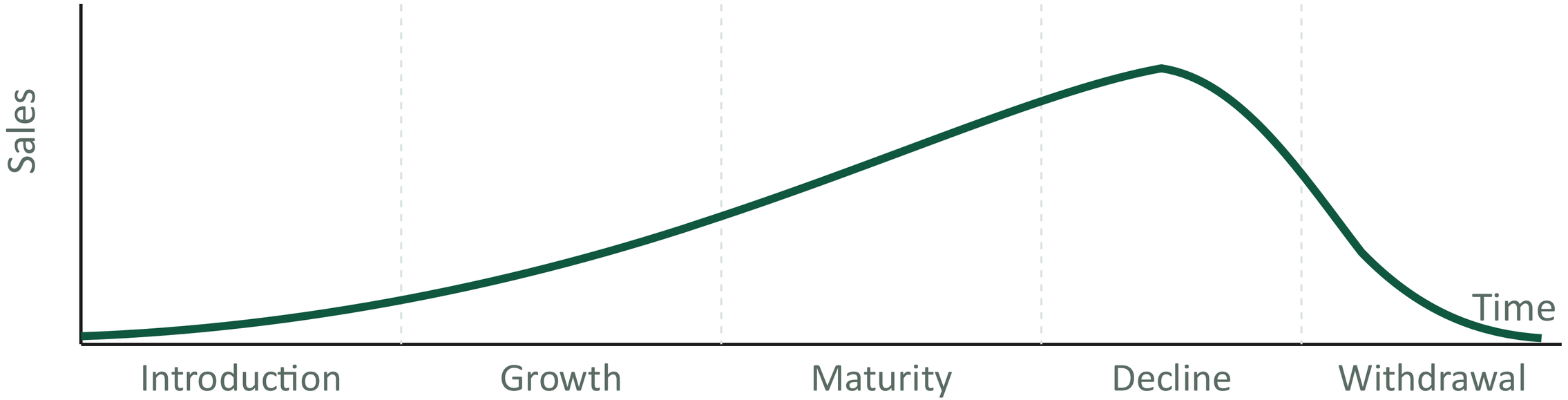
Discontinue the product and liquidate remaining inventory.

Extension strategies



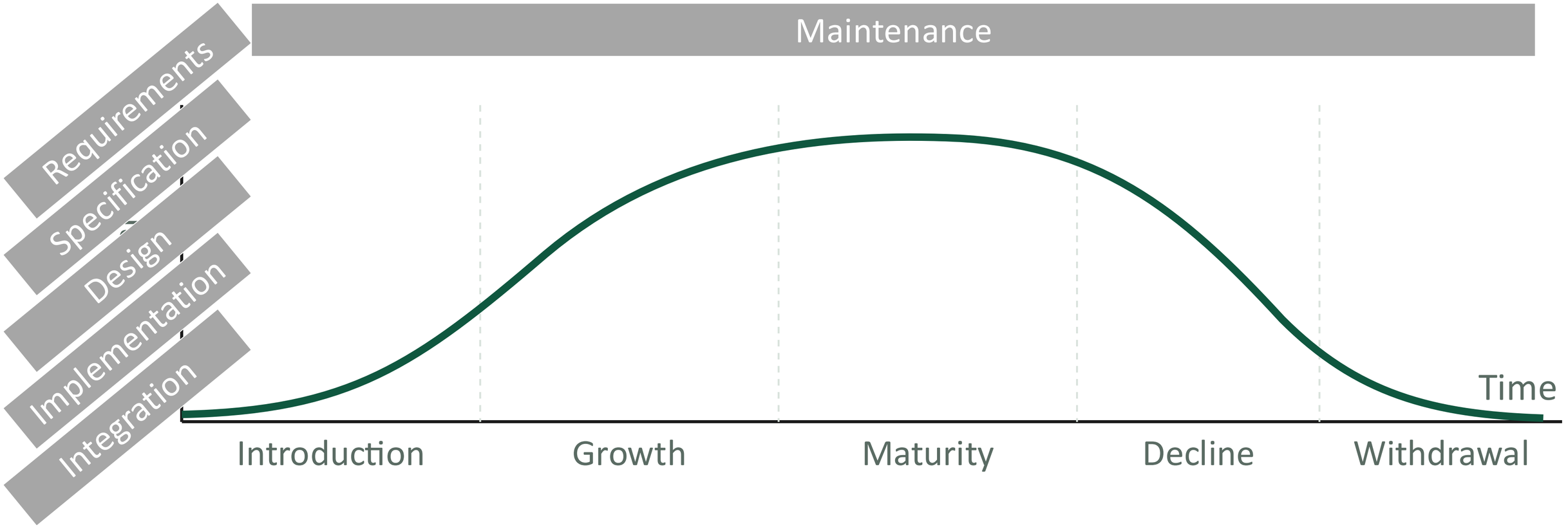
Extend maturity, delay decline — by adding features, entering new markets, or migrating platforms.

Reality

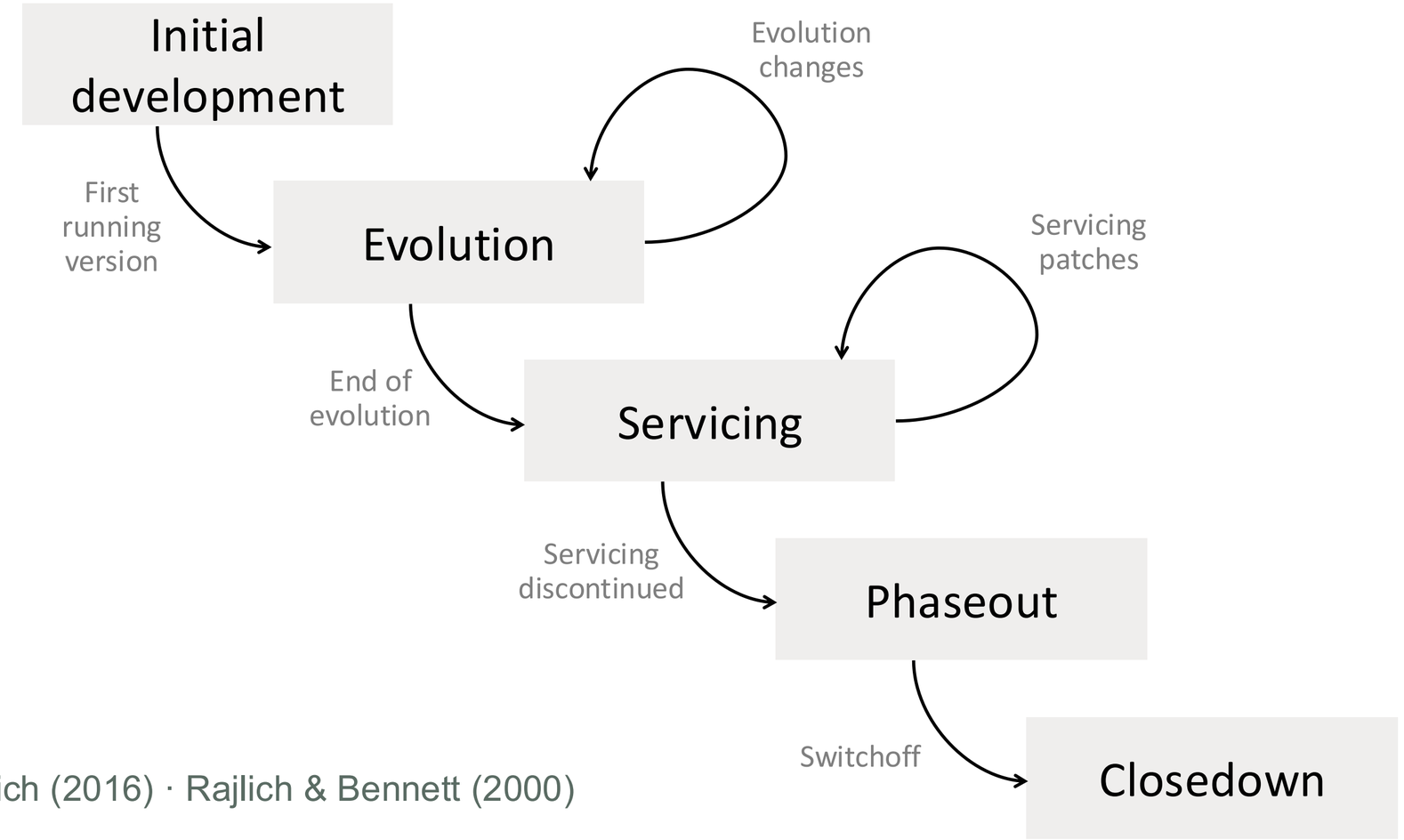


Same arc, none of the smoothness.

Traditional models vs. product life cycle



Staged model of software lifespan

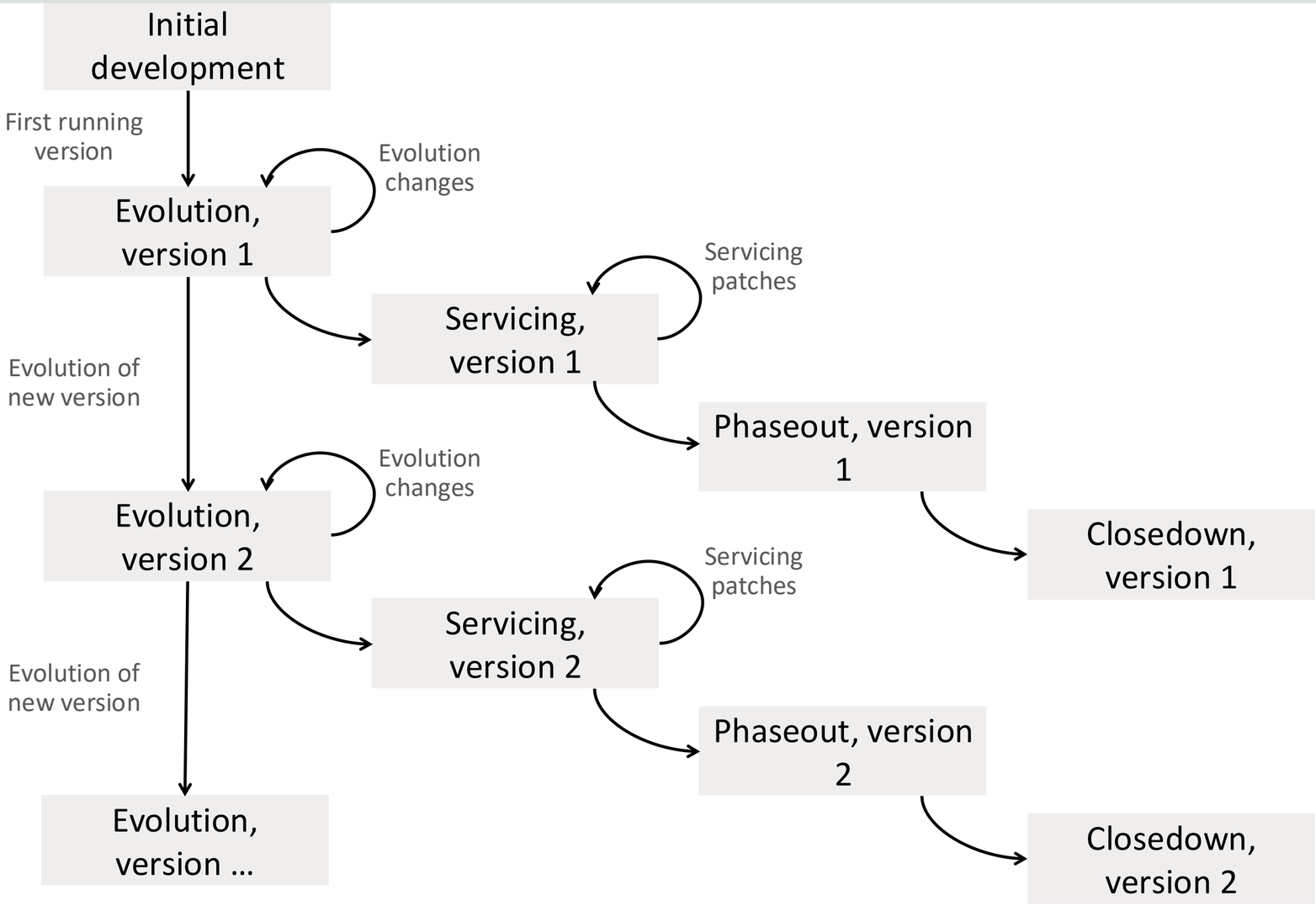


Rajlich (2016) · Rajlich & Bennett (2000)

The five stages

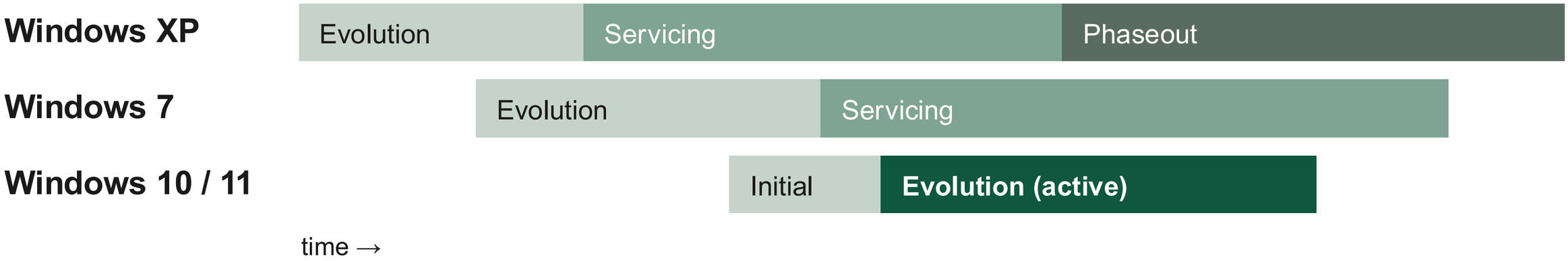
Initial development	Develop the first functioning version. Establish team expertise, architecture and technologies. Heavy investment, impending deadline.
Evolution	Adapt and correct capabilities based on experience with the software. Architecture and team knowledge make it possible; the architecture slowly loses its original integrity. Revenue is good. good.
Servicing	Coherent architecture or expertise are lost, limiting the scope of possible changes. Minor repairs, repairs, patches and wrappers only. The transition from evolution is often irreversible.
Phaseout	No more servicing; the architecture is no longer changeable. No changes in the code, though people may still use it.
Closedown	Withdraw the system and direct users to a replacement.

Versioned staged model



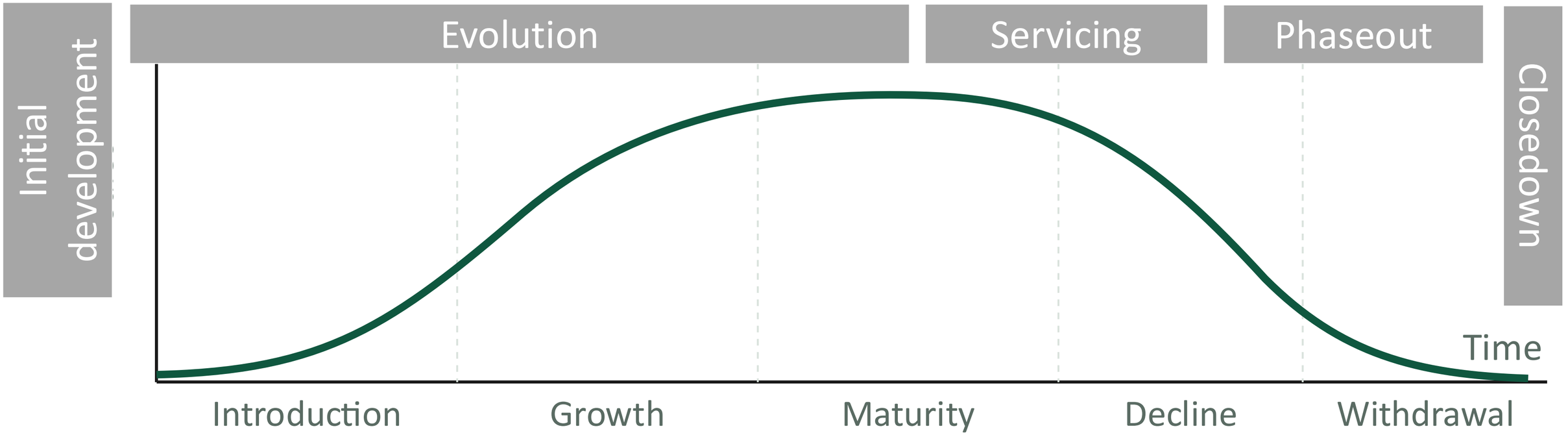
Versioned staged model

Multiple versions of the product progress through the stages simultaneously.



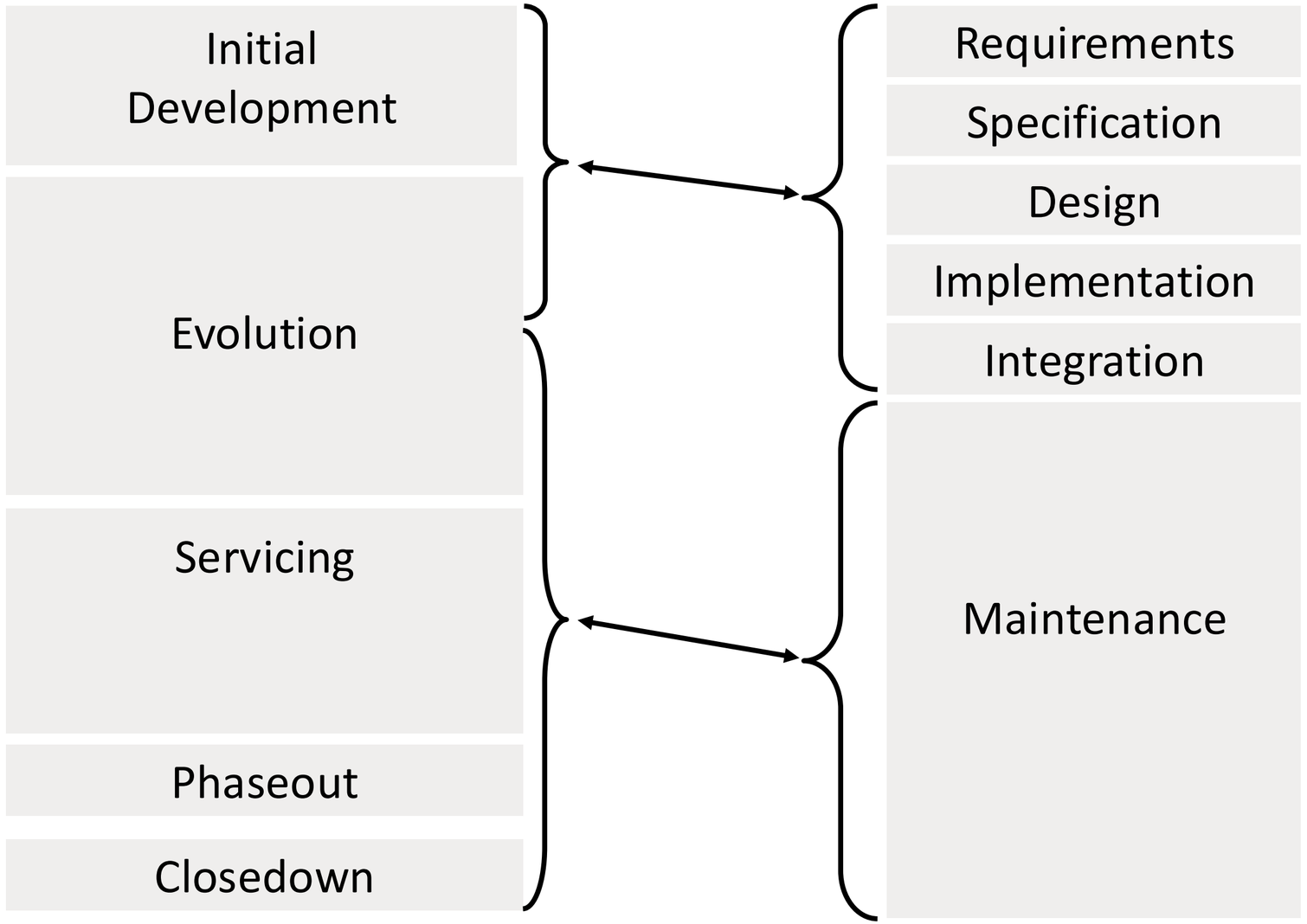
While one version is in Evolution, earlier versions are still being serviced — parallel workstreams.

Staged model vs. product life cycle



Software engineering focuses on the first three stages.

Staged model vs. traditional models



Where your sprints fit

Each sprint

You work on multiple issues.

Each issue

One incremental change.

The staged model is a product-level view. Incremental change is the issue-level process. They operate at different scales.

Incremental change

The process inside every issue you work on

Issues: the unit of change

Every change to software begins as an issue.

New feature

Add functionality that did not exist

Defect / bug

Fix incorrect behavior

Enhancement

Improve existing functionality

Task

Non-code work: documentation, configuration, setup

In your sprints, each issue you pick up is one incremental change.

Types of change

Rajlich & Gosavi (2004)

Incremental

Adds or modifies functionality. The most common type.

Contraction

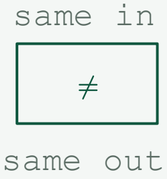
Removes obsolete functionality or code. Also called code pruning.

Replacement

Replaces a component entirely — same interface, new implementation.

Refactoring

Restructures code without changing observable behavior. Improves design.



Change type affects how you approach concept location and impact analysis.

Change impact

Local

One component or a handful of closely related ones. Consequences are predictable.

Significant

A medium but significant number of components. Rarer, but frequent enough.

Massive

Delocalized changes affecting a large part of the code. Expensive and risky.

Change strategy

Quick fix

Minimize time, accept technical debt. Critical bug in production, hard deadline.

Risk: shortcuts leave scars — cascading problems later.

Long-term investment

Clean solution, proper design. Feature work, evolution changes.

Trade-off: takes more time upfront.

Phases of software change

Interactions with the world

1 Initiation

SC design

2 Concept location

3 Impact analysis

SC implementation

4 Prefactoring

5 Actualization

6 Postfactoring

Interactions with the world

7 Conclusion

VERIFICATION

Software change: Initiation

1 **Initiation**

2 Concept location

3 Impact analysis

4 Prefactoring

5 Actualization

6 Postfactoring

↕ Verification — crosscuts 4–6

7 Conclusion

Software change starts with a **change request**.

Change requests come from a backlog of requirements (e.g. user stories) or issues (e.g. bug reports).

Created by users, developers, managers, …

They result from requirements elicitation.

Prioritization of change requests.

Assigning change requests to developers (e.g. bug triage).

Software change: design

1 Initiation

2 Concept location

3 Impact analysis

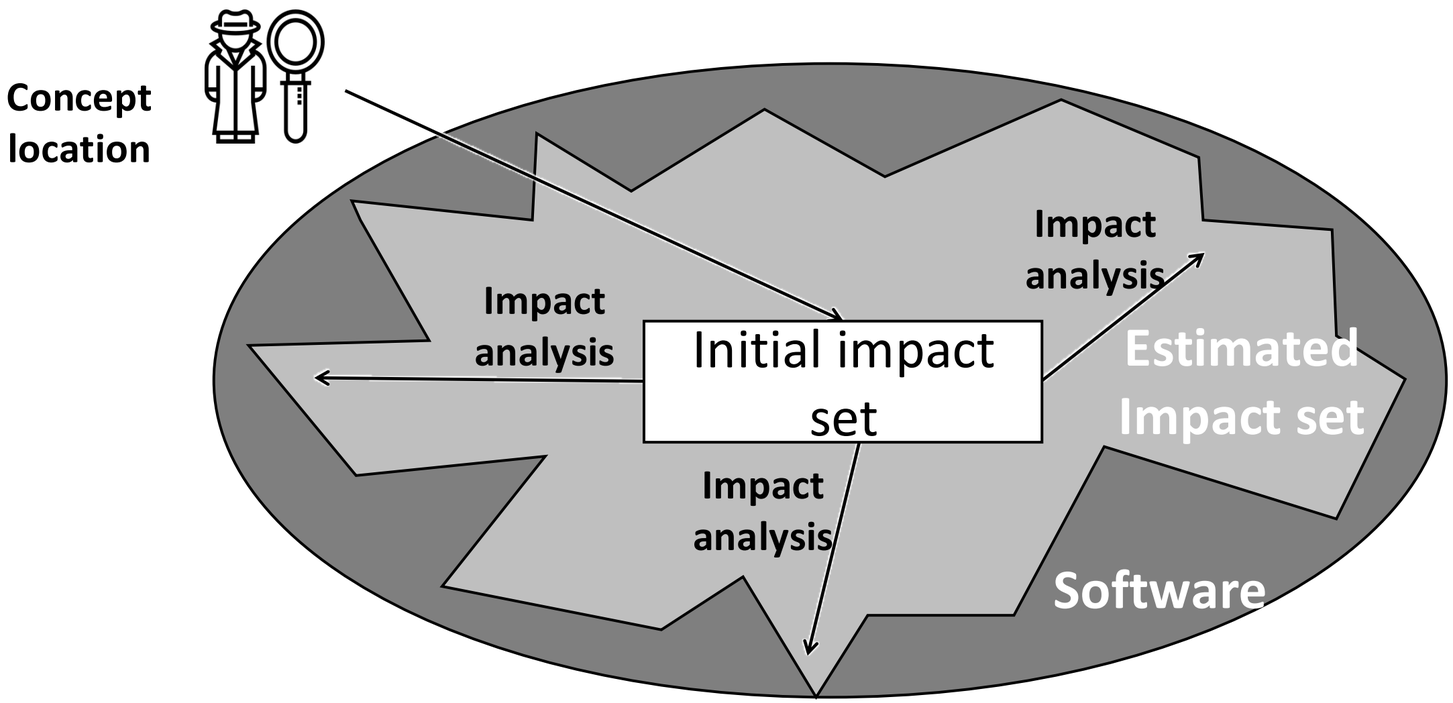
4 Prefactoring

5 Actualization

6 Postfactoring

↕ Verification — crosscuts 4–6

7 Conclusion



Concept location finds where in the software the concept lives. Impact analysis grows that into everything the change touches.

Software change: Concept location

1 Initiation
n

2 Concept location

3 Impact analysis

4 Prefactoring

5 Actualization

6 Postfactoring

↕ Verification — crosscuts 4–6

7 Conclusion

Concepts are extracted from the change request.

Change request analysis.

Concepts are located in the code and used as the starting point of the change.

Software change: Impact analysis

1 Initiation

2 Concept location

3 Impact analysis

4 Prefactoring

5 Actualization

6 Postfactoring

↕ Verification — crosscuts 4–6

7 Conclusion

Determine the strategy and impact of the change.

Classes and modules identified in concept location make up the **initial impact set**.

Class and module dependencies are analyzed, and impacted classes are added to the impact set.

Software change: Prefactoring

1 Initiation

2 Concept location

3 Impact analysis

4 Prefactoring

5 Actualization

6 Postfactoring

↕ Verification — crosscuts 4–6

7 Conclusion

Opportunistic refactoring that prepares the code for the change: it minimizes the impact of the change.

Extract class (Fowler): gather fields, methods and code snippets into a new class.

Extract superclass: create a new abstract class.

Software change: Actualization

1 Initiation

2 Concept location

3 Impact analysis

4 Prefactoring

5 Actualization

6 Postfactoring

↕ Verification — crosscuts 4–6

7 Conclusion

Create new code.

Merge it with the existing code.

Beginning with the class holding the new code, visit neighboring classes and update them.

Change propagation — similar to impact analysis.

Ripple effect.

Software change: Postfactoring

1 Initiation

2 Concept location

3 Impact analysis

4 Prefactoring

5 Actualization

6 Postfactoring

↕ Verification — crosscuts 4–6

7 Conclusion

Eliminate any anti-patterns (design bad smells) that may have been introduced.

Long method : after adding functionality, some methods may be doing too much.

Bloated / God class : after adding functionality, a class may be too large.

Software change: Verification

1 Initiation

2 Concept location

3 Impact analysis

4 Prefactoring

5 Actualization

6 Postfactoring

↕ **Verification — crosscuts 4–6**

7 Conclusion

Guarantee the correctness and quality of the change.

Testing: functional, unit, integration, …

Black box and white box testing.

Code reviews and walkthroughs.

Software change: Conclusion

1 Initiation

2 Concept location

3 Impact analysis

4 Prefactoring

5 Actualization

6 Postfactoring

↕ Verification — crosscuts 4–6

7 Conclusion

Commit finished code into the version control system.

Integrate your change with others from other developers.

Build the new baseline, update documentation, prepare a release, …

Prepare for the next change.

Test-driven development

Initiation → Concept location → **Test-first** → Actualization → **Regression testing** → Conclusion

Write the test first

The test states what the change must do before any code exists.

Write code to pass the test

Then refactor, with the test as the safety net.

Impact analysis, prefactoring and postfactoring do not disappear — they fold into the test-first and actualization steps.

What developers actually do

Decoding the Issue Resolution Process in Practice via Issue Report Analysis: A Case Study of Firefox

Antu Saha
William & Mary
Williamsburg, Virginia, USA
asaha02@wm.edu

Oscar Chaparro
William & Mary
Williamsburg, Virginia, USA
oscarch@wm.edu

***Abstract*—Effectively managing and resolving software issues is critical for maintaining and evolving software systems. Development teams often rely on issue trackers and issue reports to track and manage the work needed during issue resolution, ranging from issue reproduction and analysis to solution design, implementation, verification, and deployment. Despite the issue resolution process being generally known in the software engineering community as a sequential list of activities, it is unknown how developers implement this process in practice and how they discuss it in issue reports. This paper aims to enhance our understanding of the issue resolution process implemented in practice by analyzing the issue reports of Mozilla Firefox. We qualitatively and quantitatively analyzed the discussions found in 356 Firefox issue reports, to identify the sequences of stages that developers go through to address various software problems. We analyzed**

Understanding the issue resolution process implemented in practice is important for improving software maintenance and evolution processes. By gaining insights into how developers address software problems, we can identify bottlenecks and anomalous process implementations, align prescribed processes with actual practices, and provide developers with better guidance for issue resolution. Additionally, studying issue resolution can help identify common patterns and strategies that can be applied to similar problems in the future, and confirm the extent to which the implemented process deviates from the typical, linear resolution process from the literature.

This paper aims to enhance our understanding of the issue resolution process implemented in practice by identifying and

Saha & Chaparro, “Decoding the Issue Resolution Process in Practice via Issue Report Analysis: A Case Study of Firefox,” ICSE 2025

The issue resolution process

FIREFOX		RAJLICH
1 Reproduction	→	no equivalent phase
2 Analysis	→	1 Initiation + 2 Concept location + 3 Impact analysis
3 Solution design	→	2 Concept location + 3 Impact analysis
4 Implementation	→	4 Prefactoring + 5 Actualization + 6 Postfactoring
5 Code review	→	Verification
6 Verification	→	Verification
no equivalent stage	←	**7 Conclusion**

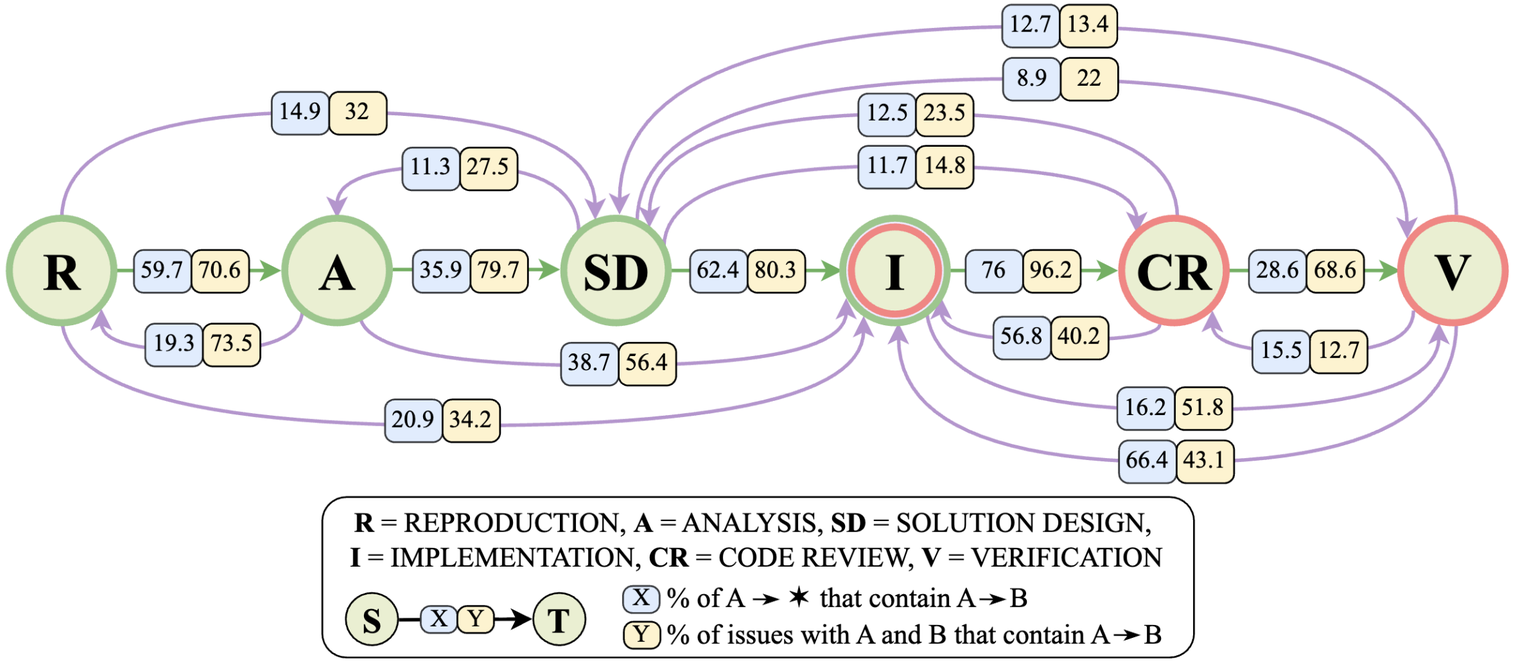
Issue resolution in practice

The process is iterative, flexible, and team-dependent.

Developers skip stages, repeat them, and go back — it is not linear!

47

distinct issue resolution patterns from six stages



In each sprint, you will work on multiple issues.

Each issue goes through its own incremental change cycle — some short, some long, some that loop back.

The process is a guide, not a script.

Concept location & impact analysis

Two phases you'll use in every sprint to work on every issue — phases 2 and 3 of incremental change

The role of concept location

Concept location is needed whenever a change is to be made.

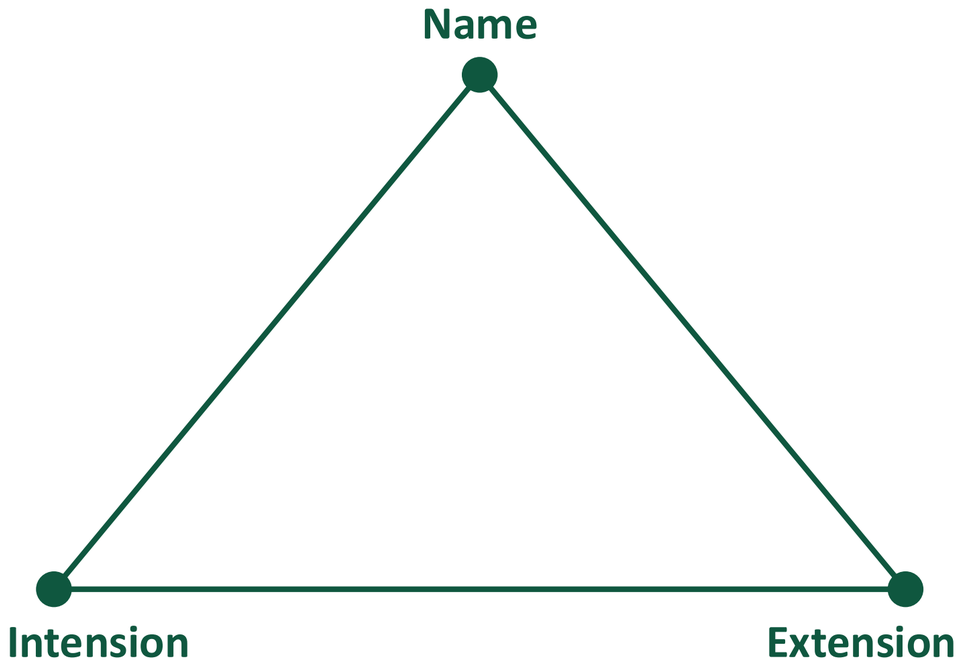
Change requests are most often formulated in terms of **domain concepts**.

> Example: *“Correct error that arises when trying to paste a text.”* The programmer must find in the code the locations where the concept **paste** is implemented. This is the start of the change.

The discourse about the **domain**	↔	The **software**

Two worlds. Concept location is the bridge between them, through concepts.

The concept triangle



Name

What we call the concept — “shopping cart,” “payment gateway”

Intension

What the concept means — “physical or digital basket to hold items/goods”

Extension

Where it is implemented in the code — classes, methods, files

Goal of concept location: start with a name, find the extension.

Spelling corner

Intension

`| inˈtenSHən | syn. connotation`

The suggesting of a meaning by a word apart from the thing it explicitly names.

An essential property, or group of properties, of a thing named by a term.

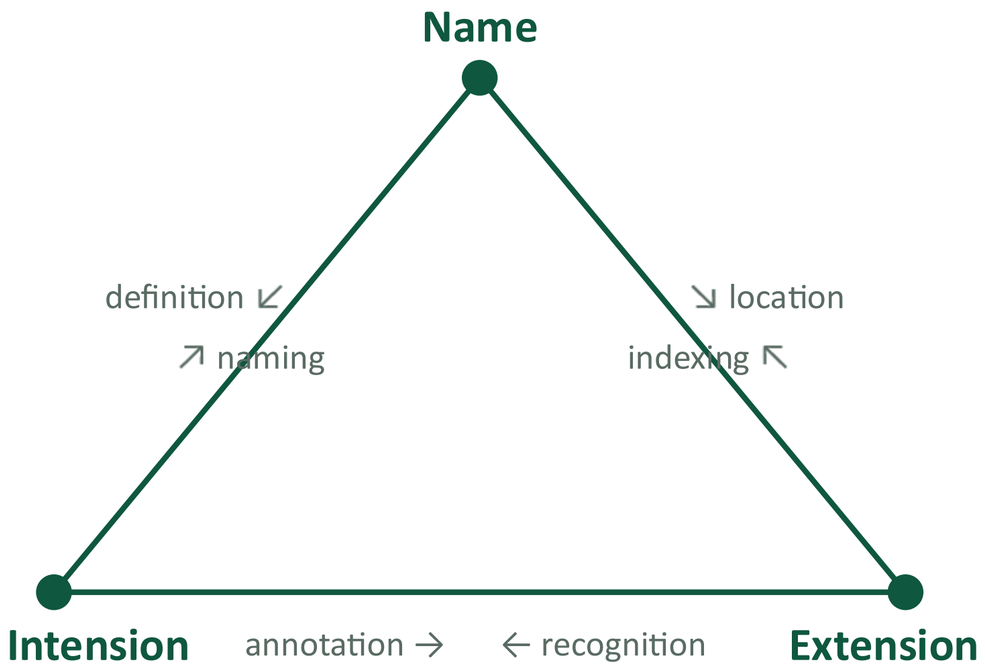
Intention

`| inˈten(t)SH(ə)n | syn. intent, purpose, aim`

What one intends to do or bring about.

Not the word we want here.

Relationships between facets



Relationships are the processes of software comprehension.

naming · definition — between name and intension

location · indexing — between name and extension

annotation · recognition — between intension and extension

And none of it is clean: homonyms, synonyms, and stale names all sit on these edges.

Why concept location is hard

Concept extensions are implemented as **code fragments** : variables, classes, methods, functions, commands, database tables, …

Concept location is a search for those specific fragments.

Easy

In small programs, or programs the programmer knows well.

Hard

In large programs, or programs the programmer does not know.

Names are ambiguous, the same code may implement several concepts, and documentation may be stale or absent.

Concept location methodologies

Human knowledge

Ask someone who already knows the code.

Traceability information & tools

Links from requirements and issues to code.

Static search

Text search over source code and documentation.

Dependency search over source code.

Dynamic search

Execution traces, debugging, instrumentation.

The next two slides are the static ones.

How to find the code

Text search (grep)

Search for keywords related to the concept. Fast — but incomplete: it misses synonyms, indirect references, dynamic dispatch.

```
$ grep -rn "ClassNode" violet/
ClassNode.java:24  · ClassDiagramGraph.java:61  · ClassRelationshipEdge.java:12
```

Is this complete? What could it miss?

How to find the code

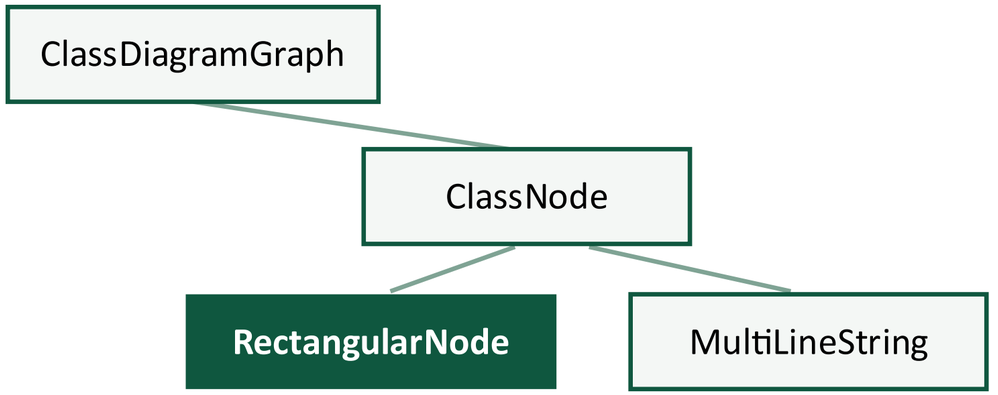
Text search (grep)

Fast, but incomplete: misses synonyms, indirect references, dynamic dispatch.

```
$ grep -rn "ClassNode" violet/
ClassNode.java:24 ClassDiagramGraph.java:61
ClassRelationshipEdge.java:12
```

Dependency search

Starting from a known class, follow call graphs and import relationships. Finds indirectly related code that text search misses.



Text search + dependency search = systematic concept location.

Concept location as a search process

1. Understanding the the problem
2. **Formulating a query**
3. Selecting a search strategy
4. Executing the search
5. Analysis of results

↺ Analysis of the results sends you back to a new query.

the cycle repeats until the extension is found.

Formulating a query

Extract the set of concepts used in the change request.

Delete concepts intended for **communication with programmers**.

Delete concepts **unlikely to be implemented in the code** : things outside the scope of the program, and things not yet implemented.

Rank the remaining concepts by the likelihood that they can be easily located.

Example ::: point-of-sale system

CHANGE REQUEST

Implement a credit card payment

Example ::: point-of-sale system

CHANGE REQUEST

Implement a credit card payment

The system only supports debit card and cash payments.

Example ::: point-of-sale system

CHANGE REQUEST

Implement ~~a credit card payment~~

The system only supports debit card and cash payments.

Communication with the programmer

Delete — it is an instruction, not a concept.

Yet to be implemented

Delete — it is not in the code, that is the point of the request.

Significant concept

Keep — this is what you search for.

Impact analysis: what else will change?

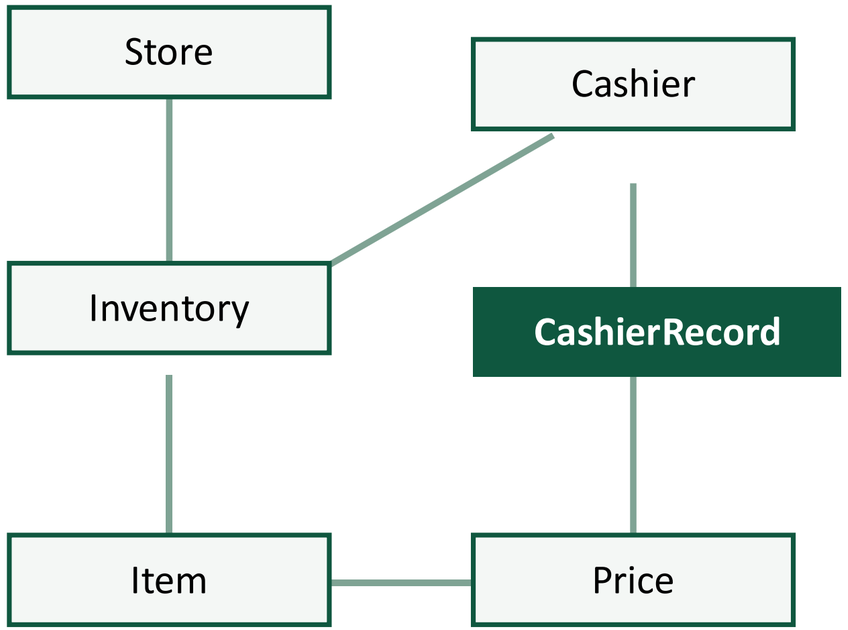
Once you know what to change, you need to know what else that might break.

Class (or method) interaction graph

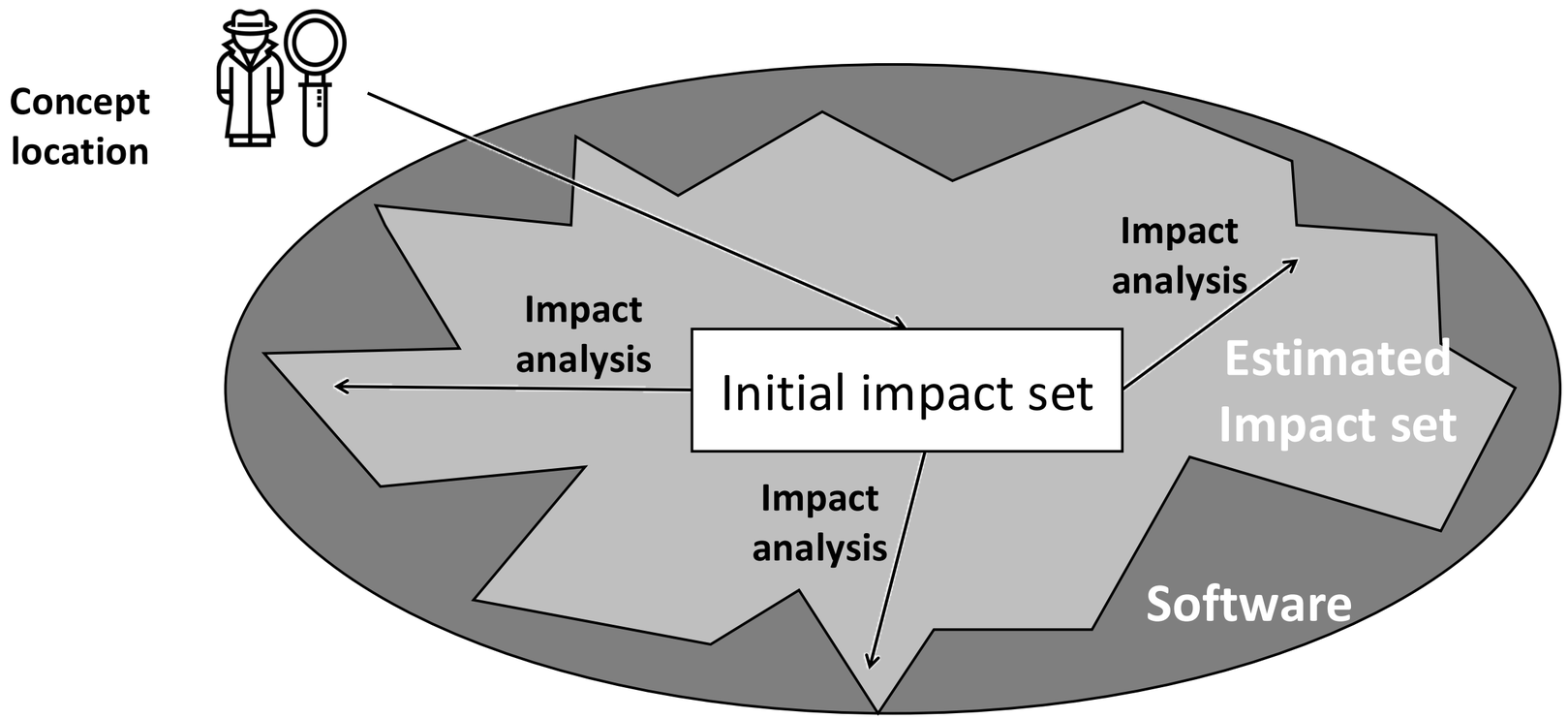
Nodes are classes or components. Directed edges are method calls and field accesses between them.

Starting from the changed class, we propagate through the graph to find what else is affected.

Example: a point-of-sale system, change request “record cashier sessions.”



Initial and estimated impact set



Concept location produces the **initial impact set** — the classes where the concept lives.

Impact analysis follows dependencies outward to produce the **estimated impact set**.

The iterative marking algorithm

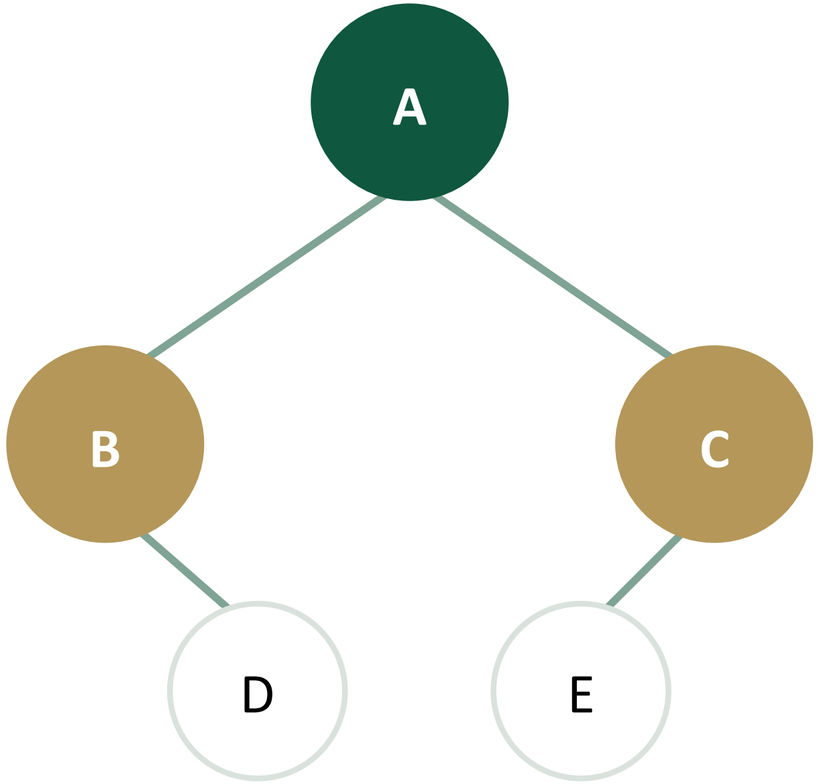
- **Blank** — never inspected, not scheduled
- **Changed** — will definitely change
- **Unchanged** — inspected, no impact
- **Next** — scheduled for inspection
- **Propagating** — does not change itself, but its neighbors may

The iterative marking algorithm

Step 1. Mark the changed class as Changed.

Step 2. Its direct neighbors become Next.

What gets marked next?

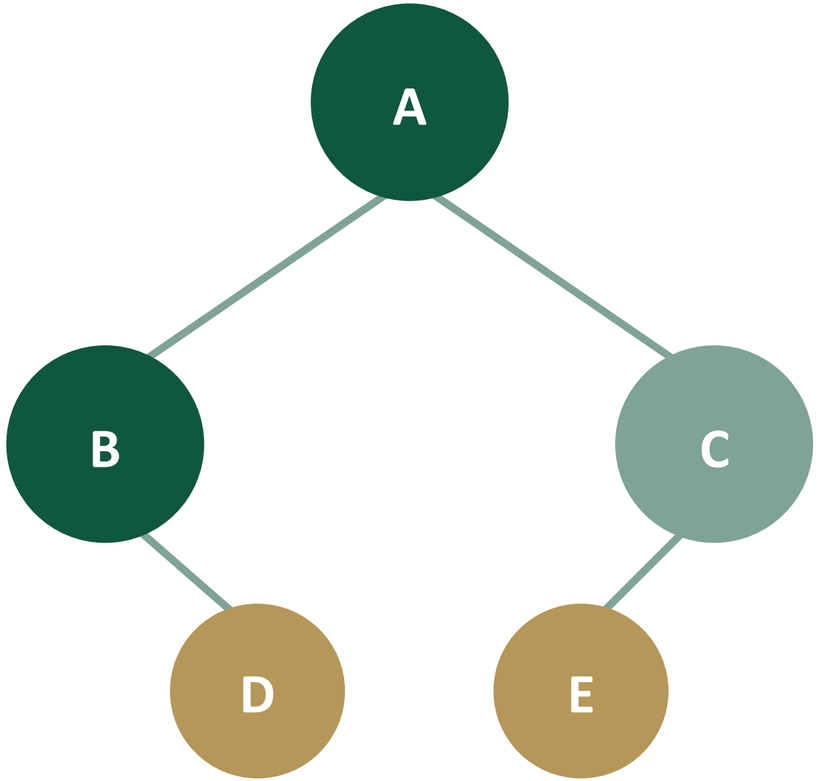


The iterative marking algorithm

Step 1. Mark the changed class as Changed.

Step 2. Its direct neighbors become Next.

Step 3. Process the Next nodes — mark each as Changed, Unchanged, or Propagating, then schedule its blank neighbors.



Alternatives in software change

A program displays a temperature in Fahrenheit. **Change request: display the temperature in Celsius.**

Where the sensor data is converted

Convert once, at the source.

Where the temperature is displayed

Convert at the point of presentation.

Two separate locations deal with temperature. The change can be done in either place — impact analysis weighs the alternatives.

The criteria

Required effort

It is easier to adjust the user interface.

Clarity of the resulting code

It is better to have all calculations of the temperature in one place.

Very often these criteria contradict each other: the easier change leads to less clarity. A conflict between short-term and long-term goals.

Underestimated impact set

Impact analysis estimates which classes are impacted.

Change propagation modifies the code of those impacted classes.

Change propagation is the moment of truth: it confirms or refutes the predictions of impact analysis.

The accuracy of impact analysis predictions is what software managers plan against.

Underestimation

Common in software engineering.

A consequence of **invisibility** — one of the essential properties from the start of the lecture.

It makes planning difficult.

Common in other fields also.

Rajlich, V. and Gosavi, P., 2004. Incremental change in object-oriented programming. *IEEE Software*, 21(4), pp.62–69.

Key takeaways

1. Software's essential properties — changeability, invisibility, complexity, conformity, discontinuity — make it fundamentally different from physical engineering.
2. SE paradigms evolved in response to anomalies: complexity → waterfall; volatility → iterative/agile.
3. Successful software lives a long life — the staged model describes product evolution over years or decades.
4. Every issue you work on follows the incremental change cycle of 8 phases. Not rigid — a vocabulary.
5. The real process is flexible and iterative — Firefox: 47 patterns from 6 stages.
6. Phases 2 and 3 — concept location and impact analysis — are your analytical tools before coding.

Sprint 0

1 · Initiation	2 · Concept location	3 · Impact analysis	4 · Prefactoring	5 · Actualization	6 · Postfactoring	7 · Verification	8 · Conclusion

Which of these phases will you actually do in Sprint 1?

EXPECTED

Initiation · Concept location · Impact analysis · Actualization · Verification

Goal of Sprint 0: get set up so you can execute all of these phases efficiently.