Software Development Lifecycle & Process Models

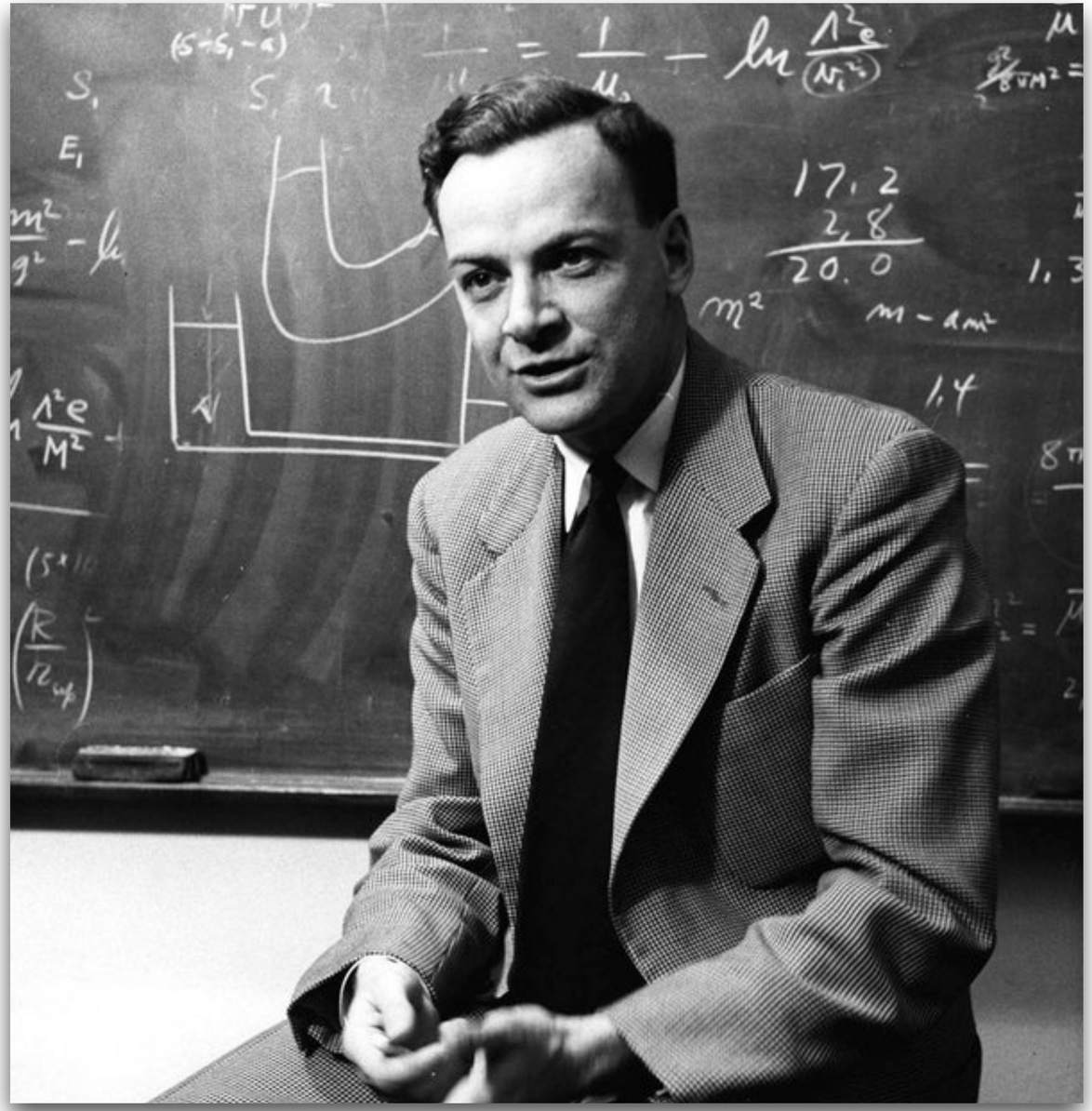
CS350 Introduction to Software Engineering

Shin Yoo

Let’s build a high quality SW.

Feynman Method

- Write down the problem.
- Think real hard.
- Write down the solution.



So we have decided to build a quality SW.

Now what? :)

We will take a 2-tier approach.

- First, break down SW development process into common stages: each stage serves a specific purpose in the whole process of developing a software system.
- Second, organise the stages systematically, so that one knows what follows what.

Software Development Lifecycle

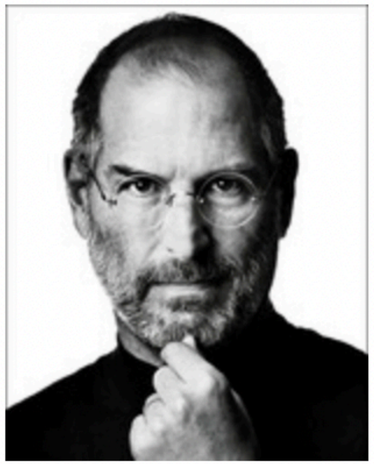
Let’s break down things we need to do into different stages

- Major stages in all software engineering projects:
  - Requirements
  - Design
  - Implementation
  - Testing
  - Maintenance
- Each stage raises different questions and problems.

Requirements

A very broad question of “what do we *exactly* want to build?”

- Some activities are human
  - Elicitation: what do the u
  - Analysis and negotiation immediately. Also, what
- Other activities focus on fo
  - Formal specifications: logical/mathematical representation of what your software needs to do.



“Some people say, "Give the customers what they want." But that's not my approach. Our job is to figure out what they're going to want before they do. I think Henry Ford once said, "If I'd asked customers what they wanted, they would have told me, 'A faster horse!'" People don't know what they want until you show it to them. That's why I never rely on market research. Our task is to read things that are not yet on the page.”

— Steve Jobs

Example: Linear Temporal Logic (LTL)

- LTL is a modal logic system that can describe a changing world (i.e., linear progression of time)
  - Two modalities, $\Box P$ (P is always true) & $\Diamond Q$ (Q is sometimes true), combined with a healthy dose of theory gives an expressive system that allows us to describe temporal behavior of systems. For example,
  - $\Box(lost(x) \Rightarrow \neg onShelf(x))$: as a rule, if a book is lost, it cannot be on the shelf
  - $\Box(request \Rightarrow \Diamond response)$: if a request is made, at some point a response should be given
- Why? To automatically verify properties (i.e., formal verification)

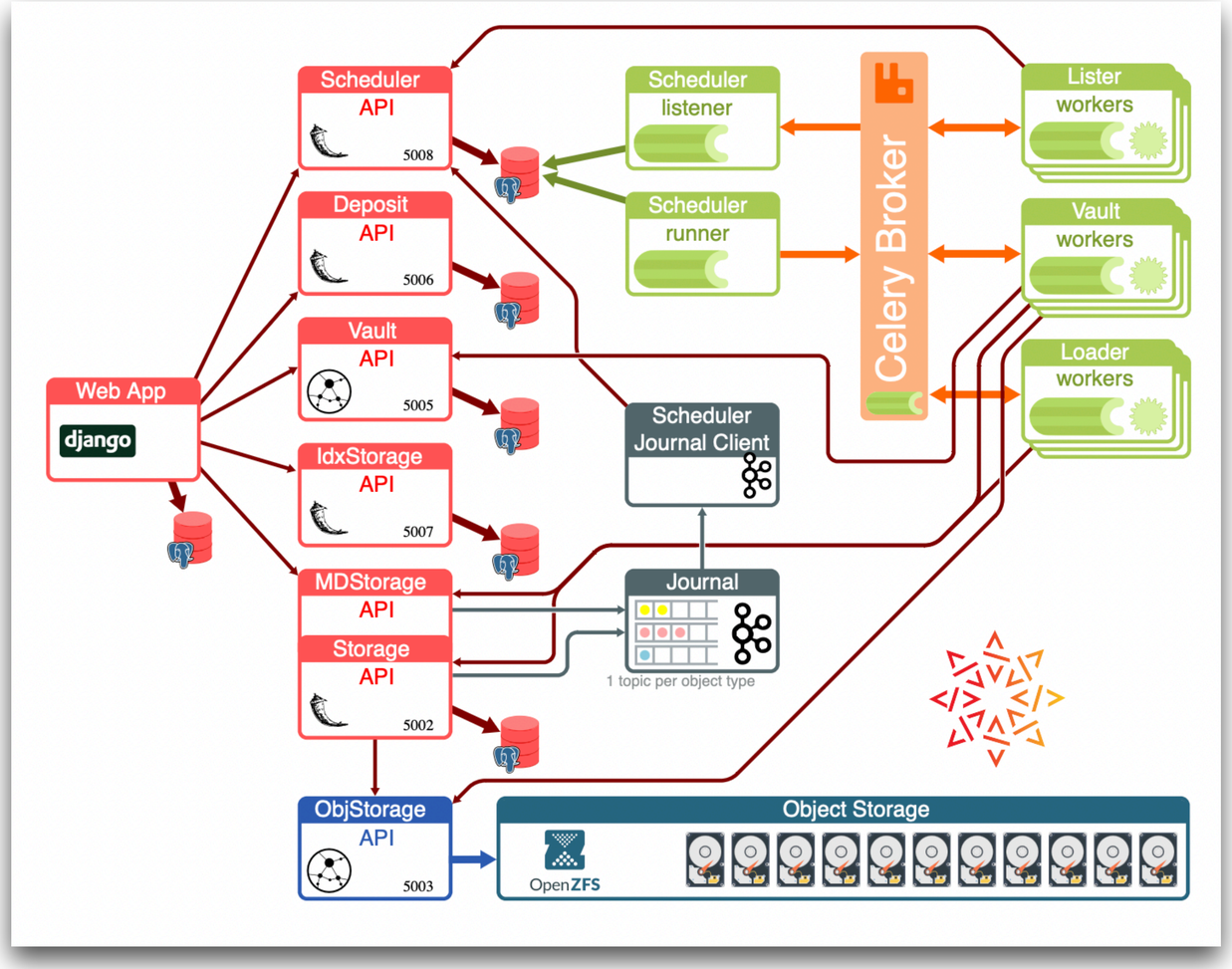
Design

“Architecture is the important stuff. Whatever that is.” - Martin Fowler

- Given the technology and building blocks we have, how do we best satisfy the requirements, while also meeting quality criteria for good software?
  - System Architecture: what is the overall structure of the entire system that can best handle the given requirements?
    - How to model the real world data?
    - What is the general paradigm (e.g., monolithic vs. microservice, native vs. client/server…)?
    - Which component/technology to include/depend on?
- A large portion of academic effort also went into languages that can clearly express design, e.t., UML.

Architecture Example

Software Heritage Project

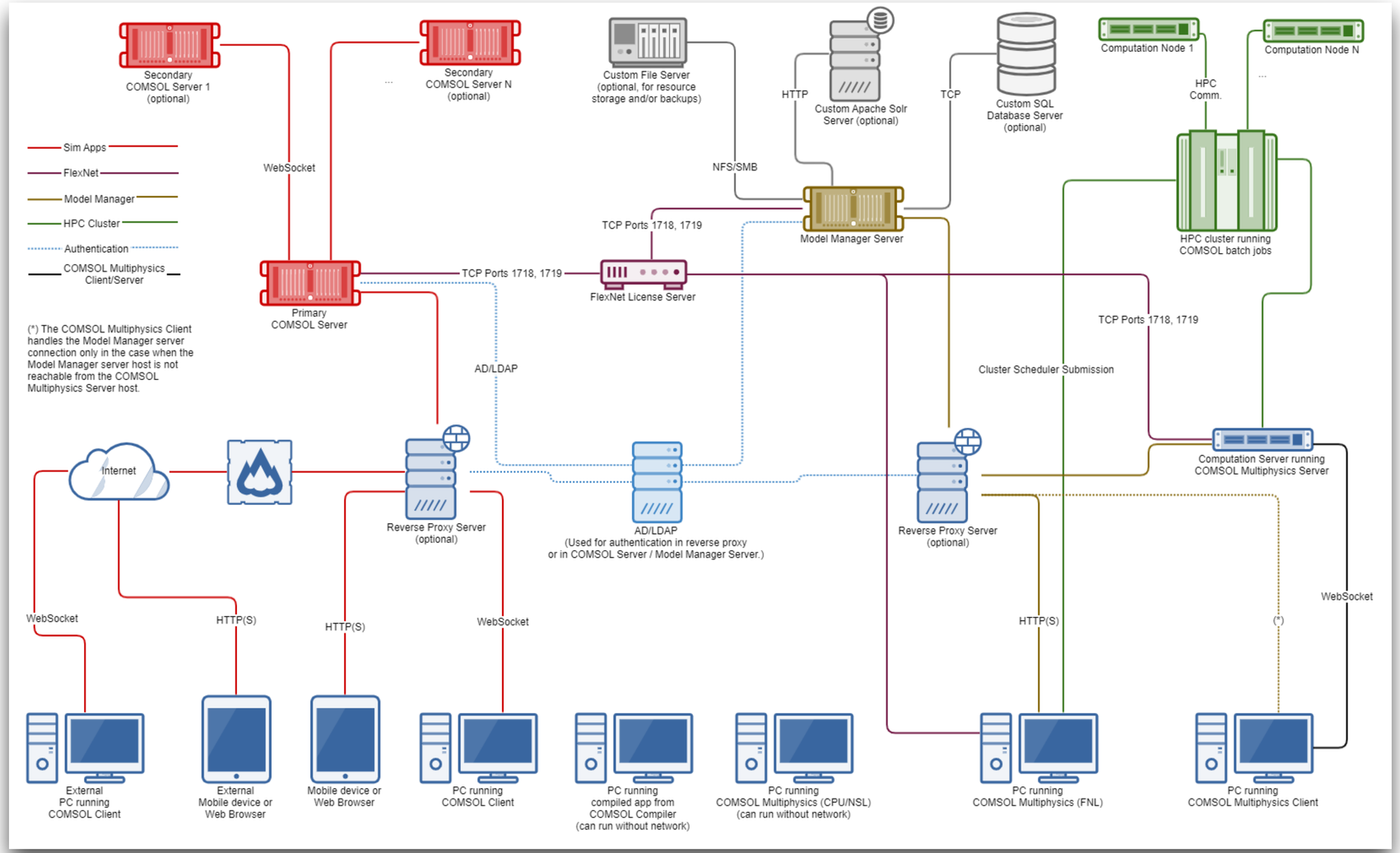


This diagram shows core software components, to communicate how the main task (archiving software projects) is broken down to smaller steps.

https://docs.softwareheritage.org/devel/architecture/overview.html

Architecture Example

COMSOL (Scientific Simulation Software)



This diagram shows how the participating hardware components are connected, with which roles.

https://www.comsol.com/support/knowledgebase/1299

Implementation

…where you write lots of code

- Strangely, this is where things get very personal for many of us: pick your favorite trolling war!
  - Emacs vs. vi(m) vs. VS Code, keyboard vs. mouse, C vs. Java vs. Python vs. JavaScript vs. Haskell vs. Rust, …
- Programming Paradigm (tightly coupled with design stage): procedural, object-oriented, functional…
- Programming Environment (IDEs)
- Programming Assistants (code completion, AI models…)
- Attempts to *derive* implementation from design automatically (from Model Driven Engineering to No Code/Low Code movement)

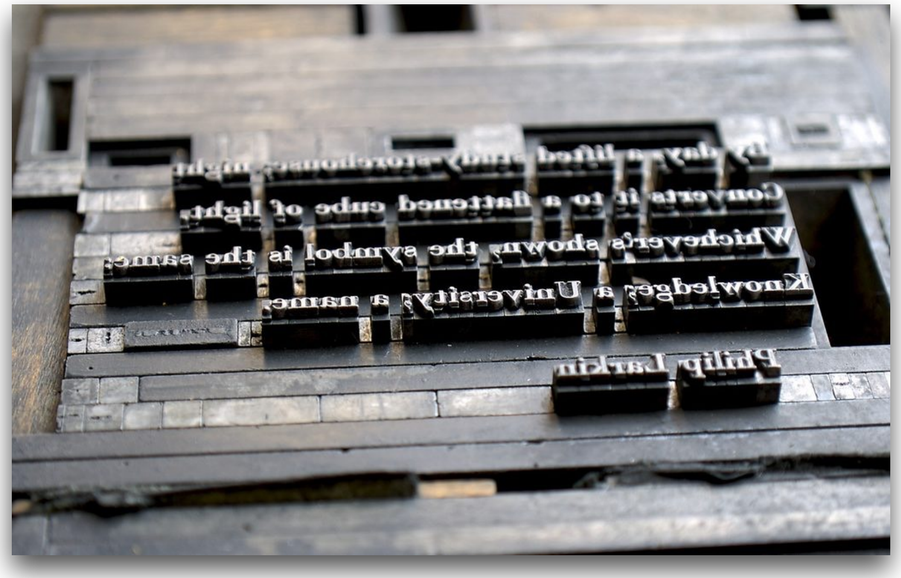
Challenges in Implementation

Why is it hard? Or at least not easy?

- In many cases, what we want to achieve with code is definitely achievable (i.e., we are not trying to resolve the halting problem…).
- Then why is it difficult?
  - Problem of acquiring relevant information (StackOverflow)
  - Problem of variability (no one has tried it on this particular environment)
  - Problem of unclear/incorrectly understood specification
  - Problem of speed (need it yesterday)
- In total, we must be writing so much boilerplate code :)

Boilerplate

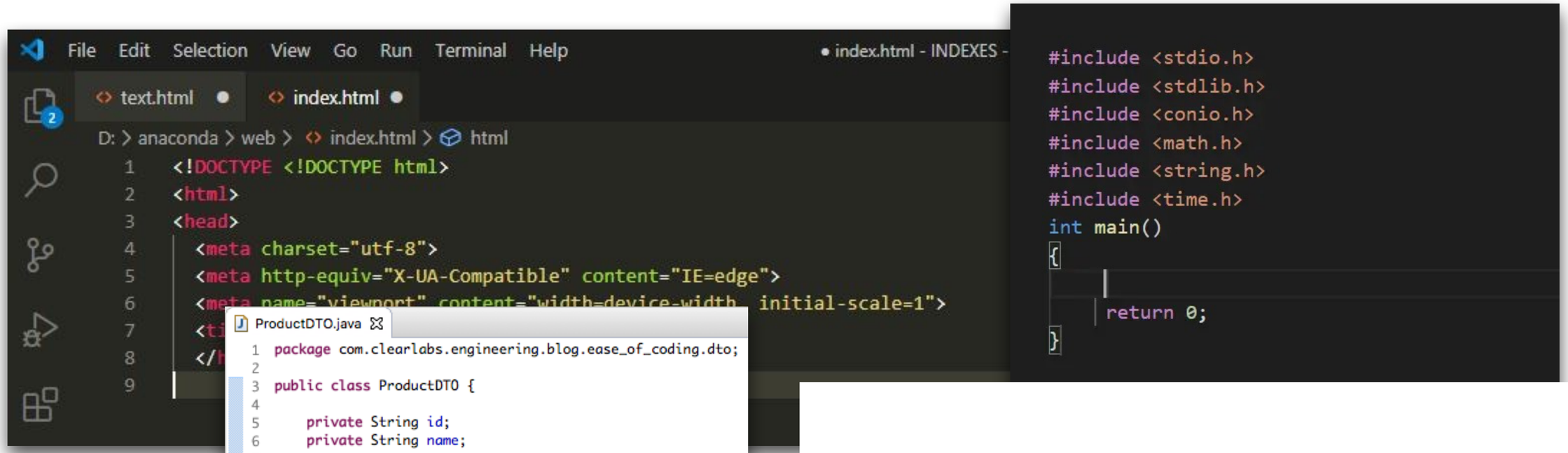
- (originally) rolled sheet of iron, with which boilers for steam engines are made
- (in newspaper) a plate with fixed texts (instead of type-setting individual letters) that can be reused without changing

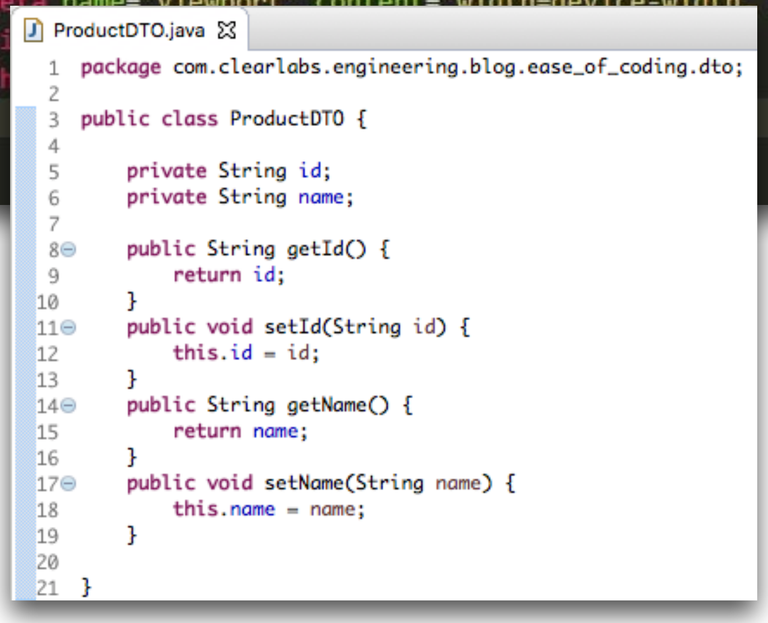


Typeset Text

Boilerplate Text

Boilerplate






```python
if __name__ == '__main__':
    # Anything placed here will never be executed in a module context.
    pass

if __name__ != '__main__':
    # Anything placed here will only be executed in a module context.
    pass
```

Testing (or, more broadly, V&V)

a.k.a. “Is this okay?”

- This is where we check whether the end product is “okay”: we often talk about Software Validation and Verification (V&V)
  - Validation (“are we building the right product?”): checking whether the end product actually satisfies the customer’s requirements
  - Verification (“are we building the product right?”): checking whether, during the process of building the product, we have made any mistakes
- In contrast, the term “Software Quality Assurance” is often used to refer to checking compliance with standards via reviews and audits

Technically “testing” is a type of V&V

Rationalists vs. Empiricists

“It is correct because I **proved** that certain errors do not exist in the system”
(Formal Verification)

“It is correct because I **tried** it several times and it ran okay”
(Software Testing)

V&V is one of the most prolific research areas.

Breakdown of ICSE 2022 Submissions

Top 10 Topics – Submitted

Topics	# Submitted Papers	# Accepted Papers	Acceptance Rate
Machine Learning with and for SE	237	74	31,22%
Software Testing	181	47	25,97%
Program Analysis	117	35	29,91%
Evolution and maintenance	105	31	29,52%
Mining Software Repositories	105	23	21,90%
Software Security	85	25	29,41%
Human Aspects of SE	68	20	29,41%
Validation and Verification	53	15	28,30%
Tools and Environments	49	12	24,49%
Reliability and Safety	46	15	32,61%

Maintenance

How do we keep the software alive?

- Any activities that take place after the delivery of the software project, with the aim of correcting faults or improving other aspects.
  - Perfective, Corrective, Adaptive, and Preventive changes
  - Refactoring: pre-defined systematic code changes that are semantics-preserving but also perfective/preventive
  - Testing & Debugging still takes place
  - Handing incoming bug reports: triage/assignment, issue tracking

Code … smell…?

- Anti-patterns that are symptoms of evolving problems (although they do not break the functionality right away)
  - Long methods/classes/ parameter lists
  - Switch statements in OO
  - Middle-man classes

Bug Triage

- In medicine, when full care cannot be provided for all patients, the available resources should be rationed based on who are the most in need of care.
- A similar thing happens when a bug is reported:
  - How severe is it? Cosmetic, or deal-breaking?
  - Who will be assigned with the task of patching it?
- **Not all bugs are fixed.**
  - Not enough time, too dangerous to attempt to fix, not worth it…

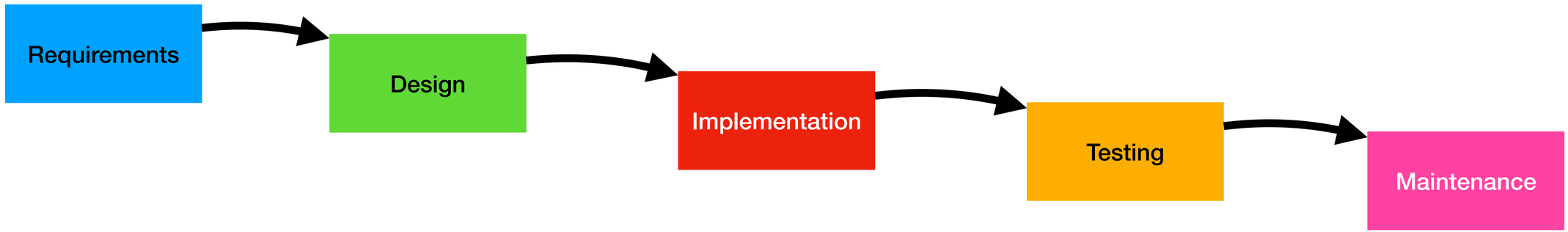
Process Models

- Now that we have the major steps (requirements, design, implementation, v&v, and maintenance), let’s connect them together to get the full picture.
- Dependencies between different stages are rather obvious - why do we have multiple process models?
  - As we gradually understood the nature of SW better, we have collectively updated how we work with SW.
  - They also reflect the changing business needs.

Waterfall Model

Starting from the very early days of computing (1950s)

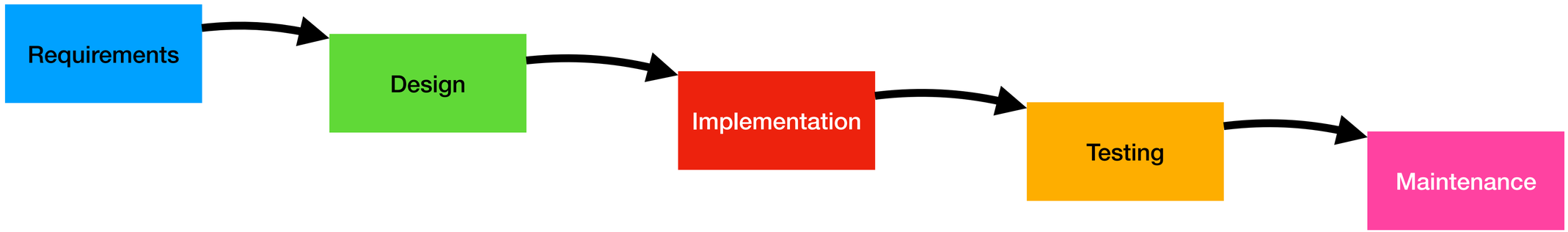
- A linear, one-directional model where one stage depends on the deliverable from the previous stage
- When does it work well?
  - Stable, well-understood requirements
  - Not time-pressured / well managed to flow in one-direction



Waterfall Model

Starting from the very early days of computing (1950s)

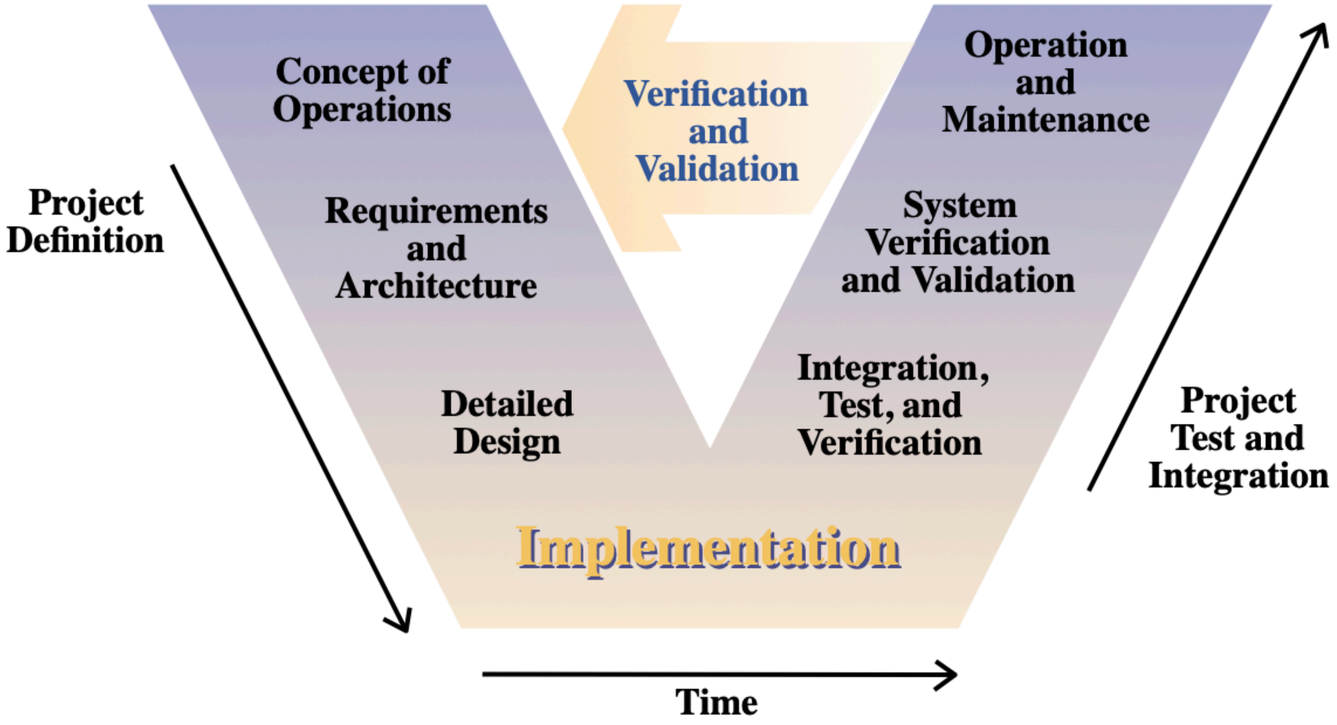
- What are the weaknesses?
  - Cannot easily accommodate changes.
  - There can be blocking stage or activities (within stage), delaying the entire process.
  - We do not have a working version until the very end.



The “V” Model

A variant of the waterfall

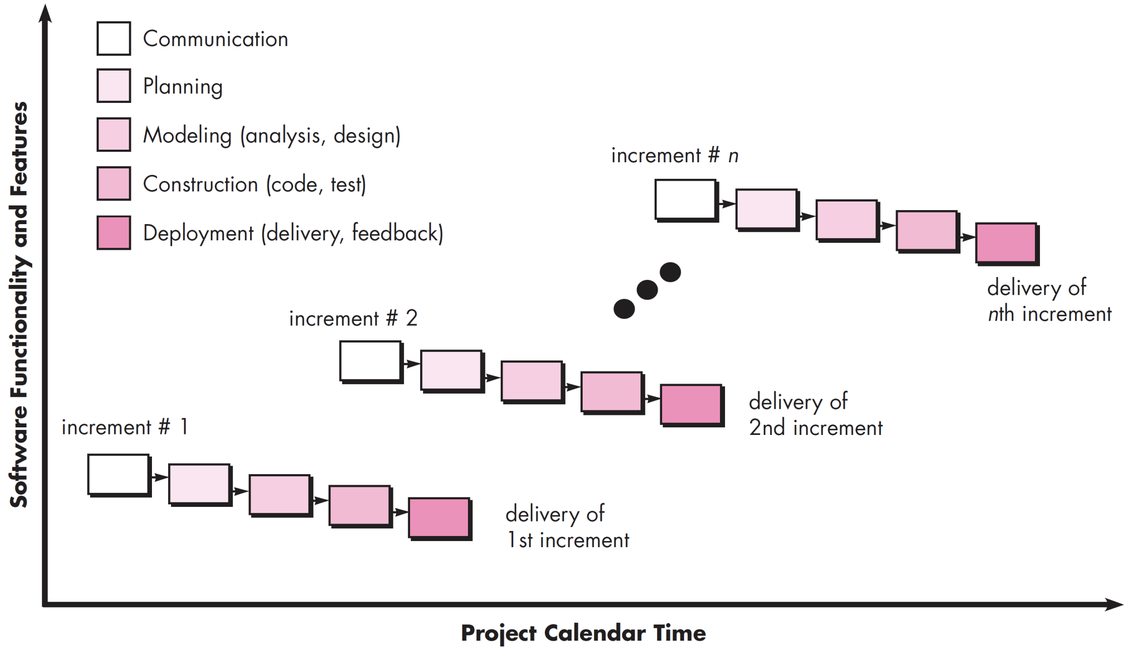
- Extend stages before and after implementation to include more granularity levels
- Couple design stages with V&V stages
- Still inherently linear (it is just folded up), but we are definitely heading into a new direction :)



Incremental Model

Tries to overcome the rigidness of the waterfall model

- Divide and Conquer! Break down the whole project into small functional goals.
- You will have a partially working product at the end of each iteration.



The Spiral Model

Barry Boehm, 1986

- Refinement of the waterfall model: iterative, but not incremental
- Four main phases in each cycle: determining objectives, risk analysis, engineering, planning next iteration
- As the spiral repeats itself, the lifecycle progresses: requirements —> prototype —> core system —> additional features…

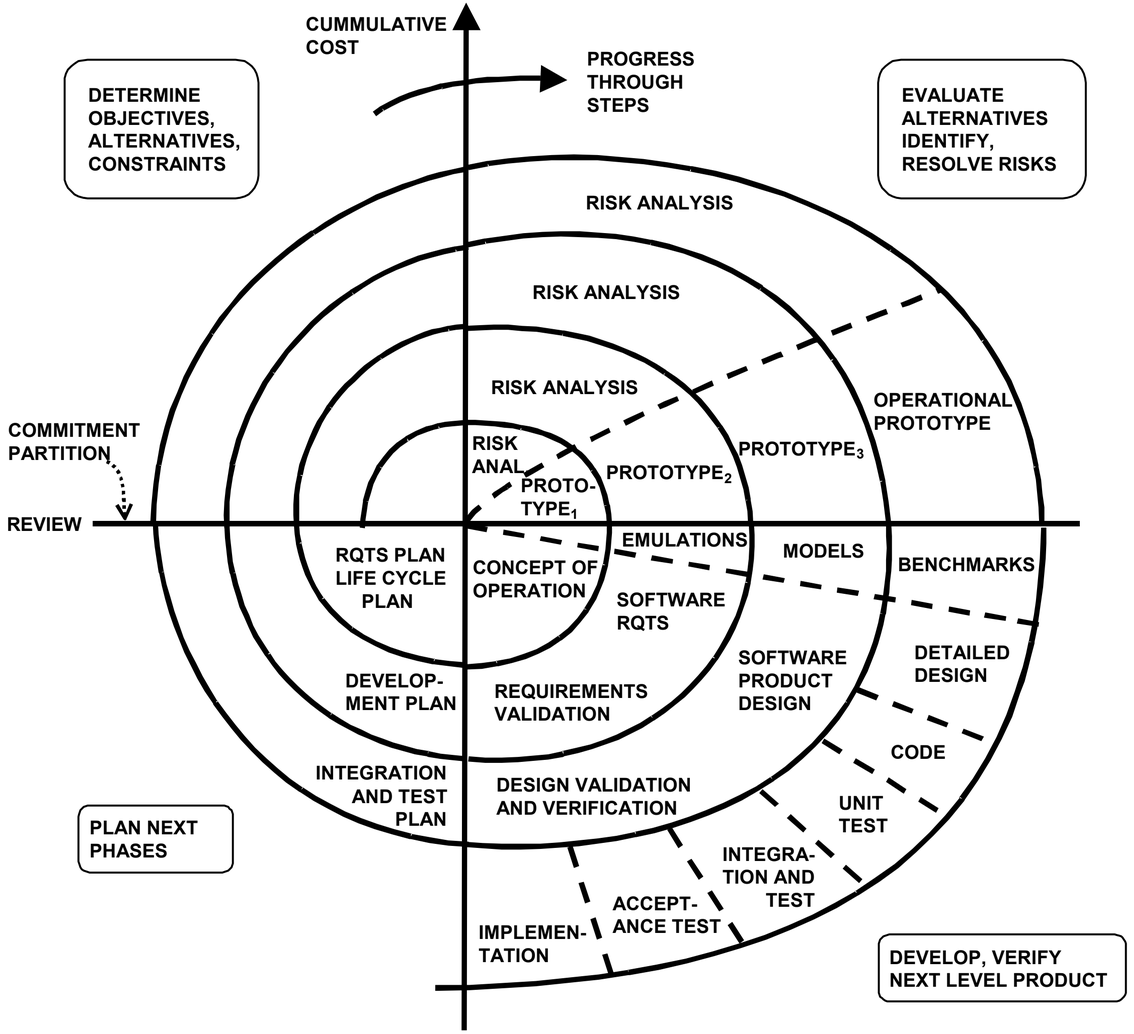
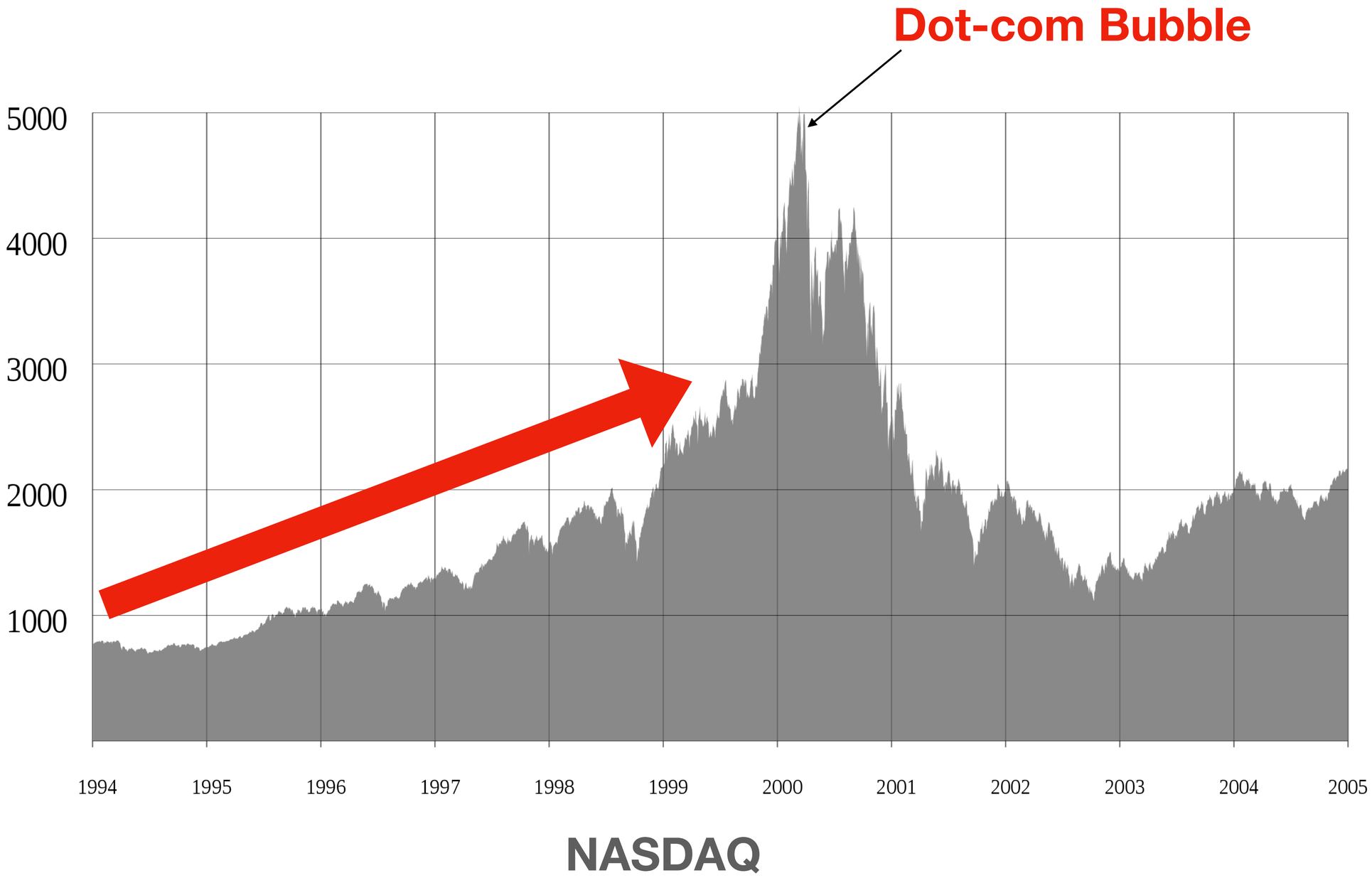


Figure 1: Original Diagram of Spiral Development

The Spiral Model

Barry Boehm, 1986

- Communicate closely with the stakeholders, so that risks can be identified and minimized.
- Create prototype not as a partially working version of the final product, but as a way to extract more complete requirements.
- Assign anchor point milestones so that cost and schedule can be adjusted.
- Weaknesses: heavy and administratively expensive (especially for smaller projects, or if the specifications are relatively clear and would not change)


Dot-com Bubble
5000
4000
3000
2000
1000
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
NASDAQ

Lightweight Process Models

Widely popular during 90s

- In the heated dot-com boom of 90s, software systems became more democratized.
  - Results: more volatile requirements, time-to-market becoming critical
- Reflecting this, many more lightweight process models have been proposed:
  - RAD (Rapid Application Development): less planning, reliant on prototyping
  - XP (Extreme Programming): pair programming, unit test, TDD (Test Driven Development)
  - Scrum: iteratively develop intermediate goals using sprints (a typically 2-week cycle) using small teams, less emphasis on formal processes

Agile Manifesto

http://agilemanifesto.org/ (2001)

- “We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
  - **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.”

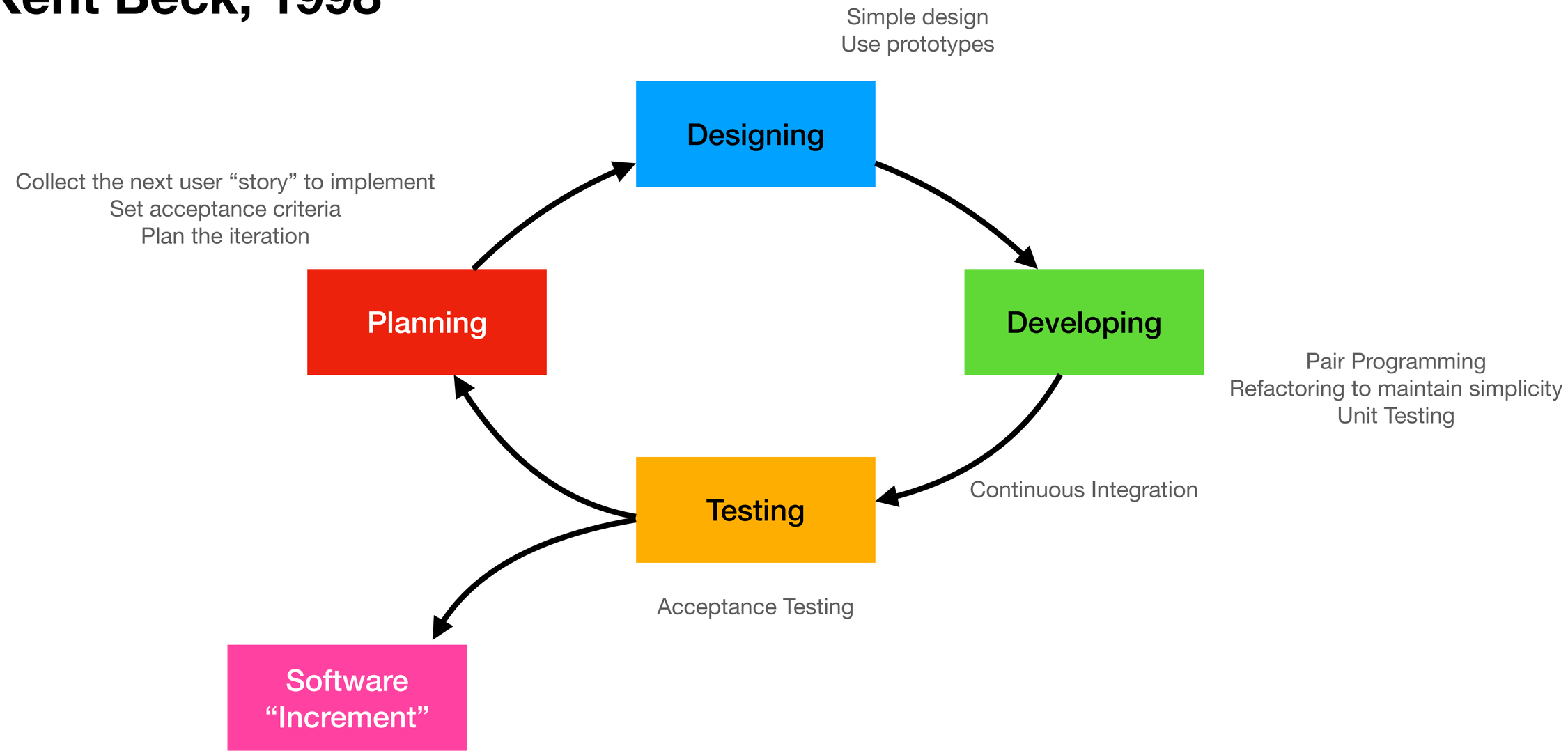
Extreme Programming (XP)

Kent Beck, 1998

- Heavy emphasis on responding to changing requirements; pushes existing ideas to their limits (hence the name).
  - Good communication with the client —> A client representative is part of the developer team and should stay at the site.
  - Develop tests early and concurrently —> Write tests first (TDD).
  - Document the system —> Pair Programming (better productivity)
  - Incremental Development —> Continuous Integration + Small Releases
  - Modular/structural teams —> pairs work on all parts of the system so that everyone knows everything

Extreme Programming (XP)

Kent Beck, 1998



XP Practices

- Incremental Planning: requirements are recorded as user stories, which are assigned to different releases based on available resources and priority. A story is broken down to actual tasks when chosen for a release.
- Small Releases: Release the minimal set of features that can add business values first, and continue to make incremental release frequently.
- Simple Design: just enough to meet the current requirements.
- Test Driven Development: use automated test framework to write tests before writing the actual functionality - tests are actually the specifications.
- Refactoring: continuously look for code that can be improved and refactor for simplicity.

XP Practices

- Pair Programming: program in pairs, checking each other's work (driver writes the code, while the navigator reviews and considers strategic directions.
- Collective Ownership: pairs of programmers work on all areas of the project so that anyone can work on anything.
- Continuous Integration: all finished tasks should be immediately integrated into the whole system, with all tests passing.
- On-site Customer: a client representative should stay with the team for the use of the XP team - client is a member of the team.
- Sustainable Pace: frequent over-times are not desirable as it eventually hurts code quality as well as team productivity.

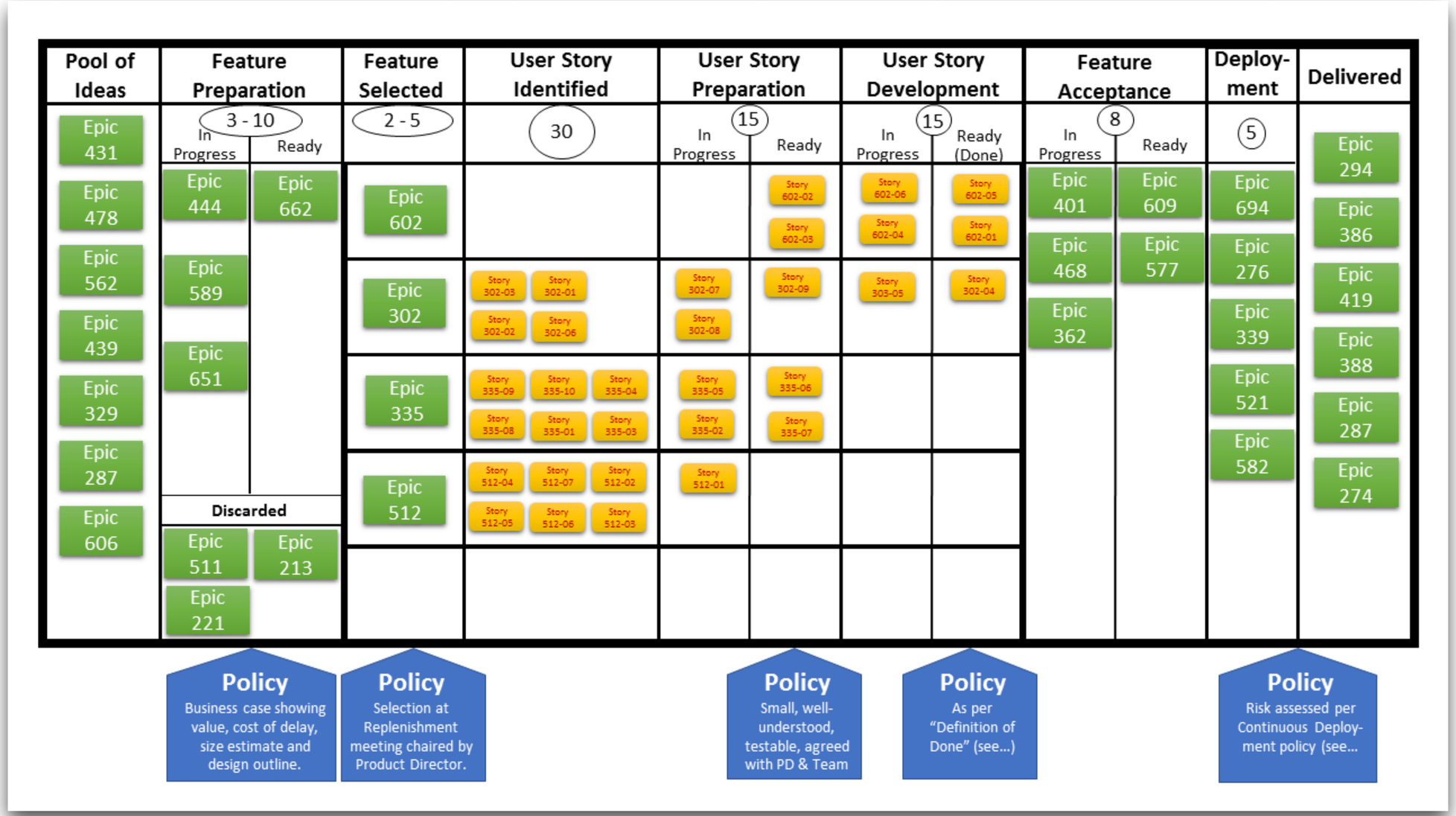
Kanban

Circa 2010

- Visualization of the entire project using Kanban (간판/看板 in Japanese) board.
- Limit Work-In-Progress items; work items are pulled (i.e., teams pick up an item if they have the capacity to do so) rather than pushed (i.e., items are assigned to teams as they are generated).

Kanban

An example Kanban board (https://en.wikipedia.org/wiki/Kanban_(development))



So what do people actually use?

- “It depends” (but note that agile was a huge impact)
- Most likely, some mixture of everything, customized for the organizational needs.
- The point is not to be a purist about any process model; rather, understand the motivation behind all process models invented so far.
  - Planning ahead thoroughly, with well documented decision trails (Waterfall, V, Spiral)
  - Being responsive and resilient to changes (Agile)