Software Engineering 2025–26

Lecture 5: Software Evolution

Or: How do we keep everything working?

Prof. Robert Harle

Department of Computer Science and Technology
University of Cambridge

Easter Term

Introduction to Software Evolution

Software is never "finished."

- Most of a software's life is spent in the **maintenance and evolution** phase.
- Systems must evolve or they become progressively less useful (Lehman's Law).
- Success in software engineering is measured by how well a system survives its own longevity.

Lehman's Laws of Software Evolution

Key observations on E-type (Evolutionary) systems:

- **Continuing Change:** A system must be continually adapted or it becomes progressively less satisfactory.
- **Increasing Complexity:** As a system evolves, its complexity increases unless work is done to maintain or reduce it.
- **Self-Regulation:** Global system evolution processes are self-regulating with statistically determinable trends and invariants.

The Four Types of Software Maintenance

- **Corrective (20%):** Fixing bugs discovered by users.
- **Adaptive (25%):** Updating the system to work in a new environment (e.g., a new OS, cloud migration).
- **Perfective (50%):** Improving the system's performance, maintainability, or readability without changing its behavior.
- **Preventive (5%):** Finding and fixing latent problems before they become actual bugs.

Notice: Most maintenance is not about fixing bugs; it is about keeping the system relevant.

What is Legacy Code?

Defining Legacy Code

Often colloquially defined as "code inherited from someone else," but more practically: **code without tests** or **code we are afraid to modify**.

- Legacy code is the foundation of almost all profitable businesses.
- It is often written in outdated languages or using deprecated frameworks.
- The original authors may have long since left the organization.

Legacy in Action: COBOL & The Pandemic

The Crisis (2020):

- Unemployment systems in several US states (NJ, KS) crashed under COVID-19 load.
- The systems were written in **COBOL** in the 1970s.

The Problem:

- The original authors were retired or deceased.
- Governors had to go on TV to recruit volunteers who knew the language.

"Our systems are 40 years old... we need COBOL programmers."

Lesson: Legacy code is often the most critical code in a business.

The High Cost of Reading

- **Cognitive Load:** Understanding the side effects of a 500-line function takes significant mental effort.
- **Mental Models:** Engineers must build a mental map of how data flows through a system they didn't design.
- **Inconsistency:** Legacy systems often have multiple architectural styles mixed together ("architectural drift").
- **The Fix:** Invest in readability today to save thousands of hours of reading time tomorrow.

Software Archaeology: Finding the "Why"

The source code is the only source of truth.

- **Git Blame:** Shows who changed each line and *when*.
- **Git Log -S:** Search the history for when a specific string (e.g., a function name) was added or removed.
- **The Goal:** Don't just look at *what* the code does; find the *intent* behind it.
- **Linked History:** Professional teams link commits to Issue Trackers (e.g., Jira). A single commit ID (`FIX-123`) can explain the business context that code comments miss.

Software Archaeology

Tools for understanding the past:

- **Version Control (Git):** Using `git blame` and `git log` to find the *intent* behind a change.
- **Issue Trackers:** Linking commits back to Jira or GitHub issues for context.
- **Implicit Knowledge:** The danger of "tribal knowledge" that isn't captured in the system.

The API Contract

Application Programming Interfaces (APIs) are promises.

- An API defines the interaction boundary between different components or systems.
- Once an API is published and used by others, changing it becomes extremely expensive.
- **The Contract:** You promise that if the caller provides Input X, you will provide Output Y.

Semantic Versioning (SemVer)

Format: MAJOR.MINOR.PATCH

- **MAJOR:** Breaking changes (incompatible API changes).
- **MINOR:** New features (backwards-compatible functionality).
- **PATCH:** Bug fixes (backwards-compatible bug fixes).

Example: 2.1.4 → 2.2.0 (New feature) → 3.0.0 (Breaking change)

What constitutes a "Breaking Change"?

If the caller's code no longer compiles or runs, it's broken.

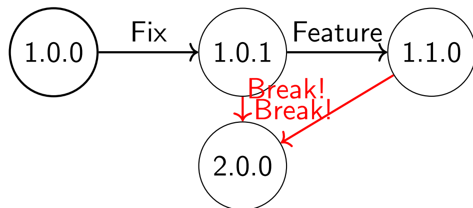
- **Structural Changes:**
  - Removing or renaming a function, class, or field.
  - Changing the type of a parameter (e.g., `int` $\rightarrow$ `string`).
  - Adding a **required** parameter to an existing function.
- **Behavioral Changes:**
  - Changing the default value of a parameter.
  - Throwing a new checked exception.
  - Changing the order of results returned by a query.
- **The Rule:** When in doubt, it's a MAJOR bump.

The "Zero-Ver" Problem (0.y.z)

Initial development is chaotic.

- SemVer says: **0.y.z** is for initial development. Anything may change at any time.
- **The Trap:** Many projects stay on `0.x` for years (e.g., early React, Terraform).
- **The Risk:** Users cannot rely on MAJOR/MINOR/PATCH logic to protect them from breakage.
- **Best Practice:** Bump to **1.0.0** as soon as the software is used in production. It is a signal of maturity and a commitment to stability.

Visualizing SemVer Transitions



- **Red arrows** indicate transitions that require callers to update their code.

Dependency Management: Constraints

How do package managers (npm, pip, cargo) decide which version to install?

- **Exact:** `1.2.3` → Only 1.2.3. (Safest, but no bug fixes).
- **Tilde (~):** `~1.2.3` → Allows PATCH updates (up to 1.2.x).
- **Caret (^):** `^1.2.3` → Allows MINOR and PATCH updates (up to 1.x.x).
  - This is the **industry default** for most modern systems.
  - It assumes authors follow SemVer strictly.
- **Lockfiles (`package-lock.json`, `Cargo.lock`):** Record the *exact* version used in the last successful build to ensure reproducibility across the team.

The Robustness Principle (Postel's Law)

"Be conservative in what you send, be liberal in what you accept." — *Jon Postel*

- **Conservative (Writer):** Stick strictly to the API contract. Don't send extra fields that might confuse others.
- **Liberal (Reader):**
  - If you receive extra fields you don't recognize, **ignore them** instead of crashing.
  - This enables **Forward Compatibility**. Newer systems can add fields without breaking older ones.
- **Evolution:** This simple rule is why the Internet (TCP/IP, HTTP, HTML) has survived for 40+ years.

Backward vs. Forward Compatibility

- **Backward Compatibility:** Newer code can read data or handle requests generated by older code. (Crucial for deployments).
- **Forward Compatibility:** Older code can gracefully handle data generated by newer code (e.g., by ignoring unknown fields).
- **Why it matters:** In a distributed system, you cannot upgrade every node at the exact same microsecond. You will always have a mix of versions running simultaneously.

The Cost of Breaking Changes

Why we avoid Major version bumps:

- **Downstream Friction:** Every user of your library must now allocate engineering time to fix their broken integration.
- **Ecosystem Split:** Some users stay on V1 while others move to V2, doubling your maintenance burden.
- **Hyrum's Law:** "With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody."

Hyrum's Law: The SimCity Hack

The Cost of Extreme Backwards Compatibility

- **The Situation:** Windows 95 developers found that **SimCity** crashed on the new OS due to a memory bug in the game.
- **The Decision:** Microsoft didn't want SimCity to be broken for users.
- **The Solution:** They added a specific check in the Windows 95 kernel: *"If the app is SimCity, use a special, slower memory allocator that masks the game's bug."*
- **The Takeaway:** Once you have enough users, every bug in your system becomes a "feature" that someone depends on.

Deprecation Cycles

How to break things gracefully:

1. **Announce:** Warn users that a feature will be removed.
2. **Deprecate:** Mark the API as `@Deprecated` (generates compiler warnings).
3. **Wait:** Give users 6-12 months (or several minor versions) to migrate.
4. **Remove:** Finally remove the code in a new MAJOR version.

Safe Refactoring

Restructuring without changing behavior.

- Refactoring is not "cleaning up"; it is a disciplined engineering technique.
- **Prerequisite:** A comprehensive suite of automated tests.
- If the tests pass before and after the change, you have (likely) preserved behavior.

The Technical Debt Quadrant

	Reckless	**Prudent**
Deliberate	"No time for design"	"We must ship now and fix later"
Inadvertent	"What's pytest?"	"Now we know how we should have done it"

- **Prudent/Deliberate** is often a valid business decision.
- **Inadvertent** debt is where the most dangerous complexity grows.

Debt in Action: The $300 Billion Y2K Bug

- **The Debt (1960s):** To save 2 bytes of expensive RAM, years were stored as two digits (98 vs 1998).
- **The Assumption:** "This code will be replaced by the year 2000."
- **The Reality:** Lehman's Law (Continuing Change) meant the systems stayed in use for 40+ years.
- **The Interest:** In 1999, the world spent an estimated **$300 Billion** to prevent global financial collapse.

Lesson: Short-term technical debt can have multi-billion dollar long-term interest.

Characterization Tests

How to refactor code that has no tests?

- You cannot write unit tests because you don't understand the requirements yet.
- Instead, write **Characterization Tests** (or Golden Master tests).
- Record the current output for a given input, and assert that the output remains identical during refactoring.
- You are testing for *consistency*, not *correctness*.

The Strangler Fig Pattern

How to replace a legacy monolith?

- **Origin:** Named after the *strangler fig* vine that grows around a tree, eventually replacing it entirely.
- **Strategy:**
  1. Create a **Proxy/Facade** in front of the legacy system.
  2. Build the new functionality as a separate service.
  3. Route specific calls to the new service; leave the rest for the legacy.
  4. Gradually migrate all functionality until the legacy system has no traffic.
- **Benefit:** Low risk; continuous delivery; avoids the dangerous "Big Bang" rewrite.

The Big Bang Rewrite: The Netscape Disaster

Why the Strangler Fig is usually better...

- **1997:** Netscape Navigator dominated the browser market.
- **The Decision:** Their code was "messy," so they decided to throw it all away and rewrite from scratch (Netscape 5.0).
- **The Result:** The rewrite took **3 years**. In that time, they shipped no new features.
- **The Death:** Microsoft's Internet Explorer took 90% of the market. Netscape went bankrupt.
- **Rule:** Never do a Big Bang rewrite of a successful system.

When to Refactor? (Code Smells)

Identifying indicators of deeper problems.

- **Long Method:** If you can't describe what a function does in one sentence, it's too long.
- **God Object:** A class that knows too much or does too much. (Violation of Single Responsibility).
- **Feature Envy:** A method that seems more interested in a class other than the one it actually is in.
- **Data Clumps:** Groups of variables that are always passed together (e.g., `x`, `y`, `z` coordinates).

A smell is not a bug; it is a signal that the design is decaying.

Refactoring Example: Extract Method

```
void printOwing() {
    printBanner();
    // Calculate outstanding
    double outstanding = 0.0;
    for (Order o : orders) {
        outstanding += o.getAmount();
    }
    // Print details
    System.out.println("name: " + name);
    System.out.println("amount: " + outstanding);
}
```

Listing 1: Before Refactoring

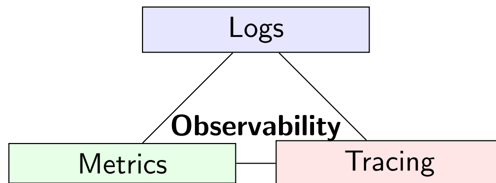
Refactoring Example: After Extract Method

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}

double getOutstanding() {
    double result = 0.0;
    for (Order o : orders) { result += o.getAmount(); }
    return result;
}

void printDetails(double outstanding) {
    System.out.println("name:␣" + name);
    System.out.println("amount:␣" + outstanding);
}
```

Monitoring vs. Observability

- **Monitoring:** Tells you *that* something is wrong. ("The CPU is at 99%"). It is about known unknowns.
- **Observability:** Tells you *why* something is wrong. It is the ability to understand the internal state of a system by looking at its external outputs. ("Which specific request caused the memory leak?").

The Telemetry Triad



- **Logs:** Discrete events (textual).
- **Metrics:** Aggregatable numbers (counters, gauges).
- **Tracing:** End-to-end request flow through microservices.

The Four Golden Signals of Monitoring

What should you actually measure?

- **Latency:** Time it takes to service a request (ms).
- **Traffic:** Demand placed on the system (e.g., HTTP requests/sec).
- **Errors:** The rate of requests that fail (e.g., 500-errors, timeouts).
- **Saturation:** How “full” is your service? (e.g., CPU, Memory, Disk I/O).

If you only measure four things, measure these.

Site Reliability Engineering (SRE)

> *"SRE is what happens when you ask a software engineer to design an operations team." — Ben Treynor Sloss (Google)*

- **The Core Philosophy:** Treat Operations as a *Software Problem*. If a task is repetitive, don't just do it—write a script to automate it.
- **The 50% Rule:** Google SREs spend max 50% of their time on "toil" (manual ops). The other 50% **must** be spent on engineering projects to improve the system.
- **Reliability is a Feature:** 100% reliability is the wrong target for almost everything. It is too expensive and slows down innovation.
- **Key Vocabulary:**
  - **SLI (Service Level Indicator):** What you measure (e.g., Latency).
  - **SLO (Service Level Objective):** The target (e.g., $<$ 200ms for 99.9% of requests).

The Error Budget

Balancing Stability vs. Velocity

- **Error Budget** = 100% - SLO.
- If your SLO is 99.9% uptime, you have a 0.1% error budget.
- If you have budget left, you can push new (risky) features.
- If the budget is exhausted, all feature development stops to focus on stability.

Blameless Post-mortems

- When a system fails in production, the goal is not to find someone to fire.
- **Blamelessness:** Assume everyone acted with the best intentions given the information they had.
- Focus on the **systemic reasons** for failure (e.g., why did the CI pipeline allow a broken build to deploy?).
- Use the post-mortem to create new automated safeguards.

Resilience Engineering: Embracing Failure

If failure is inevitable, make it frequent.

- Traditional Ops: Try to prevent failure at all costs (high MTBF - Mean Time Between Failures).
- Modern SRE: Assume things *will* fail. Focus on **MTTR (Mean Time To Recovery)**.
- **The Goal:** A system that can survive the loss of a disk, a server, or an entire data center without user impact.

The Story of Chaos Monkey (Netflix)

The Context (2010):

- Netflix was migrating from stable data centers to the "unreliable" cloud (AWS).
- They knew AWS instances would disappear randomly.

The Solution:

- They built a tool that **randomly killed production servers** during business hours.
- **Why?** It forced engineers to build services that recovered automatically.
- If you know your server might die at 2 PM on a Tuesday, you make sure it doesn't matter.

Monkey

Kill!

Production Environment

Chaos Engineering

"Chaos Engineering is the discipline of experimenting on a software system in order to build confidence in its capability to withstand turbulent conditions in production."

- **Steady State:** Define what "normal" looks like (e.g., error rate $< 0.1\%$).
- **Hypothesis:** "If we stop the database, the cache should keep the site alive."
- **Experiment:** Introduce a failure (stop the DB).
- **Verify:** Did we maintain the steady state?
- **The Simian Army:** Expanded to **Chaos Gorilla** (kills availability zones) and **Chaos Kong** (kills entire regions).

Managing Dependencies

The "Dependency Hell"

- Modern apps rely on hundreds of third-party libraries (npm, PyPI, Maven).
- **Transitive Dependencies:** Your dependency has its own dependencies.
- **Security Risk:** A vulnerability in a small, obscure library can compromise your entire system (e.g., Log4Shell).

Supply Chain Fragility: The "Left-Pad" Incident

The 11 lines of code that broke the Internet.

- **2016:** A developer deleted a tiny package called `left-pad` (11 lines of code) from the `npm` registry.
- **The Impact:** Within hours, thousands of projects (including React and Babel) failed to build.
- **The Discovery:** Developers didn't even know they depended on it; it was a **Transitive Dependency**.
- **Lesson:** You are responsible for the security and availability of every library in your tree.

Vulnerability Patching

Stay updated or get hacked.

- **CVE (Common Vulnerabilities and Exposures):** A list of publicly disclosed cybersecurity vulnerabilities.
- Engineers must regularly update dependencies even if no new features are needed.
- Automated tools (Dependabot, Snyk) can help by flagging outdated libraries.

The Software Supply Chain

Trusting the artifacts.

- How do you know the code in the `.jar` file matches the source code on GitHub?
- **SBOM (Software Bill of Materials):** A formal record containing the details and supply chain relationships of various components used in building software.
- Critical for high-security environments (defense, banking, healthcare).

Supply Chain Disasters: Log4Shell

- **The Vulnerability (2021):** A bug in `log4j`, a standard Java logging library.
- **The Attack:** Hackers could take over a server just by sending a crafted string to be logged (e.g., via a chat box or search bar).
- **The Crisis:** Companies spent weeks manually searching their systems to see if they even used `log4j`.
- **The Solution: SBOM (Software Bill of Materials)**. A machine-readable list of all components. If a bug is found in a library, you can instantly see if you are affected.

Lecture 5 Summary: Software Evolution

- **Software Never Stops Changing:** Lehman's Laws dictate that systems must evolve or become useless.
- **The 80/20 Rule:** Most of the cost is in maintenance; write code for the next engineer.
- **Legacy Code:** Is unavoidable foundation of business. Use archaeology and tests to modify it safely.
- **API as a Contract:** Long-term stability requires SemVer and careful deprecation paths.
- **Sustainability:** Manage your supply chain and technical debt to keep systems healthy.