Architecture, Complexity, and Refactoring

JD Kilgallin

CPSC:480

10/24/22

Photo: Lynn Conway, Alchetron
Known for: Out-of-order processor execution, Very-Large Scale Integrated Circuits, director of DARPA Strategic Computing Initiative, subject of IEEE Journal of Solid-State Circuits special report, LGBT inclusion in IEEE Code of Ethics

Architecture, Complexity and Refactoring

JD Kilgallin

CPSC:480

10/24/22

Pressman Ch 14, 23.3, 27.4-5

Notes

- Exercise 5 grades are posted. Most students did well, but I should have been clearer on expectations for number of comments per review. Therefore, the deadline will be **extended to tomorrow** night (**Tuesday, Oct 25, 11:59 PM**) and you may (re)-submit your work (re-open the same pull request if it's closed). Full credit should include at least **25 comments per implementation**, and the overall reviews should state your impression of **each** codebase along with the best/worst. Be sure to make a PR from a **branch** in the repo and that you **submit** the review after entering comments.
- Quiz Wednesday on lectures since midterm (including this one). Expect 2-3 more quizzes before the final, plus 2 team surveys and evaluations of final project presentations, for a total of ~15.
- Only 7 more full, examinable lectures after today!

Learning objectives

- Software patterns and frameworks
- Measuring complexity
- Managing complexity
- Types of software refactoring
- When to refactor

Software Architecture

- Recall prior definitions:
  - The relationship between major structural elements of the software, along with the style and patterns that can be used to achieve the requirements...
  - The overall structure of the software...
  - A process whereby software is decomposed into distinct but interrelated modular components...to limit complexity and facilitate implementation...
- Quality architecture maintains separation of concerns and simple implementations of cross-cutting concerns.
- Architecture quality is a major factor in maintainability and extensibility of a system by making it easier to change.
- A product can have a hierarchy of architectures at multiple levels.

Architecture Pattern

- Most problems have many solutions. Some are easier, some are better for certain situations, etc. But most solutions don't need to reinvent the wheel.
  - For example, Reddit, Keyfactor, and retail inventory systems all implement Create-Read-Update-Delete (CRUD) operations for different resources (i.e. posts, certificates, goods) and have similar architectures as a result.
- Recall: a pattern is a recognizable, repeatable template for a solution to one class of problems that has been proven to be effective.
  - We discussed the Model-View-Controller (MVC) pattern and briefly mentioned a few others.
- May dictate modules' internal construction or only describe interface.
- A product may use multiple architecture patterns.
- There are also patterns for classes, interfaces, and other entities.

Layered architectures

- Most conventional, standalone desktop applications & web applications
- Aka 3-tier architecture (*n*-tier more generally, but n is usually 3).
- MVC is one type of layered architecture.
- Data input goes from outer layer to inner, output vice versa:
  - Presentation layer – User I/O (GUI or CLI, possibly another application)
  - Application layer – Logic, program control flow and data routing
  - Data access layer – Database (persistent storage).
- Changes require re-deployment of the whole application or a large portion, usually with downtime.
- Prone to some tangling and scattering, and challenges with testing.

Event-driven architectures

- Aka (or at least similar to) publisher-subscriber or producer-consumer.
- Suitable for some large cloud applications, parallel-processing computations, and distributed systems.
- Loosely coupled, but can be difficult to test.
- Messages or events are generated/published/produced, then filtered, routed and stored by other components, and finally acted on by subscribers or consumers of the event or message.

Microservices architectures

- Suitable for cloud applications, & applications using a DevOps process
- Independent modules that minimize responsibilities of each module.
- Can accommodate large, globally distributed development teams.
- Maximizes separation of concerns and addresses the architectural cross-cutting concerns of extensibility, scalability, and testability easily
- Allows dynamic, incremental upgrade of different product components at different rates with no application downtime.
- Not all applications can be broken into microservices easily.
- Best example is the Amazon website: independent services for search, reviews, recommendations, payments, ads, images, etc.

Frameworks

- A framework is an infrastructure component that facilitates a software architecture with some aspects pre-built or automated.
- A concrete implementation of a software architecture pattern.
- Extension points or plug points are interfaces a framework provides to apply the framework to the specific context of the problem to be solved with the application.
- Enforces consistent source code and behavior, but can be limiting (difficult to switch frameworks, so may be stuck with what the framework allows).
- Some examples: Angular, Node.js, React, Django, and Flask are popular frameworks for web development; Unity and Unreal are game development frameworks; Selenium is a software testing framework.
- Some combinations of other technologies constitute a framework; e.g. the LAMP stack consists of Linux, Apache, MySQL, and PHP, which allows web applications to be written using free software.

Complexity

- Applies to both the product and its source code, in different ways.
- A good software architecture limits the complexity that needs to be considered at any point in software development and maintenance.
- Recall Lehman's Law of Increasing Complexity — as an E-type system evolves, its complexity increases unless work is done to maintain or reduce it.
- Use of architecture patterns and frameworks helps ensure that software development and maintenance can continue even as the product increases in complexity.
- "Small" commercial software products contain 10k-1 million lines of code (17k for Keyfactor C agent). Large software systems can contain >100 million, so complexity is a major concern in all products.

Metrics

- Ability to quantify attributes of software, including source code and product behavior visible to end users.
- Can [attempt to] measure quality, complexity, size, usability, etc.
- Ideal metrics are:
  - Easy to measure (measurements that require excessive work have less value)
  - Easy to understand and interpret (esoteric metrics provide less value)
  - Relevant (measuring something that doesn't matter has no value)
  - Intuitive (should align with subjective expert impressions of the attribute)
  - Stable (small changes in the software, especially ones that aren't intended to affect the metric, should not cause large change in measurement of metric)
  - Predictable (should be able to anticipate the effect of a change on a metric)
- Consider units of a metric especially for relevance & intuition – does "number of variables times number of developers" make sense?

Cyclomatic Complexity

- Number of distinct code paths through a program
  - Two paths are distinct ("linearly independent") if you can list the instructions in order, and there's at least one instruction that's in one but not the other.
  - Every conditional ("if" statement, ternary "? :", etc) increases CC by number of Boolean terms ("if (a ^ b) {…}" = "if (a) { if (b) {…}}"; CC += 2).
  - Every loop increases CC by 1; at the end of the loop, you can go on or go back.
- If you drew a state diagram with **every** line of code as its own state, the cyclomatic complexity is E - N + P, where E is the number of transitions (edges), N is the number of states (nodes), and P is the number of entry points or disjoint components.
- Can be computed for any level: an individual function, class, module, application, or even a larger system of multiple applications.

Cyclomatic Complexity

- Functions with a CC above 10-15 are excessively complex, and good candidates for decomposition into smaller functions.
- Cyclomatic complexity for a large system can be on the order of the number of lines of code, so approaching a billion.
- Modules with higher complexity per line of code are, in general, at higher risk for bugs, and harder to maintain. Average CC does depend on language, as well as size and type of project.
- Cyclomatic complexity corresponds to the number of test cases needed to provide complete code coverage; one test per code path allows all code to be tested (code coverage of tests will be discussed next week).

Scope Creep

- Some complexity comes from implementing new functionality between releases, with the product including more requirements and more code each time.
- Some complexity comes from adding bells and whistles to existing functionality that goes beyond the requirements (implementation/ design cannot be traced all the way back to initial requirements).
- *Scope creep* (aka feature creep, kitchen sink syndrome, etc) is the tendency toward unplanned growth of features developed in a product release without underlying requirements changes.
- Frequently comes from developers saying "oh it would be trivial to add this additional piece" or "we might need this in the future", but project management is ultimately responsible for preventing over-engineering that impacts project success.

Managing Complexity

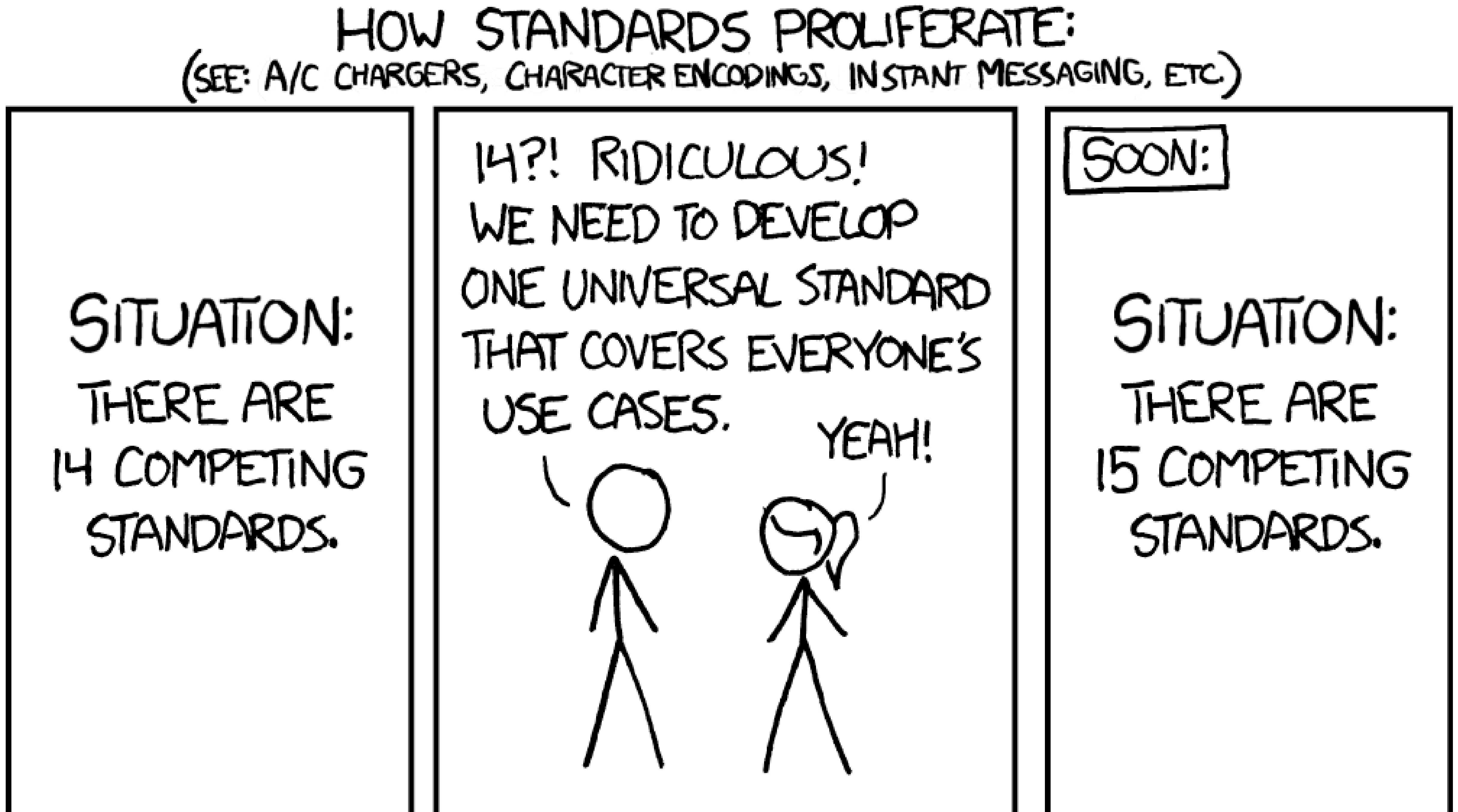
- The easiest way to manage complexity of the product *and* its source code is to invest in getting the architecture right from the beginning.
- The second factor is to ensure that all implementation is traceable to a bona fide requirement, and project management dictates what gets prioritized:
  - Paradox of Choice - More options/features do not make customers happier; in fact, it can cause stress, confusion, and frustration, and give the impression that it's hard to make the software do exactly what they want.
  - "A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away" -Antoine de Saint-Exupéry
  - "The perfect is the enemy of the good" - There is limited value in improving functionality that already accomplishes its goal "well enough"
  - "Premature optimization is the root of all evil" -Donald Knuth - Simple implementations are often preferable to more efficient but more complex alternatives
  - "Sooner or later, you have to shoot the engineer and put the product in to production" - A good product released today is usually preferable to a better product 6 months late

Refactoring

- Changing the structure of a software system or component without changing its functionality.
- Common reasons include eliminating technical debt, improving product performance or reliability, making code more readable/ testable/maintainable, swapping libraries or technologies used for an operation, or because you don't understand someone else's code.
- Inevitable in large projects; may comprise 20-30% of developers' time!
- Frequently done in conjunction with new development; halting new feature development is unappealing to stakeholders. This is a factor in why it takes more work to modify a large product than a small one.
- Can provide a major productivity boost to the rest of the team.

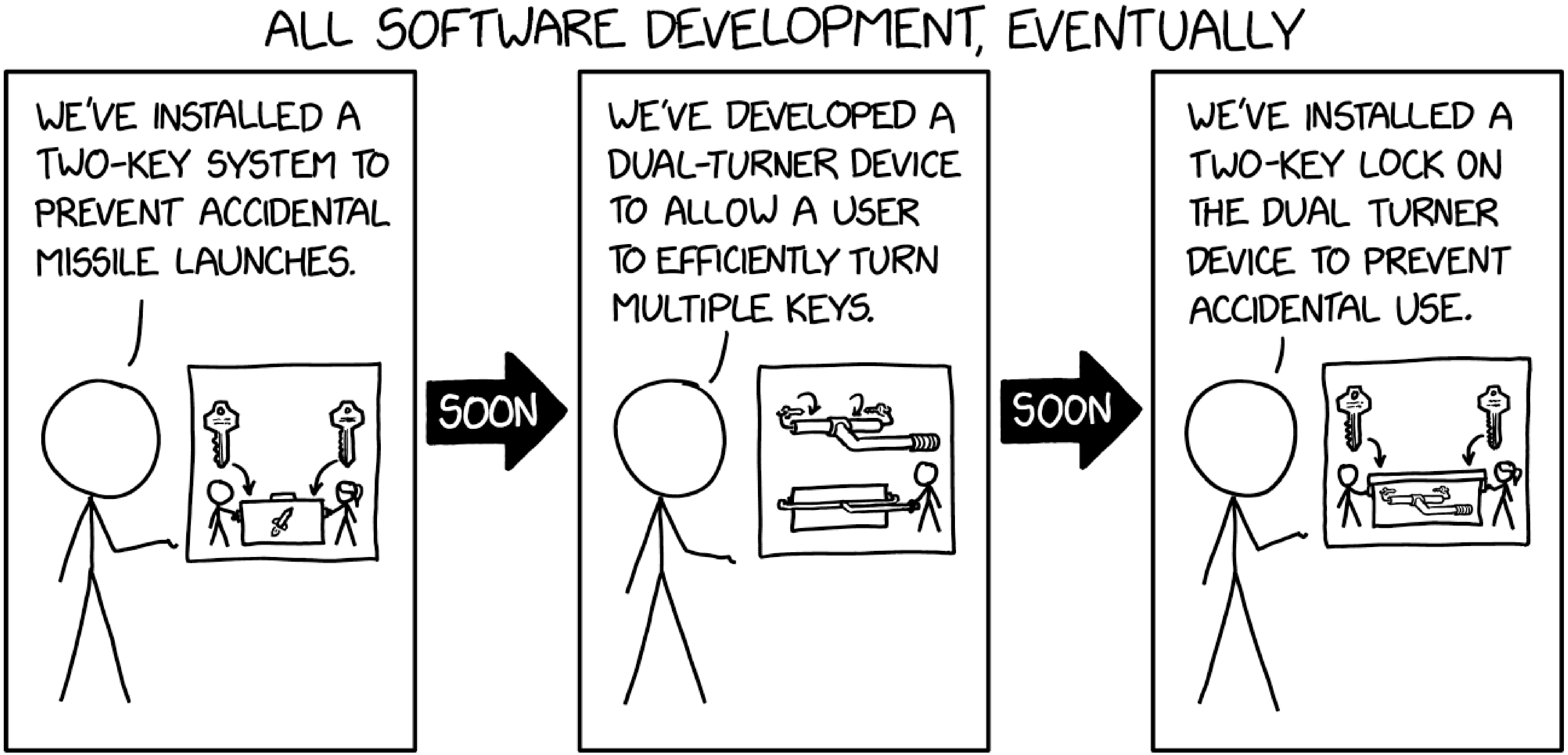
Data refactoring

- Adjustments to the format or schema of data.
- Attempts to standardize formats with minor differences, reduce the number of formats, or optimize operations on the data.
- Can be done within a component, at the interface between components, and in persistent storage.
- May be minor (changing attribute names for consistency) or a complete overhaul - this usually happens when a product has evolved well beyond its initial requirements and scope.
- Some techniques aim to facilitate this or reduce the need to refactor data - e.g. Entity-Attribute-Value (EAV) schemas and some "no-SQL" database technologies allow storing distinct types in the same location of a database without a rigid schema.


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

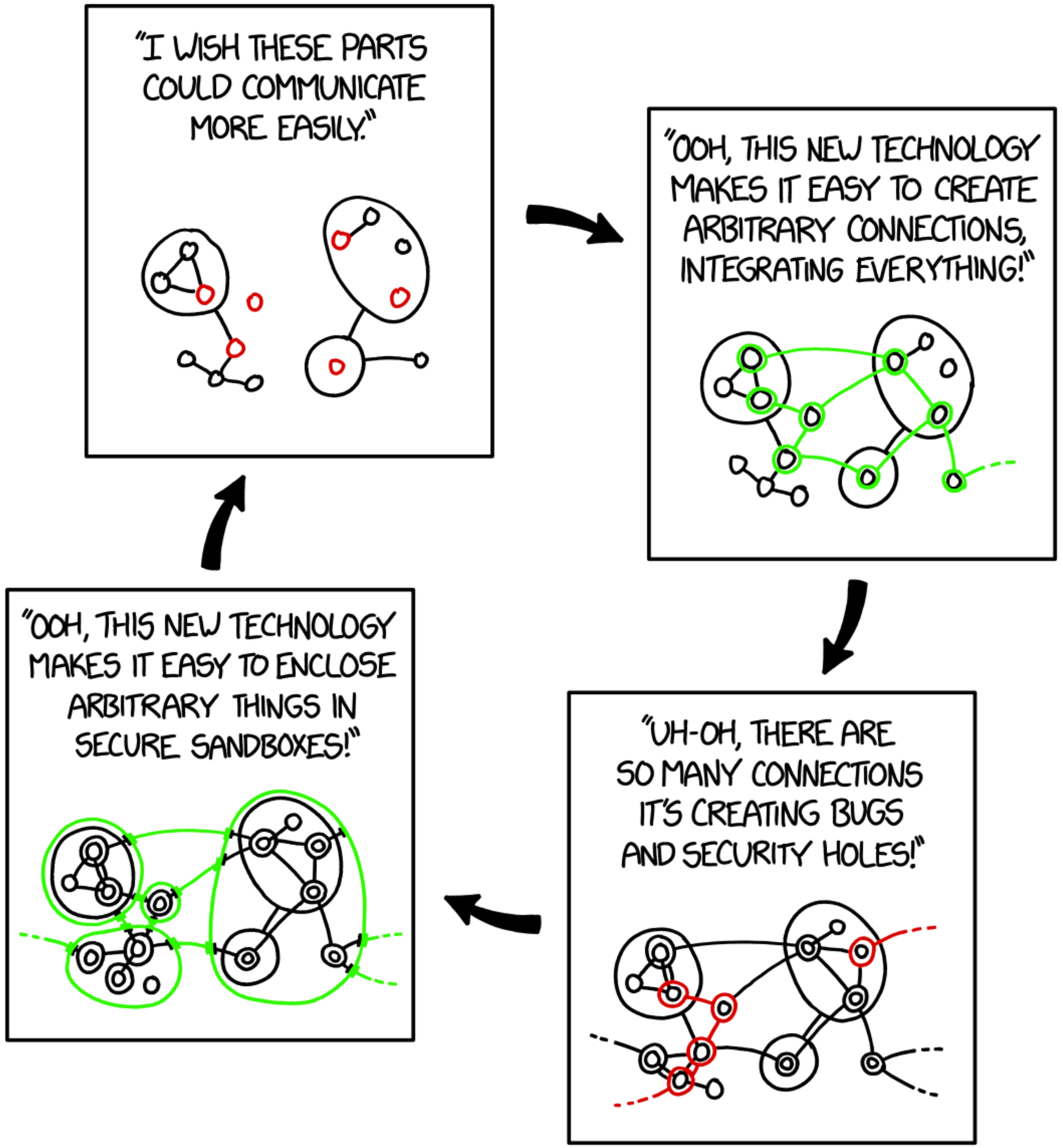
Code refactoring

- The most common type of refactoring.
- Goal is to produce higher-quality code than the previous code that accomplished the same task.
- Frequently involves decomposing tangled concerns and complex functions into smaller pieces, or consolidating scattered but related code into a single function, class, or module.
- Code that's affected most by requirement changes and new requirements is the most likely to need refactoring repeatedly.
- Legacy code may be written in a much different style than the rest of the codebase, by developers no longer on the team, and possibly in a different language; needs to be rewritten to conform to current expectations.


ALL SOFTWARE DEVELOPMENT, EVENTUALLY
WE'VE INSTALLED A TWO-KEY SYSTEM TO PREVENT ACCIDENTAL MISSILE LAUNCHES.
SOON
WE'VE DEVELOPED A DUAL-TURNER DEVICE TO ALLOW A USER TO EFFICIENTLY TURN MULTIPLE KEYS.
SOON
WE'VE INSTALLED A TWO-KEY LOCK ON THE DUAL TURNER DEVICE TO PREVENT ACCIDENTAL USE.

Architecture Refactoring

- Changes the architecture of the system, frequently with the goal of improving scalability, extensibility, and testability. For example, separating a single module into a layered application, or separating a layered application into independent microservices.
- May be required when adapting the product to run in a new environment or when changing frameworks, languages, etc.
- May be required when adding significant new functionality, or when changing handling of a cross-cutting concern (such as access control).
- Disruptive to development, and difficult to accomplish.


"I WISH THESE PARTS COULD COMMUNICATE MORE EASILY."
"OOH, THIS NEW TECHNOLOGY MAKES IT EASY TO CREATE ARBITRARY CONNECTIONS, INTEGRATING EVERYTHING!"
"UH-OH, THERE ARE SO MANY CONNECTIONS IT'S CREATING BUGS AND SECURITY HOLES!"
"OOH, THIS NEW TECHNOLOGY MAKES IT EASY TO ENCLOSE ARBITRARY THINGS IN SECURE SANDBOXES!"

Forward Engineering

- Designing and implementing product architecture or code for requirements that don't yet exist.
- Anticipating the likely future needs of a software product can significantly reduce the amount of refactoring that needs to be done in the future, and is worth considering in requirements analysis and project planning.
- Writing code that is easy to extend and modify is great.
- Writing code that can handle use cases "out of the box" that it doesn't need to handle currently is a problem.

Reengineering

- Sometimes the best way to proceed with a system is to reevaluate the entire set of requirements and architecture decisions.
- Lessons learned from one project that's evolved over many iterations can be applied to make a new product that is easier to use and develop.
- Frequently coincides with major branding and marketing changes.
- New technologies that weren't available when the first project was started may allow the product to do things it couldn't do before.
- Minimizing the amount of new code that needs to be written is good; old code can be reused where it's appropriate, but just because an implementation was already developed doesn't mean it's the best fit for the new product.

References

- Lynn Conway. Sneha Girap. Oct 2017. Alchetron.
- Top 5 Software Architecture Patterns. Archna Oberoi. July 2021. Daffodil Software.
- Code Metrics - Cyclomatic Complexity. Mike Jones and Gordon Hogenson. April 2022. Microsoft Learn.
- What is feature creep and how to manage it. Georgi Todorov. June 2021. Paddle.
- How Much Time Do Developers Spend Actually Writing Code? Chris Grams. Oct 2019. The New Stack.
- Standards. Randall Munroe. July 2011. xkcd.
- Two Key System. Randall Munroe. Sept 2022. xkcd.
- Sandboxing Cycle. Randall Munroe. Sept 2018. xkcd.

- *Reading for next lecture: Pressman ch 15-16*