Handbook of Software Engineering Methods

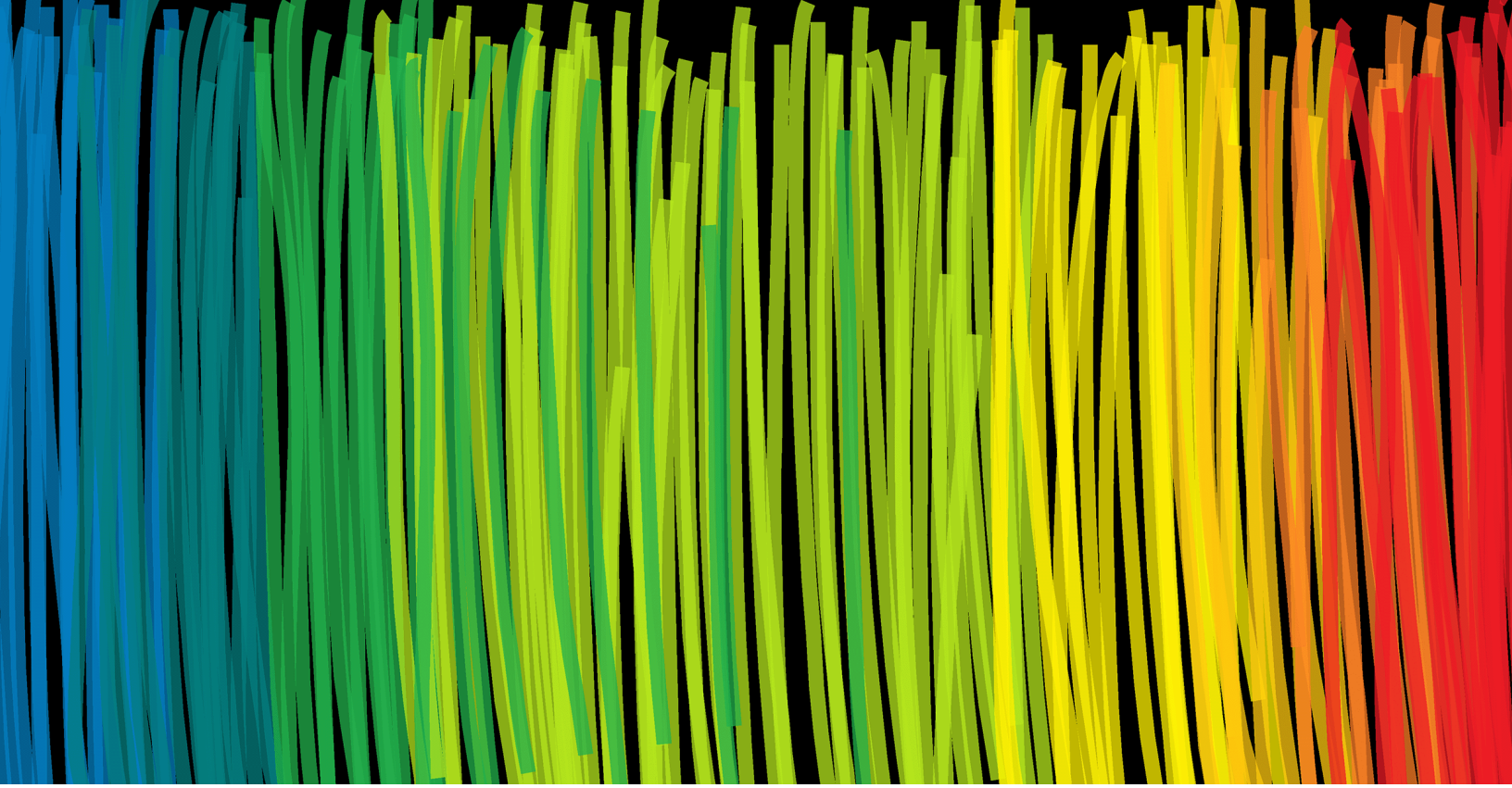
Handbook of Software Engineering Methods

LARA LETAW

OREGON STATE UNIVERSITY
CORVALLIS, OR

open.oregonstate.education/setextbook

Contents



Introduction

I won't tell you how to be a software engineer; you'll learn that over time by doing it.

Instead, **this book is about software engineering methods**—ways people achieve specific objectives in software engineering—that can save your project. My hope is that after reading this book (or parts of it), you'll feel better equipped for software engineering.

What's Software Engineering?

The **definition of software engineering** we will use is:

> "Systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software." (International Organization for Standardization et al., 2017)

The definition was agreed upon by the International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), and the Institute of Electrical and Electronics Engineers (IEEE) and published in their glossary of systems and software engineering vocabulary (International Organization for Standardization, 2017). The 500+ page document is meant to be internationally applicable to the field of information technology.

What's the Purpose of Software Engineering?

Software engineering can help people create sustainable, extensible programs that solve problems people care about. Sustainable means it's feasible for the program to grow, exist, and be maintained. Extensible means it's feasible to add more features.

What's the Philosophy Behind This Book?

My beliefs about software engineering influenced how I wrote this book. Some of my strongest beliefs about software engineering are described below.

Software Engineering Is Not Black and White

Throughout the book, I explain how software engineering is a gray area of computer science. "Right" answers can be hard to find and may not be reproducible in different contexts. Software engineering as a field also keeps changing as research scientists gather new findings, engineers develop new technologies, visionaries define new methods, and the outside world changes (e.g., a pandemic happened while I was writing the first edition of this book, and that changed how software engineering teams collaborate). Whereas in programming you might ask, "Is this algorithm correct?" questions in software engineering are more like, "How does my team know this software is ready to release?" or "People keep misinterpreting my code; how do I shift it toward better understandability and maintainability?"

It's Not Necessary to Study Every Detail of Software Engineering

I'm not going to tell you everything about software engineering because (1) what you need to know can be drastically different depending on context; (2) if I tried to, this book would be thousands of pages and possibly useless; and (3) many topics are best learned on the job. Instead, I'll introduce a set of software engineering methods that are known to be useful across multiple contexts, give guidance on when and why to use them, and point to resources for when you want more information.

Agile Isn't Perfect, But I Really Like It (and Other People Do, Too)

This book is geared toward Agile software development. That's because Agile development environments have become extremely popular—and because I like Agile. It matches how I think and has been appropriate for nearly all the projects I've worked on. But you're not me, and Agile isn't the be-all and end-all, so I'm planning to incorporate more from other software process models in the future.

What's This Book Like?

This book was written iteratively ("Do something. Do it again, but better") and incrementally ("Do a little more"). Lots of software is written the same way.

It has **eight major topics**:

1. **Agile**: Collaboration-oriented philosophy of creating software that values doing over comprehensive planning and documentation.
2. **Project Management and Teamwork**: Working in an organized way—and with other people.
3. **Requirements**: Being clear about what's expected of the software.
4. **Unified Modeling Language Class and Sequence Diagrams**: A couple types of diagrams useful for communicating how your code works (or should work).
5. **Monolith versus Microservice Architectures**: Two contrasting high-level ways to organize code.
6. **Paper Prototyping**: Creating a good user interface design before coding it.
7. **Inclusivity Heuristics**: Guidelines for making software work well for people who are not like you.
8. **Code Smells and Refactoring**: Making your code nicer to work with.

This book is **short** and meant to be **readable**.

- Important concepts are bolded.
- Glossary terms are italicized on their first use.
- Relevant side notes are embedded throughout.
- References are listed at the end of each chapter in case you need more information.

My aim is to enable you to quickly (1) determine whether each topic or method is relevant to your situation and (2) get a basic understanding of the topic or method so you can discuss it with others or have a starting point for exploring more.

What's New in the Second Edition?

Summary of changes:

- The text has been converted from PDF/LaTeX to HTML/CSS to improve accessibility.
- Hosting has moved from GitHub to https://open.oregonstate.education/.
- Alternative text has been added to all images.
- References, figures, and tables now conform to APA style.
- More in-line citations have been incorporated throughout.
- Renamed "Additional Resources" sections to "References" for clarity; less relevant references have been removed.
- Introduction: Added a section on changes to the new edition; adopted ISO/IEC/IEEE standard definition of software engineering; added a section about the purpose of software engineering; removed discussion of future additions because my ideas keep changing and I don't like making pseudo-promises; updated my contact information; updated acknowledgments.
- Chapter 2, "Project Management and Teamwork": Added a project network diagram figure, removed an unnecessary table to improve accessibility, and improved the summary.
- Chapter 3, "Requirements": Added examples for INVEST; added more quality attributes and explanations; added a "constraints" type of nonfunctional requirements; enhanced the user story, given-when-then, and Definition of Done examples; linked to more use case examples; added more information about what makes a good requirement; and improved the summary.
- Chapter 4, "Unified Modeling Language Class and Sequence Diagrams": Linked to more real-world diagram examples; increased the image size; removed an unnecessary table to improve accessibility; and clarified "association."
- Chapter 5, "Monolith versus Microservice Architectures": Added a technical case study.
- Chapter 7, "Inclusivity Heuristics": Updated name of the heuristics (previously called "Cognitive Style Heuristics"); added a spectra chart for more clearly explaining persona cognitive styles; added persona reactions to examples; reduced the number of examples; rewrote the whole chapter; added more brains to the artwork.
- Chapter 8, "Code Smells and Refactoring": Improved summary.
- Tweaked writing throughout.

Giving Feedback

I welcome your content requests, suggestions, and other feedback. Please email me at setextbook@lara.tech.

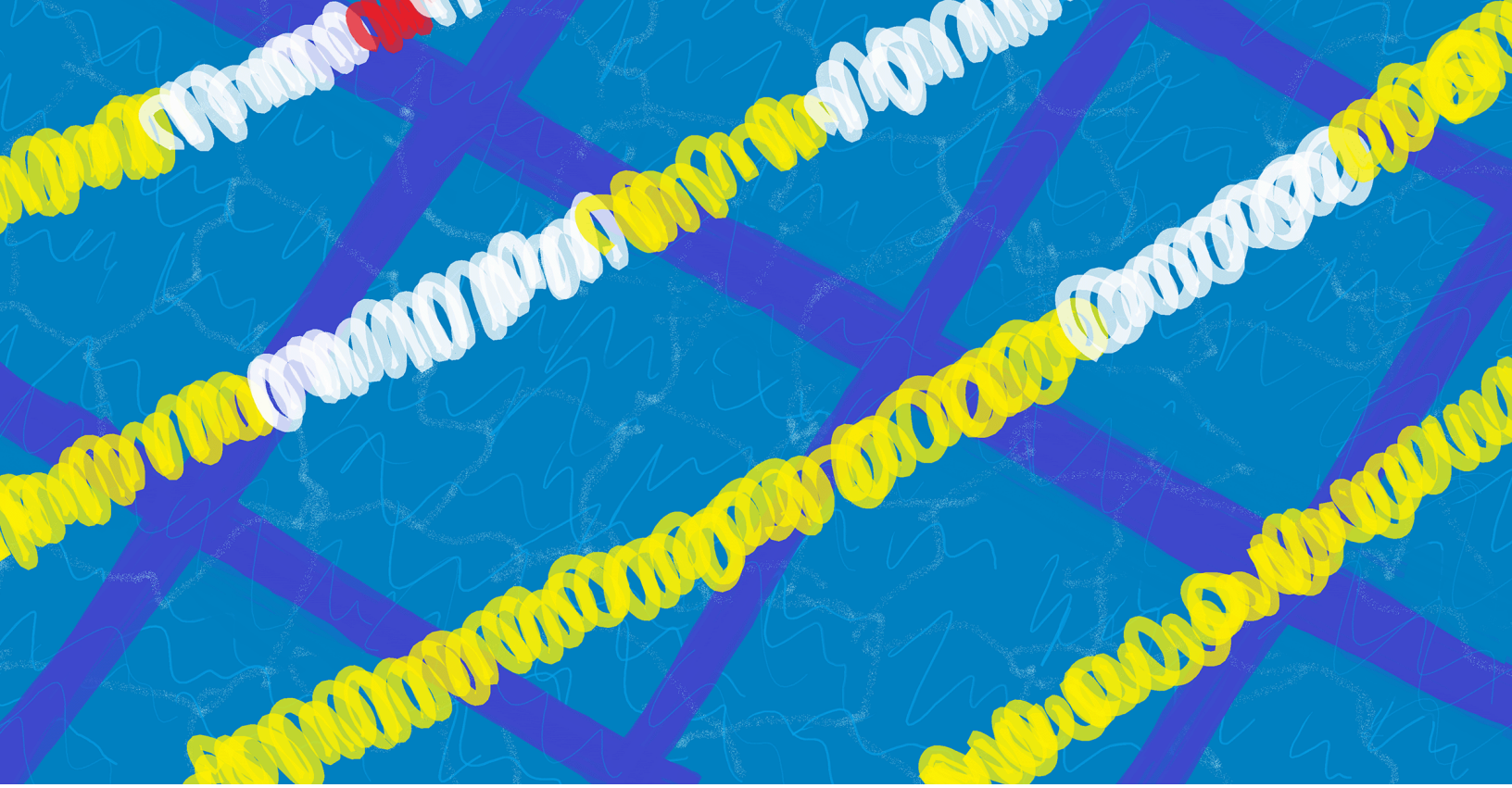
Acknowledgements

Thanks to Edward Isajanyan for helping make the second edition of this textbook screen-reader-friendly. Thanks to Caius Brindescu, Raffaele De Amicis, Sèanar Letaw, and Tiffany Rockwell for their feedback, advice, and support. Thanks to family and friends for their support. Thanks to the many software engineering students and other individuals who gave feedback, including Richard Brinkley, Maximillian Davensmith, Brian Doyle, Mark De Guzman, and Jack LaBarba. Thanks to Tom Weller and Scott Ashford for their letters of support. Thanks to Ashleigh McKown, the editor of this text. Thanks to the Oregon State University Open Educational Resources (OER) Unit, especially Stefanie Buck and Mark Lane, for making the whole effort possible.

Media Attributions

Reference

International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers. (2017). *Systems and software engineering—Vocabulary* (ISO/IEC/IEEE Standard No. 24765:2017). https://www.iso.org/standard/71952.html



Chapter 1

Agile

This book is geared toward Agile, but there are other software process models. Each has a different way of proceeding through the software development life cycle (SDLC). This chapter starts by describing Scrum, the SDLC, Agile, and a contrasting software process model called Waterfall. That is followed by a discussion of Scrum (an Agile framework) and notable Agile methods.

This chapter will give you the flavor of Agile and Scrum rather than being a comprehensive guide. For more detailed information about topics introduced here, see the References section at the end of the chapter.

1.1 The Software Development Life Cycle

The software development lifecycle (SDLC) is the progression of a software project through five SDLC stages:

1. **Requirements**: Figuring out and writing down what the software must do, how well, and under what limitations or constraints.
2. **Design**: Determining how the software's code will be structured and how users will interact with

the software.

3. **Implementation**: Using the requirements and design to code the software.
4. **Testing**: Checking that the code was written without fault (verification) and that the software is what the users or client wants (validation).
5. **Maintenance**: Improving software's existing functionality and code.

There are different ways to travel through the SDLC stages. Patterns of traveling through the stages are called software process models. Commonly, people compare the Agile software process model with the Waterfall model.

Agile, guided by the Agile *Manifesto* (Beck et al., 2001), moves through the SDLC approximately like in Figure 1.1.

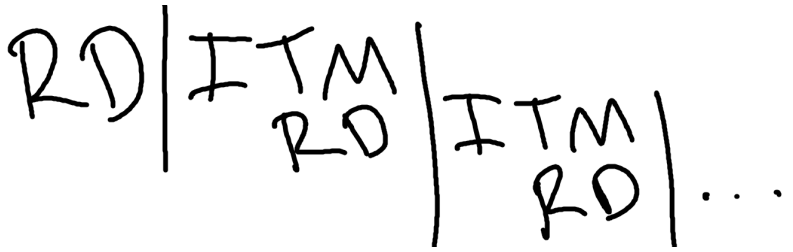


Figure 1.1 *How Agile Projects Move through SDLC Stages*

Note. The vertical lines represent development cycle boundaries. Planning (R,D) for the next development cycle starts during the previous cycle. R, requirements; D, design; I, implementation; T, testing; M, maintenance.

Agile development cycles are relatively short and numerous. Releases are frequent and incremental. Each cycle, there's a little more working functionality. There are multiple ways to go about developing and managing software in an Agile way, such as by using the Scrum framework (Schwaber & Sutherland, 2020) or Extreme Programming (XP) (Beck & Andres, 2004; Wells, 2013).

Waterfall moves through the SDLC approximately like in Figure 1.2.

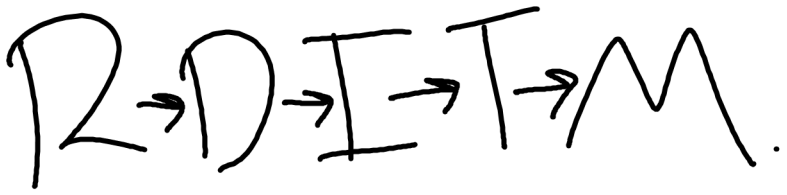


Figure 1.2 *How Waterfall Projects Move through SDLC Stages*

Movement is fairly linear and sequential. Each stage depends on the previous stage having been completed. Lots of documentation is produced.

Ironically, people often associate Waterfall with an article that describes Waterfall's major flaws. The second figure in Royce's (1970) article depicts the Waterfall model with seven stages and downward movement from one stage to the next, suggesting that movement to the previous stage is not allowed—you can't swim up a waterfall. Later in the article, Royce suggests modifications to the Waterfall model, such as making and implementing a preliminary program design (then going back to the requirements stage as needed).

Waterfall might not make sense for many software projects, but how about for building a bridge?

1.1.1 Why Care about Agile, Other Software Process Models, and Software Engineering Methods?

The 2015 CHAOS report contains aggregate data about more than 25,000 software projects.

Some findings about software projects:

9% of Agile projects failed
29% of Waterfall projects failed
23% of large Agile projects failed
42% of large Waterfall projects failed
4% of small Agile projects failed
11% of small Waterfall projects failed

- So you can **detect and/or understand what a software development team is doing**. When you're new to a team, having a general understanding of different software process models can **help you ask good questions, identify what you see the team doing, and look competent in front of your team and managers**.

- So you have **ideas** to choose from when you need to select a software process model or method for a new project. You might need to choose or recommend how your team proceeds.
- So you have ideas to choose from when a project is in trouble. According to CHAOS Report from the Standish Group International, Inc. (2015), during fiscal years 2011 to 2015, **17% to 22% of software projects failed** of the 25,000+ software projects in their database, with the likelihood of project failure **increasing drastically with project size**. Sometimes, you can save a project if you have the right methods.

Since this book is focused on Agile, the remainder of the chapter summarizes the Agile software process model, one Agile framework (Scrum), and a few Agile methods.

1.2 Agile, Scrum, and Agile Methods

1.2.1 Agile

The Agile philosophy is summed up by the Agile *Manifesto for Software Development* (Beck et al., 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.

Why does this book have a whole chapter about Agile and not one about Waterfall or any other software process model? Because most organizations use Agile for software or IT projects.

For example, an HP survey of 601 respondents (Hewlett Packard Enterprise, 2017) found the following distribution of what organizations use as their primary software process model:

- 51%: Leaning toward Agile
- 46%: Hybrid

- 16%: Pure Agile
- 7%: Leaning toward Waterfall
- 2%: Pure Waterfall

Why do organizations choose Agile? According to HP, out of 403 organizations that have primarily adopted Agile, the following percentage of respondents agreed with the following statements about Agile development:

- 54%: **Enhances collaboration** between teams that don't usually work together.
- 52%: **Increases** the level of **software quality** in organizations.
- 49%: Results in increased **customer satisfaction.**
- 43%: Shortens **time to market**.
- 42%: **Reduces cost** of development.

1.2.2 Scrum

Scrum is a well-known framework for software project management. It aligns with the Agile philosophy. For example, the Scrum Guide (the ever-evolving manual for Scrum; Schwaber & Sutherland, 2020) says that to reflect the "responding to change" value, a software project should be broken into development Sprints that are usually two to four weeks long. Each Sprint has a Sprint Plan. Sprint Plans can be defined shortly before the Sprint; Teams (and their customers) might only know what is happening with the project's development for a couple weeks at a time.

Scrum Teams fit their own methods into the Scrum framework, which the current version of the Scrum Guide divides into three categories: the **team**, the **events**, and the **artifacts**. To give you a quick, convenient introduction to Scrum, each element of the framework is listed below, by category.

The Team. The Scrum Team "consists of one **Scrum Master**, one **Product Owner**, and **Developers**."

- **Scrum Master**: "accountable for **establishing Scrum** as defined in the Scrum Guide."
- **Product Owner**: "accountable for **maximizing the value of the product** resulting from the work of the Scrum Team."
- **Developers**: "people in the Scrum Team that are **committed to creating** any aspect of a usable Increment each Sprint."

The Scrum Master's focus is **process**, the Product Owner's focus is the **product** (software), and the Developers' focus is **creating** a product while following Scrum processes.

The Events. There are five Scrum events:

- **The Sprint**: **fixed-length development periods** of "**one month or less** . . . A new Sprint starts immediately after the conclusion of the previous Sprint."
- **Sprint Planning**: "initiates the Sprint by **laying out the work** to be performed."
- **Daily Scrum**: "a **15-minute** event for the Developers of the Scrum Team . . . focuses on progress toward the Sprint Goal and produces an actionable plan for the next day of work."
- **Sprint Review**: "to **inspect the outcome of the Sprint** and determine future adaptations. The **Scrum Team** presents the results of their work to key **stakeholders . . .**"
- **Sprint Retrospective**: "to **plan** ways to increase quality and effectiveness . . . **Scrum Team** inspects how the last Sprint went . . ."

A Sprint is a development period that occurs in a series of Sprints, which are each laid out during Sprint Planning. Each day, the Developers have a 15-minute meeting about planning the next workday. Sprints end with a Sprint Review (team and stakeholders) and a Sprint Retrospective (team only).

The Artifacts. There are three Scrum artifacts:

- **Product Backlog**: "an emergent, **ordered list of what is needed** to improve the product."
- **Sprint Backlog**: "composed of the **Sprint Goal** (why), the set of **Product Backlog items selected** for the Sprint (what), as well as an **actionable plan** for delivering the Increment (how)."
- **Increment**: "a concrete **stepping stone toward the Product Goal**."

The Product Backlog contains a rough list of tasks the Scrum Team is planning to do some time, but the tasks haven't yet been scheduled and may not be defined in detail. The Sprint Backlog contains tasks the team has decided to work on and has added details about completing the tasks. An Increment is an achievement toward creating the product (e.g., finishing a feature implementation).

The *Scrum Guide* (Schwaber & Sutherland, 2020) describes the Scrum framework elements in more detail and defines some of the terms that were unexplained here (e.g., Sprint Goal).

1.2.3 Agile Methods

There are several notable Agile methods that can be used within Scrum (or other frameworks, or other software process models). A few of them:

- **Scrum board**: A way to organize and visualize tasks or work as cards on a board. The board has columns for different categories, and each card is placed within a column. A Scrum board could be a physical bulletin board with sticky notes or index cards. It is also a common feature of task management software.
- **Spike**: A quick and to-the-point investigation for gathering information to help the team answer a question or choose a development path.
- **User story**: A short description of a software feature from the perspective of fulfilling a user need (e.g., using this format: As a <role> I can <capability>, so that <receive benefit>). Tasks, priorities, time/cost estimates, and acceptance criteria may be associated with a user story.

1.3 Summary

"Agile" has associated values but no concrete meaning: it's a philosophy, and there's not just one way to follow it. Agile frameworks such as Scrum give more concrete guidance on software development and project management. Scrum is defined by the current version of the *Scrum Guide* (Schwaber & Sutherland, 2020), which changes frequently.

References

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile software development*. https://agilemanifesto.org/

Beck, K., & Andres, C. (2004). *Extreme programming explained: Embrace change* (2nd ed.). Addison-Wesley.

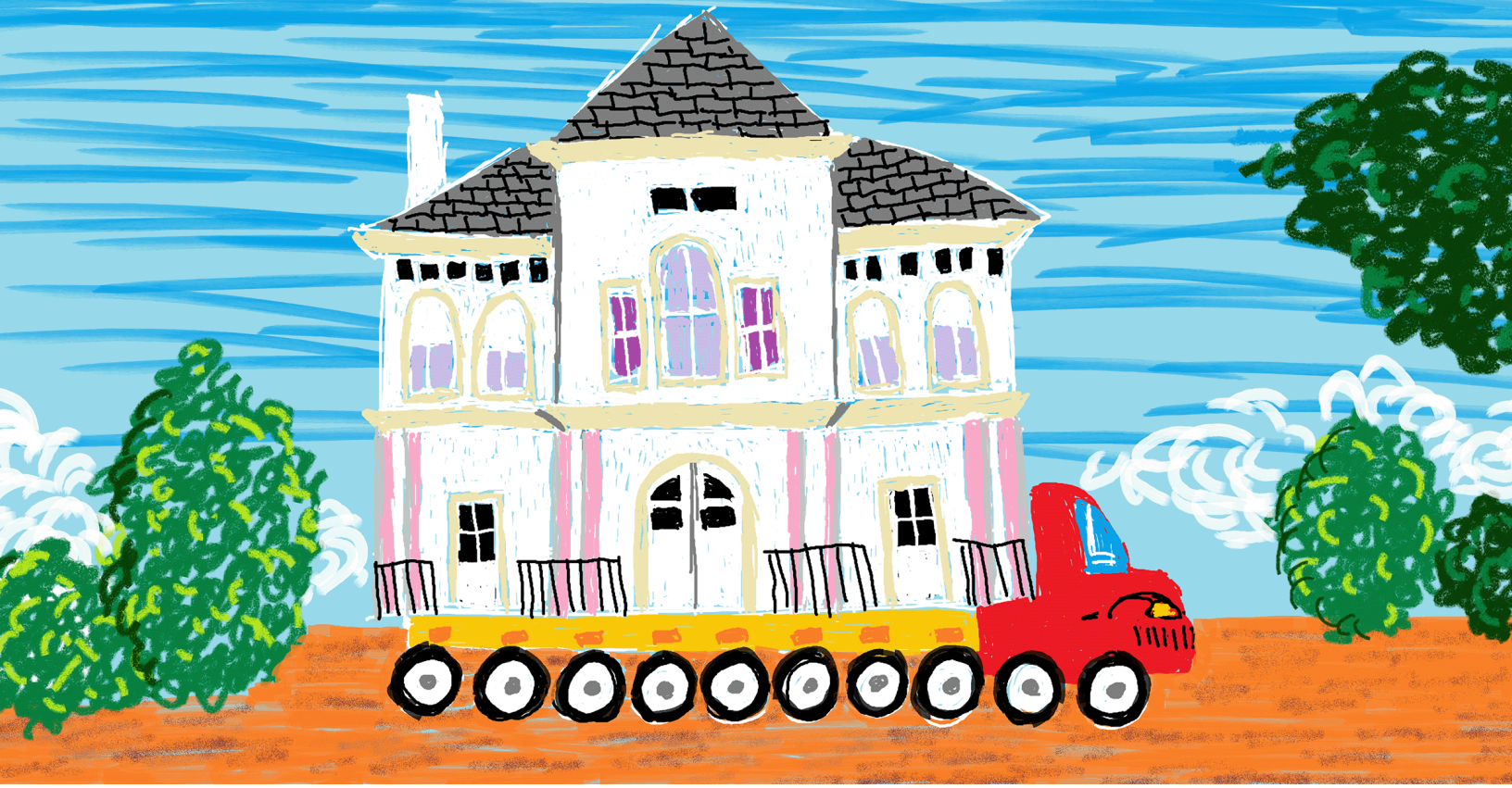
Hewlett Packard Enterprise (2017). *Agile is the new normal: Adopting Agile project management*. https://softwaretestinggenius.com/docs/4aa5-7619.pdf

Royce, W. W. (1970). Managing the development of large software systems. *Proceedings of IEEE WESCON, 26*, 1-9.

Schwaber, K., & Sutherland, J. (2020, November). *The 2020 scrum guide*. https://scrumguides.org/scrum-guide.html

Standish Group International, Inc. (2015). *CHAOS report 2015*. https://standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf

Wells, D. (2013, October 13). *Extreme programming: A gentle introduction.* http://www.extremeprogramming.org/



Chapter 2

Project Management and Teamwork

Project management is the process of planning and executing a project while balancing the time, cost, and scope constraints. Time, cost, and scope are known as the triple constraint.

How does one minimize time and money spent on a project while delivering an adequate feature set? Risk management is key. Risk is the estimated probability of a loss given a set of known and unknown factors. Risk can be stated as high, medium, low, or numerically. Ways to mitigate risk include **defining and keeping track of your project, communicating** with your project team, **researching the implications** of decisions, **developing backup plans**, and selecting **suitable tools**.

This chapter covers a variety of project management methods, including those related to **teamwork**. The methods are not limited to one type of software development environment, but this chapter, like the rest of this book, is geared toward Agile. There are many more methods that aren't discussed here; this chapter provides a starter set of well-known methods and highlights different areas of project management.

2.1 Why Learn about Project Management?

Why is there a chapter about project management if this book is intended for people who want to become or are currently software engineers?

- You might **become** a project manager (e.g., your employer asks you to fill the role, or you're interested in a new position).
- You might **have** a project manager. Understanding some basics of project management can help you understand what they're doing (e.g., using a RACI matrix to define who on the team does what) and what they're trying to tell you about the project.
- You might need to **self-manage** (e.g., within an organization that has a flattened hierarchy or within an Agile team).

2.2 Triple Constraint

Project management is partially about **optimization**: How can we use our limited financial and personnel resources to complete our project by the deadline, without going over budget? These concerns are often summarized as needing to balance three constraints.

> Authors in other fields sometimes consider quality separate from scope. In software engineering, requirements include quality.

- **Time**: duration of the project, intermediate deadlines
- **Cost**: monetary, personnel, and other project resources
- **Scope**: what the project is meant to accomplish and the requirements of the project, including quality

This set of three is called the triple constraint.

It can be difficult to balance these three constraints. Common challenges:

- You're meeting with a client who says, "Oh, I forgot to mention we want this feature. That won't be a big deal, right?"
- You realize late in the project that to implement feature A, you'll need to implement B, C, and D as well.
- Your team's estimates were overly optimistic (the planning fallacy).

These situations are so common that you can **assume they're going to happen** and come up with a mitigation plan even before the project starts. But many situations are more complicated (more factors with

more interrelationships), more unique to your context, and have factors that leak from your professional life to your personal life. Here are some examples:

- You're working on a project with a friend who is an excellent coder but only available for the next three months (**time**). They also have their own ideas about where they want the project to go (**scope**). You know your friend will be more enthusiastic about the project if they have more control, and that means quicker implementation and less work for you (**cost**). But that'd mean sacrificing some of your own feature priorities (**scope**).
- You're working with a five-person team. Your colleague needs help, but all hours must be billed to a project, you're getting pressured to stay close to the budget, and you bill at a higher rate than your colleague (**cost**). If your colleague doesn't get help, they might spend extra hours self-training (**cost**), could switch to a different project, and there's a small chance they'll make the project take longer (**time**). **Scope** is fixed: the product must satisfy all its requirements.

Making strategic project decisions involves adjusting project constraints. If you want to reduce time and cost spent on a project or increase project scope, you'll need a corresponding change in one or more other constraints (van Wyngaard et al., 2012).

2.3 Managerial Skill Mix

What skills are required for managing a project? There are three broad categories comprising the managerial skill mix (MSM) (Badawy, 1995).

- **Interpersonal**: Communicating effectively with anyone likely to affect the project (e.g., engineers on your team, managers, clients, contractors, IT support, etc.).
- **Technical**: Using methods and equipment effectively (e.g., knowledge of appropriate processes, understanding and writing code, etc.).
- **Administrative and conceptual**: Understanding the "big picture" vision (conceptual) and being able to move macro-level pieces (e.g., teams, departments, divisions, etc.) toward that vision (administrative).

High-level managers (e.g., CEOs) tend to need a different mix of skills than lower-level managers (e.g., project managers). For example, a project manager might need strong interpersonal and technical skills while only occasionally considering the big picture of how a project fits into organization's overall vision

(Badawy, 1995). Since this chapter is about project management, we will focus on interpersonal and technical skills.

2.4 Interpersonal Skills: Team Communication

One way to reduce risk is to improve team communication, which can increase the likelihood of project success.

As background for this section, consider Tuckman's model of team development (Stuart, 2014; Tuckman, 1965; Tuckman & Jensen, 1977).

1. **Forming**: Team members become oriented through testing each other's boundaries and establishing dependency relationships with peers, leaders, and existing team standards.
2. **Storming**: Team members resist group influence, their peers, their peers' ideas, and tasks.
3. **Norming**: Team develops cohesiveness, devises new standards and roles, and members express personal opinions related to tasks.
4. **Performing**: Team roles become flexible; team dynamics and structure serve the function of the team and task performance.
5. **Adjourning**: Team disbands.

The rest of this section discusses specific methods a team can use to improve communication. Consider where each might fit in to these stages (there's not just one answer).

2.4.1 Establishing Ground Rules

> When deciding on ground rules, your team might choose to incorporate ground rules or standards already established by others, such as the *IEEE Code of Ethics* (Institute of Electrical and Electronics Engineers, 2020) or *Agile Manifesto* (Beck et al., 2001).

Team ground rules are a preemptive or reactive method for reducing team conflict and dysfunction. Ground rules might already exist when a team forms, others might develop as the team becomes normalized, and revisions might happen as the team proceeds with their work and identifies new team concerns or opportunities. To be effective, the ground rules need buy-in from the whole team. What the ground rules should cover or should be varies by team, but below are some questions that may help.

- What is our **vision** for what this team is or what we're trying to accomplish together? (Clients choose us because we're honest and transparent.)
- What do we **prioritize** most? (Delivering a high-quality product ahead of the deadline, input from all team members, honoring diverse end-users, making the big bucks.)
- What methods will we use for **day-to-day communication**? (No interrupting, no 'splaining, listen to and acknowledge what other people are saying, ask people if they're busy before starting a long conversation).
- What methods will we use to **communicate** with each other **during conflict**? (We'll use nonviolent communication, we'll focus on the solving the problem instead of who to blame.)
- What expectations do we have for **work habits**? (Tuesdays from 1:00 to 3:00 p.m. is silent time; be five minutes early to meetings.)
- What expectations do we have for **responsiveness**? (Respond within two hours during regular work hours; have the team Discord open during regular work hours.)
- What will we do when team members **fail expectations**? (We'll discuss any team problems on Fridays at 3:00 p.m.)
- How will we **get to know each other**? (We'll discuss each other's cognitive styles; we'll set up a chat channel for socializing.)

The end product of answering questions like these could be a list of short statements that's posted somewhere people will see it regularly.

The questions your team asks, and the answers, will vary depending on the individuals on the team and on context (e.g., culture). Whatever those questions and answers are, ideally they will feel meaningful and authentic. If your team gets the feeling the ground rules are silly, phony, too aspirational, too inflexible, or too authoritative, that could invalidate your team's efforts toward creating the ground rules.

2.4.2 Defining Roles and Responsibilities: RACI Matrix

A RACI matrix is a chart for defining who is responsible (R) and accountable (A) for a task or deliverable, and who should be consulted (C) or informed (I).

Example RACI Matrix. A RACI matrix is often formatted as a table, but it can also be written as a list, as in the following example.

Project Phase: Minimum Viable Product (MVP)

- Focus groups
    - Frontend Developers: C
    - Frontend Designers: R
    - Frontend Lead: R / A
    - Backend Developers: C
    - Backend Lead: C
    - Team Lead: R / A

- Requirements specification
    - Frontend Developers: R
    - Frontend Designers: R
    - Frontend Lead: A / I
    - Backend Developers: R
    - Backend Lead: A / I
    - Team Lead: C / I

- Throwaway code design
    - Frontend Lead: I
    - Backend Developers: R
    - Backend Lead: A
    - Team Lead: I

- Implementation
    - Frontend Developers: R
    - Frontend Designers: C
    - Frontend Lead: A

    - Backend Developers: R
    - Backend Lead: A
    - Team Lead: C / I

- User acceptance testing
    - Frontend Developers: R
    - Frontend Designers: R
    - Frontend Lead: R / A
    - Backend Developers: R
    - Backend Lead: C
    - Team Lead: C / I

Interpreting a RACI Matrix. One person might have multiple roles. Task or deliverables can be organized into phases.

- **Responsible (R)**: Who will do the work.
- **Accountable (A)**: Who will approve the work and make sure it gets done.
- **Consulted (C)**: Who can discuss and offer advice about the work.
- **Informed (I)**: Whom to keep up to date about the status of the work.

A RACI matrix is a method for reducing risk. If your team doesn't know who needs to do what (or forgets, or can plausibly deny knowing), that can increase the probability of a negative events and outcomes (e.g., shipping a broken product to customers because nobody was assigned to quality assurance).

2.4.3 Measuring and Building Consensus: Fist of Five Method

> Meanings of single-finger hand gestures vary around the world. For example, in the United States, putting your thumb up means "good job," in Germany and Hungary it means "one," in Japan it means "five," and in Australia, Greece, and the Middle East it means "up yours!" (Cotton, 2013).

Fist of five is a method for checking and building consensus within a group of people. One person (e.g., team leader) makes a statement or proposes an idea to a group, and each person communicates their level or agreement or support by holding up a fist or up to five fingers. It has become associated with Agile (Belling, 2020), but students of different ages use it, too (e.g., Fletcher, 2002; Hulshult & Krehbiel, 2019).

What each number of fingers means:

- **None**: Strong reject. Blocks consensus.
- **One**: Reject. Major issues need resolving now.
- **Two**: Weak reject. Minor issues need resolving now.
- **Three**: Weak accept. Minor issues can be resolved later.
- **Four**: Accept. No issues.
- **Five**: Strong accept. Willing to lead or champion.

If anyone suggests rejecting the statement or idea by holding up two or fewer fingers, the team can stop, discuss, make changes, and vote again until there's sufficient consensus. It's up to the team or its leader to decide how much consensus is needed.

The fist of five method can reduce risk by (1) bringing problems to light and (2) increasing team motivation, ownership, and investment.

2.5 Technical Skills: Project Definition

This section contains methods for helping with the **technical side** of defining a project, including definition of scope, prioritization, estimation, scheduling, and task management.

2.5.1 Project Scope

In an Agile software development environment, a project's scope is implied through sets of tasks (e.g., release plan, Product Backlog, iteration plan, Sprint Backlog). Each iteration might have a goal (e.g., a Sprint Goal) that summarizes what the set of tasks is meant to accomplish, which is also part of defining scope for Agile projects. The scope is purposely flexible and emerges as the project proceeds.

In other environments, the project scope (a.k.a. statement of work) is a specific document stating the project's objective, deliverables (outputs), milestones, technical requirements, and limitations/exclusions.

2.5.2 Balancing Constraints: Project Priority Matrix

Above, we talked about the three major constraints of project management—time, cost, and scope—and that balancing them isn't always straightforward. What should the balance be? How do I know whether I'm achieving balance? How does this fit into how the project is run? One method for more concretely stating the desired balance is the project priority matrix. Table 2.1 shows a sample blank project priority matrix.

Table 2.1 Blank Project Priority Matrix

	Time	Scope	Cost
Constrain			
Enhance			
Accept			

- **Constrain**: The constraint is fixed (can get better but must not get worse).
- **Enhance**: Try to improve (e.g., take less time, spend less, have more features).
- **Accept**: Can worsen (e.g., more time, more personnel, fewer features) if necessary.

Table 2.2 shows a sample completed project priority matrix. For this example, imagine you have a grant from the National Institutes of Health (NIH) to write and test software for a medical device that automatically regulates a person's pain level.

Table 2.2 Completed Project Priority Matrix

	Time	Scope	Cost
Constrain			√ (cost)
Enhance		√ (scope)	
Accept	√ (time)		

Note. The text in parentheses is provided to make the table screen-reader-friendly.

Each checkmark in the filled example represents the following.

- **Scope**: Fixed. Your team must do what they said they'd do and cannot scrimp on quality. If the device only partially works, that would be a disaster—you'll be testing it on human subjects!
- **Cost**: Needs to be tightly controlled because the grant is for a fixed amount and funded by taxpay-

ers.

- **Time**: While the project hopefully stays on track and delivers as promised, if needed, your team can submit intermediate results to the NIH and perhaps use those results to get another grant.

Ideally, the project priority matrix would be defined before the project starts (with the client) and referenced throughout the project as needed. Developing and adhering to the matrix can reduce risk by helping the team or project manager balance constraints in ways that are acceptable to the client.

2.5.3 Task Prioritization: Eisenhower Matrix

Individual tasks, too, need relative prioritization. In an Agile Scrum environment, this would be the responsibility of the Product Owner and in Agile Extreme Programming (XP), it's the customer.

But how are task priorities decided? One high-level method is called the Eisenhower matrix (Table 2.3).

Table 2.3 Eisenhower Matrix

	Urgent	**Not Urgent**
Important	**Do**	**Decide**
Not Important	**Delegate**	**Delete**

Each cell in the Eisenhower matrix means the following.

- **Do** (urgent, important): Needs to be done correctly and now. An example is documenting your undocumented code so that a new hire can start contributing.
- **Decide** (not urgent, important): Needs to be done correctly but not immediately. An example is refactoring your currently working code. Such a task needs to be done eventually and done right—maybe the new hire can handle it in a couple months.
- **Delegate** (urgent, not important): Needs to be done now, but mistakes can be absorbed (e.g., tolerated, corrected later). An example is needing to initialize the task management system so the team can begin defining tasks. If it's not done right, that's fine—the developers and managers will adjust the setup as needed. The task would be a good learning task for the new hire, who doesn't have much to do right now.
- **Delete** (not urgent, not important): Doesn't need to be done correctly or any time soon. Can be eliminated. An example is implementing a loading screen that looks like a game of pong, but you're the only one on the team who thinks that's a cool idea.

Doing a first-pass task prioritization using an Eisenhower matrix can reduce risk by both **conserving resources** and **using resources thoughtfully** (including yourself). It can also help with getting out of the mode of "putting out fires" (concentrating on the urgent tasks), which can result in important but nonurgent tasks getting eternally left at the end of the to-do list (perhaps resulting in project failure).

2.5.4 Finer-Grained Prioritization

What happens when there are **multiple important tasks to complete that have the same level of urgency**? How does one decide which is more important? Here are some methods for deciding which task has higher priority when they seem roughly equivalent.

- For implementation tasks (e.g., coding, architecture, other implementation choices, etc.), **ask an expert**. They might know from experience which tasks have more unknowns, more risk, dependencies, and so on.
- If it's an implementation task and you're meant to be an expert, you can do a focused research effort called a spike to gather more information about the task, which in turn can help you prioritize it. **To do a spike**:

    1. Come up with a question.
    2. Try to figure out the answer by reading (e.g., documentation, other people's opinions) and experimenting (e.g., coding in a sandbox). You will probably get ideas for more questions in the process.
    3. Repeat until you have enough information.

    A good way to do a spike is to start doing the task and see what obstacles you run into. **Example**: You need to set up a local server for testing and then write a test suite. You have experience writing test suites but have never set up a server. After doing a spike, you realize that some of the tests you're going to write rely on the local server having a static IP address, which you learned is not the default. Based on your findings, you decide to prioritize the server setup because (1) the test suite depends on it and (2) the server setup task still has many unknowns, and you're not sure how long it'll take to eliminate those.
- Think about **dependencies**: Who's waiting on you to complete the task? How many other tasks depend on this task? **Example**: You estimate it'll take 15 minutes to complete a task that two other people are waiting on. You decide to do that before your four-hour task. Seems like the obvious choice—but if you're not aware of which tasks depend on yours or are deep into solo work mode,

you might make a suboptimal choice.

- If you're deciding which feature to implement, you can **ask the customer or users** directly (e.g., through a phone call, focus group, survey) or indirectly (e.g., by looking at support tickets, asking the marketing team, detecting an unmet need based on how people use other software).
- Other ways to select features include **voting** (e.g., within your team) or pairwise comparison (e.g., Is Feature A more valuable than Feature B? If so, is Feature C more valuable than Feature A?).

A natural side effect of prioritization is finding how long it'll take to complete a task, what dependencies exist, who the players are, and what the end user wants. All this knowledge contributes to risk mitigation.

2.5.5 Estimation: Story Points, Ideal Days, and Planning Poker

Intertwined with prioritization is estimation, or figuring out ahead of time how long a task is likely to take. But what does "how long" mean, and how do we figure it out ?

According to the Agile community (Cohn, 2006), there are **two methods for stating the size of a task**.

1. Story points: Assign a number to a task representing its size relative to other tasks. For example, a software installation and a virus scan might both be a 1 if they take roughly the same amount of time and effort, have roughly the same amount of risk, and the like. Implementing a major feature might, however, be an 8. Your team decides how far the scale goes.
2. Ideal days: Assign a number of days you think it'd take to complete the task if there were no other tasks or distractions. For example, if it takes me 5 minutes to remove a single square foot of grass from my lawn and I have 100 square feet to remove, that is 8 hours and 20 minutes total, so about one ideal day (if your workdays are 8 or 9 hours).

> Common scales for story points are 1 to 10, Fibonacci, and powers of 2. The latter two are meant to help make sizing a task easier by putting more distance between the numbers in the scale; deciding between a 4 and an 8 can be easier than deciding between a 4 and a 5.

Once story points or ideal days are assigned, a team can make statements like, "This month, we will complete 50 story points," "10 ideal days," and so on. Work completed (in story points or ideal days) is, in Agile teams, called the velocity. Teams can make initial estimates about velocity and then adjust depending on how accurate those estimates end up being.

But **how are estimates assigned** to a task? Another Agile idea is planning poker (Cohn, 2006; Mahnič & Hovelja, 2012). With this method, the team gets together to discuss a set of tasks, and each person gets a set of cards with the different possible story points, ideal days, or other aspects a task can be assigned. One person describes the task, the team asks questions as needed, and then each person privately decides on an estimate by selecting a card (keeping it face down or hidden). Once everyone is ready, the cards are revealed. Variations in estimates are expected, and part of the process: differences open a discussion. Someone making a high estimate, for example, may think of good reasons why a task is likely to take a long time. Someone making a low estimate may identify an efficient idea nobody else thought of. The team discusses and, once ready, can repeat the process until estimates become sufficiently consistent.

2.5.6 Scheduling: Project Network Diagram

Once a set of tasks has been defined, prioritized, and estimated, those tasks can be scheduled. Scheduling a task is placing it within the time line and context of a project. The context of a project includes other tasks, personnel, and non-personnel resources (e.g., equipment), and milestones. One method for defining and visualizing a project's schedule is using a project network diagram, which is a directed graph showing a project's tasks, the sequence in which they should be completed, and the dependency relationships between the tasks. The nodes in the digraph represent tasks, and the lines with arrows represent dependency or sequence relationships. A project network diagram moves left to right, where left is earlier in time. Figure 2.1 shows an example.

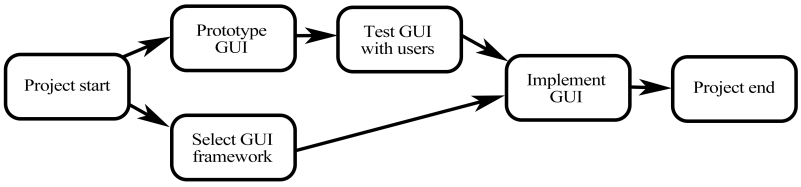


Figure 2.1 *Example Simple Project Network Diagram (No Estimates)*

Note. This format of project network diagram is called Activity-on-Node (AON) (Larson & Gray, 2018).

For a task to be represented as a node on a project network diagram, it needs to (at a minimum) be distinct from other tasks, and its dependent tasks (a.k.a. predecessors) must be known. A project network diagram becomes more useful if estimates for the tasks are also known, however.

Constructing a project network diagram. A project network diagram can be created manually or automatically generated by software. To automatically generate a project network using software (e.g., MS Project, Lucidchart), you'd need to enter the project data in tabular form, such as in a spreadsheet. Table 2.4 shows an example.

Table 2.4 Project Scheduling Data

Task ID	Task	Predecessors	Duration (hours)
4	Implement GUI	1,3	50
3	Test GUI with users	2	5
2	Prototype GUI		8
1	Select GUI framework		2

Note. These data could generate the project network diagram in Figure 2.1.

In Table 2.4, even though Task 2 must happen before Task 4, it's not listed as a predecessor because it's not an immediate predecessor.

Depending on the software you choose for creating your project network diagram, you might have access to more complex options like specific dates by which individual tasks must be completed.

2.5.7 Task Management Systems

A task management system can be used to organize tasks, task details (e.g., description, acceptance criteria, assignee, status), and other relevant information (e.g., which iteration or phase the task belongs to). They're useful for organizing and storing information about tasks, but also for the satisfaction of marking a task as done! Task management systems like Asana, Jira, and Trello are strongly oriented toward team collaboration. Some of these systems are also Agile oriented in that they offer Agile-inspired features (e.g., templates).

Common features of task management systems:

- Create, remove, update, and delete tasks.
- Enter task name, description, notes/comments, and add attachments.
- View tasks as a list, as cards on a board, or within a time line (e.g., Gantt chart).
- Organize tasks into projects.
- Assign tasks to different team members, with due dates.
- Enter task status (e.g., in progress, done).
- Get email notifications about tasks.
- Add tags, keywords, and categories.

Task management systems don't have a universal way to generate project network diagrams. For that, you might need a fully featured project management system (e.g., MS Project). But you may find that a Gantt chart or road map feature meets your needs and is available within your task management system.

2.6 Summary

Project management and teamwork can reduce the risk of a project failing and make it possible to complete larger projects. Part of good project management is balancing time, scope, and cost (the triple constraint).

Project management methods to help with team communication include establishing ground rules, defining roles and responsibilities in a RACI matrix, and measuring and building consensus using the fist of five method.

Methods for defining a project include prioritizing tasks using a project priority matrix or an Eisenhower matrix, estimating tasks using story points, ideal days, and planning poker, visualizing a project schedule using a project network diagram, and using the features of a task management system.

References

Badawy, M. K. (1995). *Developing managerial skills in engineers and scientists: Succeeding as a technical manager*. Van Nostrand Reinhold.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile software development*. https://agilemanifesto.org/

Belling, S. (2020). Agile values and practices. In *Succeeding with Agile Hybrids*, 47–61. Springer.

Cohn, M. (2006). *Agile estimating and planning*. Prentice Hall Professional Technical Reference.

Cotton, G. (2013, August 13). *Gestures to avoid in cross-cultural business: In other words, "keep your fingers to yourself!"* HuffPost. https://www.huffpost.com/entry/cross-cultural-gestures_b_3437653

Fletcher, A. (2002). *Firestarter Youth Power Curriculum*. Freechild Institute for Youth Engagement. https://freechildinstitute.files.wordpress.com/2023/04/firestarter-participant-guidebook.pdf

Hulshult, A. R., & Krehbiel, T. C. (2019). Using eight agile practices in an online course to improve student learning and Team Project Quality. *Journal of Higher Education Theory and Practice*, *19*(3). https://doi.org/10.33423/jhetp.v19i3.2116

Institute of Electrical and Electronics Engineers. (2020, June). IEEE code of Ethics. IEEE. https://www.ieee.org/about/corporate/governance/p7-8.html

Larson, E. W., & Gray, C. F. (2018). *Project management the managerial process*. McGraw-Hill Education.

Mahnič, V., & Hovelja, T. (2012). On using planning poker for estimating user stories. *Journal of Systems and Software*, *85*(9), 2086–2095. https://doi.org/10.1016/j.jss.2012.04.005

Stuart, A. (2014). *Ground rules for a high performing team*. Paper presented at PMI Global Congress 2014—North America, Phoenix, AZ. Project Management Institute.

Tuckman, B. W. (1965). Developmental sequence in small groups. *Psychological Bulletin*, *63*(6), 384–399. https://doi.org/10.1037/h0022100

Tuckman, B. W., & Jensen, M. A. (1977). Stages of small-group development revisited. *Group and Organization Studies*, *2*(4), 419–427. https://doi.org/10.1177/105960117700200404

van Wyngaard, C. J., Pretorius, J. H., & Pretorius, L. (2012). *Theory of the triple constraint—A conceptual review*. Paper presented at the 2012 IEEE International Conference on Industrial Engineering and Engineering Management, Hong Kong, China. https://doi.org/10.1109/ieem.2012.6838095

Chapter 3

Requirements

A software requirement is a rule the software must conform to: what it must do, how well, and within what constraints or limits.

3.1 Types of Requirements

There are two main **types of requirements**:

1. Functional requirements are "A description of a behavior that a system will exhibit under specific conditions" (Wiegers & Beatty, 2013, p. 599). For example, "If the user activates the 'log in' button, the login page will appear." Functional requirements answer the question, "What must the software do?"
2. Nonfunctional requirements are "A description of a property or characteristic that a system must exhibit or a constraint that it must respect" (Wiegers & Beatty, 2013, p. 600). For example, "If the user activates the 'log in' button, the login page will appear within 500 milliseconds." This nonfunctional requirement has a characteristic that the system must exhibit: responsiveness. Responsiveness is also called a quality attribute. An example of a nonfunctional requirement about

respecting a constraint is, “The GUI toolkit must be able to display non-rectangular windows.”

Figure 3.1 shows a simple example of a design failing to reflect a nonfunctional requirement and a functional requirement.

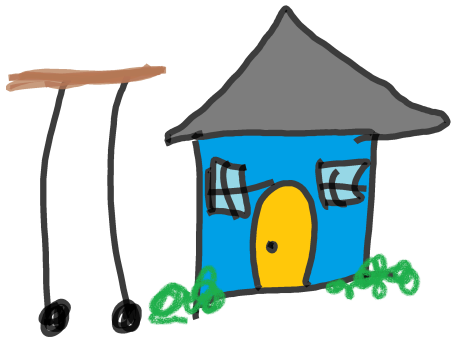


Figure 3.1 *Two Failed Requirements*

Note. This rolling table fails the nonfunctional requirement of fitting through an average door and the functional requirement of having four legs.

3.2 Why Requirements Matter

The design and implementation of software should, ideally, follow from the requirements. Here are some **ways requirements are helpful** and reasons they are important:

- When developers aren’t given requirements, they **might prioritize functionality they personally think is important or fun** to implement, but what developers want to implement might not make the project successful.
- When multiple developers are working on the same code, requirements can **help them stay in sync** and **pursue the same goal**. Without requirements, time, effort, and money can be wasted implementing conflicting code.
- When requirements aren’t specified, it’s easier for project stakeholders (e.g., clients, partners, investors, consultants, management) to **influence the project toward satisfying their own** (possibly fleeting) **wants or needs**. This can result in the project drifting away from what it was originally intended to do—and can lead to project failure.
- Requirements are **helpful for communicating** about software with stakeholders, **keeping track** of everything that needs to get done, and helping you and the client **decide what really needs to get**

done (clients sometimes don't know what they really need).

3.3 What Makes a Good Requirement

Teams or organizations can choose their own standards for what makes a good requirement. Here is **one set of standards** (Texas Department of Information Resources, 2008):

Requirements should be . . .

- **Correct**: What they say is right.
- **Unambiguous**: There is only one way to interpret them.
- **Complete**: They cover all that's important.
- **Consistent**: They aren't contradictory.
- **Ranked for importance and/or stability**.
- **Verifiable or testable**: There's a way to figure out if they're satisfied.
- **Modifiable**: They can be changed.
- **Traceable**: It's possible to figure out where they came from.

Requirements should also be . . .

- **Cross-referenced to earlier documents** that relate.
- **Uniquely identifiable**.
- **Organized for maximum readability.**

3.4 Requirements Elicitation

The process of gathering requirements is called requirements elicitation. Requirements can come from any stakeholder, including clients, managers, users, governments, developers of software to be integrated with yours, the development team, and yourself. Requirements elicitation involves both detecting stakeholders' wants and needs and using your professional judgment to decide which requirements to focus on.

To detect stakeholders' wants and needs, communicate and observe. Some methods:

- **Interviews**: Structured (questions defined ahead of time), semi-structured (some questions predefined, some generated during interview), or unstructured conversations.
- **Focus groups**: Small, group conversations in which the participants discuss topics among themselves, with moderator guidance.
- **Lab studies**: Participants perform tasks in a controlled setting (e.g., try to use an early prototype, then give feedback).
- **Exploratory research**: Multiple methods of immersing oneself within the world of relevant people and products, with the purpose of gaining knowledge and developing empathy for stakeholders. For example, after doing a fly-on-the-wall observation, you realize that people can't find Aisle 25 because it's in an unexpected place. You decide to prioritize the Aisle Map feature in the store's app.

Depending on the software development environment, **these methods might be the jurisdiction of specialist researchers in marketing or interaction design**. Hanington and Martin (2019) describe these specialist methods (and many other relevant methods) in more detail.

Developers can elicit requirements, too, by having conversations with stakeholders. There are **factors that can affect the success** of that approach, however.

- **Stakeholders might not have experience or expertise**. Developers can help bridge the gap between what the stakeholder wants and what is technically feasible and reasonable (e.g., given time, cost, and scope, what is also known as the triple constraint).
- **Stakeholders might not have good ideas**. They might be incorrect about what they or other people want or will use. Developers can sometimes provide guidance toward better ideas, but developers can also have bad ideas. Methods such as focus groups, usability testing, and releasing a minimum viable product (MVP) can help with figuring out whether users will use (and pay for) the software.
- **Stakeholders might not know what they want**. They may have a rough idea, or an idea that's at odds with their wants or needs.
- **Stakeholders might want what's bad for them or others**. For example, users want apps that make their face beautiful in photos, such features may promote unrealistic beauty standards.
- **Stakeholders are humans**. They communicate imperfectly.

With experience, you can learn how to effectively gather relevant information from stakeholders and make your own judgments about how that information translates into requirements.

3.5 Nonfunctional Requirements

Nonfunctional requirements describe how well the software needs to perform or what constraints it must respect.

Examples of nonfunctional requirements:

- Response time should be a few seconds or less in all operating environments.
- The front-end design must be evaluated using the Inclusivity Heuristics by at least two people each Sprint.
- The software must be available 24 hours a day, seven days a week, and must have an uptime of 99.99%.

Notice that **each nonfunctional requirement has a quantity**. That helps make it testable (a criterion for a good requirement).

3.5.1 Quality Attributes

> There is a long list of quality attributes on Wikipedia's "List of system quality attributes" page (Wikimedia Foundation, 2023).

Quality attributes are words for describing "a service or performance characteristic of software" (Wiegers & Beatty, 2013, p. 601). **Some common quality attributes** are as follows.

- **Maintainability**: Amount of effort needed for developers to update, refactor, or otherwise modify the software's code.
- **Portability**: Amount of effort needed to run the software on different platforms.
- **Reliability**: How often the software's functions succeed or fail.
- **Efficiency**: Number of resources the software requires.
- **Integrity**: How frequently the software loses data.

- **Memorability**: Amount of time users must spend relearning functionality.
- **Flexibility**: Number of different ways the software can be used.
- **Interoperability**: Ease with which the software can integrate with other software.
- **Reusability**: Extent to which the code can easily be used to solve other problems.

Each quality attribute can be converted to a scale. For example, the lowest value on a reliability scale for a single could be "the function succeeds 0% of the time," and 100% would of course be the opposite pole. Given this scale, we can specify a nonfunctional requirement by defining a performance threshold:

- The function must have high reliability (succeeds >99% of the time).

When you select quality attributes for your software, you are prioritizing what qualities matter most to you/your team/the project. Ideally, your team would keep these quality attributes (and the corresponding nonfunctional requirements) in mind for the duration of the project. If the software is not meeting the nonfunctional requirements, either the software or the threshold of acceptability needs to change.

3.5.2 Constraints

Some nonfunctional requirements are not about quality attributes and are instead about staying within constraints. The following are **example types of constraints** (Wiegers & Beatty, 2013):

- Those limiting technology choices (programming languages, frameworks, databases, application programming interface (API) types, etc.).
- Those limiting what platforms are targeted (e.g., mobile versus desktop, iOS versus Android).
- Those limiting what about the software can change (e.g., for backward compatibility).
- Those limiting how code can be written (e.g., following particular coding and documentation standards).
- Those limiting how data can be handled (e.g., must only be stored on US servers).

A conceptual difference between constraints and quality attributes is that constraints are often externally mandated, while quality attributes can be chosen internally by the team.

3.6 Functional Requirements

Functional requirements described what the software must do.

Example functional requirements:

- When the "register" button is activated, the user's information is added to the database and a "thank you for registering" screen displays.
- As a wholesaler, I want to see the wholesale and retail prices when I go to "product view" so that I know how much money I'm going to make.
- Given a user has performed at least one editing action, when they activate the "action history" window, they see a list of editing actions they have taken.

Each of these functional requirements is formatted differently. There isn't a name for the first format; it simply states what should happen when a particular action is taken in the software. The second uses user story format, which is common in Agile software development. This format emphasizes the user, what the user is trying to do, and their motivations. The third requirement uses the given-when-then format (see Agile Alliance for more information), which incorporates context. This format is commonly used to write user story acceptance criteria: a set of statements that, when true, indicate that the user story has been completed.

A more formal way to write functional requirements is the use case format, which follows a template. Figure 3.2 contains an **example use case using a simple template**.

Name: Generate list of recovered patients

Actor: Clinician

Flow:

1. Clinician authenticates using smart card.
2. Software confirms user credentials and permissions for specific machine.
3. Software logs access.
4. Software displays patient search.
5. Clinician selects “Advanced Patient Search.”
6. Software confirms user access permissions for advanced search page.
7. Clinician selects ailment and patient status.
8. Clinician executes search using “Search” button.
9. Software returns results.
10. Software logs query.

Figure 3.2 *Simple Use Case*

3.6.1 User Stories

User stories are a method for specifying functional requirements. They describe a small piece of the software’s functionality in a simple and easy-to-read sentence. They are written in plain English so that nontechnical people (e.g., users, clients, other stakeholders) can understand them.

The body of a user story is commonly written using this format
As a <ROLE>, I want <SOME FUNCTIONALITY> so that I get <SOME BENEFIT>

User stories can be written on 3 × 5 index cards and then stuck on a wall or whiteboard. They can also be typed into task and project management systems (e.g., Jira, Asana, and the like). Figure 3.3 provides a few examples of user stories within the context of a project (they have priorities and other project-related information attached to them).

US-023: Disabling Comments
Priority: Highest (8)
Sprint: 2
Assigned to: Emrah Tuukka

As an admin, I want to disable comments so that I can control spam and spread of disinformation.

US-034: Personalized Avatar Background
Priority: Lowest (1)
Sprint: 3
Assigned to: Ade Einarr

As a registered user, I want to change the background around my face on my avatar so that I can personalize my experience.

US-012: App Purpose
Priority: Highest (8)
Sprint: 1
Assigned to: Randomira Philibert

As a new user, I want to read about what features the app provides so that I can decide whether to use it.

Figure 3.3 *User Story Functional Requirement Examples*

Want more examples of user stories? Mountain Goat Software provides 200 example user stories [PDF] (Cohn, 2004). They list only the "As a . . ." part of the user story requirement.

Anyone on the team—or any project stakeholder—might come up with user stories. Once the user stories are initially defined, they can be used to start a conversation with the client and others on the team. Clients can guide you on setting priorities for user stories. This conversation is also a good time to get more details about the user stories, which should be added to the card.

Want examples of comically bad user stories? Check out the Shit User Story Twitter feed (@ShitUserStory)

What makes a good user story? Besides the characteristics of good requirements listed earlier in this chapter, the *INVEST* acronym (Wake, 2003) can help you remember characteristics of good user stories:

- (I) **Independent**: Does not have unnecessary dependencies or overlap with other user stories.
    - Two user stories that overlap:
        - "As a new user, I want to register so that . . ."
        - "As a new user, I want to register using my Google account so that . . ."
    - Set of user stories that don't overlap (but some user stories need to be completed before others—that's ok):
        - "As a new user, I want to view the registration page so that . . ."
        - "As a new user, I want to register using Facebook so that . . ."
        - "As a new user, I want to register using Google so that . . ."
        - "As a new user, I want my registration details to be stored so that . . ."

- (N) **Negotiable**: Encourages instead of discourages discussion and gives developers flexibility.
    - Does not encourage discussion: "As a logged in user, I want to choose either black or white so that . . ."
    - Encourages discussion: "As a logged in user, I want to choose from multiple colors so that . . ."

- (V) **Valuable**: Fulfills a user need.
    - Does not fulfill a user need: "As an Enterprise user, I want to watch a little race car drive around the screen so that I can do something fun while requesting API end points."
    - Fulfills a user need: "As an Enterprise user, I want to import my API end point requests so that my requests take less time and are less tedious."

- (E) **Estimable**: Can be given a time estimate.
    - Difficult to give a time estimate: "As a new user, I want enough encouragement to register so that I'll register."
    - Easier to estimate: "As a new user, I want to compare plan pricing so that I can decide which plan to choose."

- (S) **Small**: Can fit into a single development period (e.g., a two-week Sprint)
    - Probably too large for a Sprint: "As a user, I want to play chess on my phone so that I have

something to do while waiting at the pharmacy."

    - Smaller: "As a user, I want to move my pawn so that I can take my turn in chess."

- (T) **Testable**: Possible to determine it's done.
    - Difficult to determine whether it's done: "As a guest user, I want to be satisfied with my experience so that I will want to sign up."
    - Less difficult: "As a guest user, I want to try out the AI text generator without registering first so that I can decide whether to subscribe."

There is some overlap between INVEST and the general characteristics of good requirements mentioned above (which is comforting), but you might find that INVEST is easier to remember.

How do you know when a user story is done? This is negotiated with the client and added to the user story as acceptance criteria. Acceptance criteria say what must be true about the functionality specified by the user story for the user story to be considered done (i.e., establishing the Definition of Done for the user story). Figure 3.4 adds a DoD to one of the user stories from Figure 3.3. The DoD is composed of acceptance criteria following the given-when-then format.

US-023: Disabling Comments
Priority: Highest (8)
Sprint: 2
Assigned to: Emrah Tuukka

As an admin, I want to disable comments so that I can control spam and the spread of disinformation.

Definition of Done

- Given the user is logged in as a user, when they navigate to "Settings," then there is a "Disable Comments" button.
- Given the user is on the "Settings" page, when they activate "Disable Comments," then a status message appears that indicates the action was successful. The message appears within 10 milliseconds.
- Given the user has activated "Disable Comments," when they navigate to a "Post" page, then "Comments disabled" appears in the "Comments" section, and no comments are showing.

Figure 3.4 *User Story with Definitions of Done Example*

Once each of the acceptance criteria are confirmed to be done, the user story can be considered "DONE-done."

Ideally, testing the acceptance criteria can be automated. Figure 3.5 provides example pseudocode for testing an acceptance criterion.

```
def test_go_to_time():
  # given
  assert os.isWindows(),"Not Windows!"
  player.open()
  player.play_video('test.mkv')

  # when
  user.send_keyboard_shortcut("Ctrl-T")

   # then
   assert player.screen.is_showing(GOTOTIME)
```

Figure 3.5 *Pseudocode for Testing an Example Acceptance Criterion*

3.6.2 Use Cases

Use cases are a more formal method of specifying functional requirements. They are structured descriptions of what a system is required to do when a user interacts. Figure 3.2 showed a simple use case example, and additional examples can be found in the Digital.gov Usability Starter Kit PDF about use cases and personas (US General Services Administration, 2014).

As use cases are less common in Agile, the remainder of this section will provide only a summary of how use cases are structured.

Required Parts of a Use Case

Every use case has the following.

- **Name**: A short title for the use case that often starts with a verb (e.g., "Schedule weekly wellness check"). The name briefly states the user objective the use case will describe.
- **Actor(s)**: The user or users (human/nonhuman/computer) that are interacting with the software (e.g., "Medical staff").
- **Flow of events**: Sequence of actions describing the interaction between the actor and the software (a.k.a. "basic course of action" or "success scenario").

Sometimes, the actor is implied through the flow of events (e.g., "Shopper selects the calendar icon"). Other times, the actor is stated separately from the flow of events (e.g., "Actor: Shopper").

Additional Parts of a Use Case

The following are sometimes included in use cases.

- **Identifier**: A unique way of referring to the use case (e.g., UC-002).
- **Preconditions**: What must be true before the flow (e.g., "The shopper has added at least one product to their shopping cart").
- **Postconditions**: What must be true after the flow (e.g., "The shopper received an order confirmation email").
- **Business relevance**: Justification for why the use case exists.
- **Dependencies**: Other use cases the use case relies on. The unique identifier is handy for this part.
- **Extensions**: Contingencies, alternate routes, and branches to other use cases.
- **Priorities**: The importance of the use case.
- **Nonfunctional requirements**: How well the software must perform during the flow.

3.7 Requirements Specification

The process of writing down requirements is called requirements specification. Used as a noun, requirements specification refers to the document that contains the requirements. That document may also be a software requirements specification (SRS). The best way to learn about SRSs is to look at some.

> Another type of software document, which is sometimes confused with an SRS, is a software design document (SDD). If the SRS is what the software should be, the SDD is what the software is. There is often overlap between these two documents.

Freely available SRS examples (including some for open source software):

- SRS for apps and a data repository for distributing manufacturing data [PDF] (Hedberg et al., 2017).
- SRS for data system that assesses conservation practices [PDF] (CEAP, 2006).

- SRS for an app that splits and merges PDFs [PDF] (Spyridonos, 2010).
- SRS for software that processes electroencephalography data [PDF] (OpenVIBE, 2018).
- SRS for library software [PDF] (Eaker, 2006).

If any links are broken, try the Wayback Machine.

3.8 Summary

There are two main types of software requirements: functional and nonfunctional.

A functional requirement is "a description of a behavior that a system will exhibit under specific conditions" (Wiegers & Beatty, 2013, p. 599). There are different formats for writing functional requirements such as the given-when-then format, user stories, and use cases. User stories are used in Agile software development. The given-when-then format is used for writing user story acceptance criteria. A set of acceptance criteria is used to determine whether a user story has been completed (Definition of Done).

A nonfunctional requirement is "a description of a property or characteristic that a system must exhibit or a constraint that it must respect" (Wiegers & Beatty, 2013, p. 600). Quality attributes are words for describing "a service or performance characteristic of software" (Wiegers & Beatty, 2013, p. 601).

There are standards for what makes a good requirement, such as being correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable, and traceable. INVEST is an acronym for remembering standards for good user stories: independent, negotiable, valuable, estimable, small, and testable.

An SRS can contain both nonfunctional and functional requirements.

References

Agile Alliance. (n.d.). *What is "given – when – then"?* https://www.agilealliance.org/glossary/gwt/

CEAP. Conservation Effects Assessment Project. (2006). *System requirements specification for STEWARDS.* US Department of Agriculture, Agricultural Research Service. https://www.nrcs.usda.gov/publications/ceap-watershed-2006-stewards-design.pdf

Cohn, M. (2004). *Example user stories*. Mountain Goat Software. https://www.mountaingoatsoftware.com/uploads/documents/example-user-stories.pdf

Eaker, F. (2006, November). *Software requirements specification*. Vyasa. https://vyasa.sourceforge.net/vyasa_software_requirements_specification.pdf

Hanington, B. M., & Martin, B. (2019). *Universal Methods of Design: 125 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers.

Hedberg, T., Helu, M., & Newrock, M. (2017, December). *Software requirements specification to distribute manufacturing data*. NIST Advanced Manufacturing Series 300-2. National Institute of Standards and Technology. https://nvlpubs.nist.gov/nistpubs/ams/NIST.AMS.300-2.pdf

OpenVIBE. (2018, April). *Inria Innovation Lab Certivibe v 1.0 software requirement specification*. http://openvibe.inria.fr/openvibe/wp-content/uploads/2018/04/CERT-Software-Requirement-Specification.pdf

@ShitUserStory. (n.d.). *Shit User Story*. Twitter. https://twitter.com/shituserstory

Spyridonos, P. (2010, February 6). *Software requirements specification for PDF split and merge requirements for version 2.1.0*. University of Kentucky Software Verification and Validation Lab. https://selab.netlab.uky.edu/~ashlee/cs617/project2/PDFSam.pdf

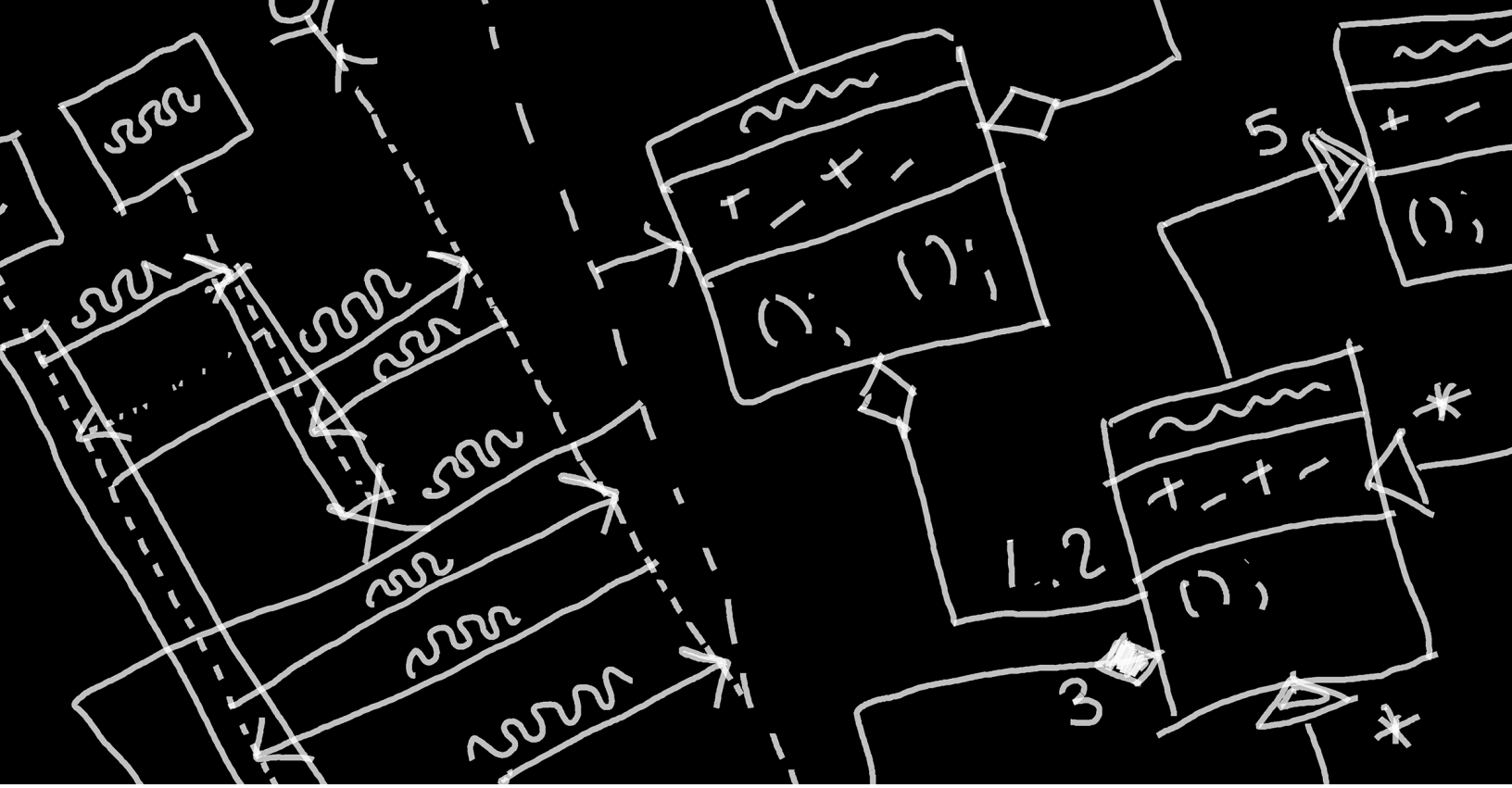
Texas Department of Information Resources. (2008, January 14). *Software requirements specification instructions*. https://dir.texas.gov/sites/default/files/Requirements%20Traceability%20Matrix%20Instructions.pdf

US General Services Administration. (2014, January). *USDA personas and use cases*. https://s3.amazonaws.com/digitalgov/_legacy-img/2014/01/Marsh-Personas.pdf

Wake, B. (2003, August 17). *Invest in good stories, and Smart Tasks*. XP123 Exploring Extreme Programming. https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

Wiegers, K., & Beatty, J. (2013). *Software requirements* (3rd ed.). Developer Best Practices Series. Microsoft Press.

Wikimedia Foundation. (2023, March 23). *List of system quality attributes*. https://en.wikipedia.org/wiki/List_of_system_quality_attributes



Chapter 4

Unified Modeling Language Class and Sequence Diagrams

> "Nobody, not even the creators of the UML, understand or use all of it." (Fowler, 2004)

After a discussion of diagrams in general, this chapter covers two common Unified Modeling Language (UML) diagram types: class diagrams and sequence diagrams.

4.1 How Diagrams Help

Diagrams can help in at least two major ways:

1. They can **help you plan software** you will create. Once you've created diagrams for planning your software, you can use them to communicate to the development team what will/should be implemented and decide (evaluate) whether your plans are any good (e.g., are clear, are logical, reflect your project's desired quality attributes, and so on).
2. They can **help you describe software** you've already created. If your software is already created,

diagrams are good for documentation and, as mentioned above, for evaluating how satisfactory your software is. The purpose of including diagrams in documentation is to communicate something about your software to somebody. There are many different audiences you could be trying to communicate with.

Example audiences for your diagrams include other developers on the project, your supervisor or manager, developers who might be interested in joining the team, developers who want to integrate with your system, curious end users, and students of software engineering. Depending on the integrated development environment (IDE)/tools you're using, diagrams can be automatically generated from your code, which helps make documentation maintenance easier and more likely to happen.

4.2 What Diagrams Must Do Well

To be helpful, diagrams must communicate **clearly** and at an **appropriate level of detail** for your intended audience. If your intended audience does not understand your diagram—or misunderstands it—your diagram has failed.

4.3 What Is UML?

Unified Modeling Language is a family of graphical notations for describing and designing software through diagrams. It is especially applicable to object-oriented software, but some parts of UML are applicable to many types of software. Different UML notations are used for different types of UML diagrams, each of which has a specific purpose. UML was first published in 1994, became a standard of the Object Management Group (OMG) in 1997, and became an ISO standard in 2005. UML is currently on version 2.

4.4 Why Use UML?

There are multiple **benefits** of creating diagrams using UML:

- UML gives you (1) **notation for designing** software so that your implementation will be structured and (2) **notation for describing** the existing design of software so that you can evaluate whether the design is any good.
- UML diagramming **forces you to think** about software design in a structured way. When people try to design software in their minds, they can be sloppy about it—thinking about the aspects of

the design they want to think about. UML can encourage you to face the trickier parts of software design.

- UML diagramming gives you a view of the software at **different levels of design** (e.g., class level, component level, package level).
- UML provides a **common language** between software professionals. Because UML is well known, it gives developers and managers a way to communicate in detail about software. That being said, expect to encounter variations in how UML notation is used—it can be difficult to remember UML notation; developers will make mistakes or adapt the notation to their own way of thinking. It can help to provide a legend or explanation of what your notation means.
- UML diagrams give you a way to tell people about your software's structure **without asking them to look through code**. This is nice, for example, when onboarding new developers or communicating with managers.

4.5 Why NOT Use UML?

> Some IDEs will automatically generate some types of UML diagrams from your code. This is nice because it's easy to regenerate your diagram when your code changes. The generated diagrams can sometimes have more detail than you want, however, making them less good for communicating.

There are some **drawbacks** to UML diagramming:

- People tend to **vary their UML notation**, which can cause **confusion**. Some tips for avoiding that problem include (1) keeping your notation basic and (2) explaining more complex notation.
- Getting UML notation right **can take a lot of time**. Remember that diagrams are for communicating. If creating the diagram takes longer than explaining the code a different way, the diagram isn't helping.
- UML diagrams **can require a lot of maintenance**. If your software design changes frequently, so must your UML diagrams if you want them to be accurate. Fortunately, some IDEs can generate some UML diagrams from your code.

4.6 Class Diagrams

A class diagram describes a system's classes and the static relationships that exist among them. Class diagrams also show properties and operations of a class. Properties represent the structure of a class (e.g., instance variables) and operations represent the functionality provided by the class (e.g., methods; Fowler, 2004).

Figure 4.1 shows an example class diagram. In the diagram, there are relationships between three classes: Customer, Order, and SharedOrder. An Order has one Customer—but the same Customer can be on multiple Orders. A SharedOrder is a type of Order that can have multiple Customers. The classes have attributes (e.g., id) and operations (e.g., getId()).

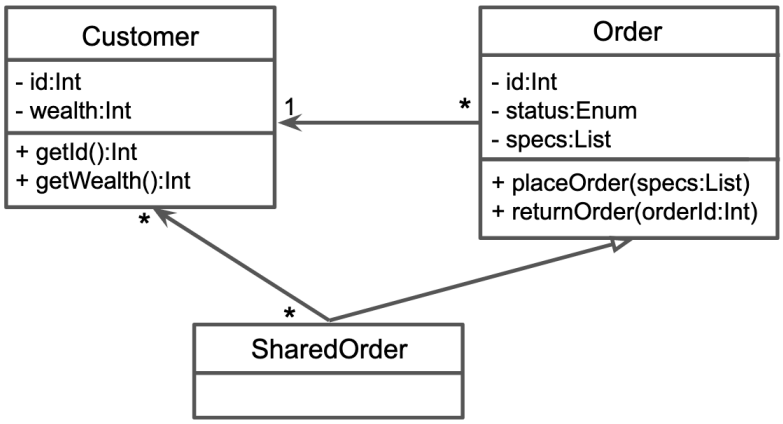


Figure 4.1 *Simple Class Diagram*

The next section explains each of the notational elements shown in the example.

4.6.1 UML Class Diagram Notation

This section contains some of the most common UML class diagram notation.

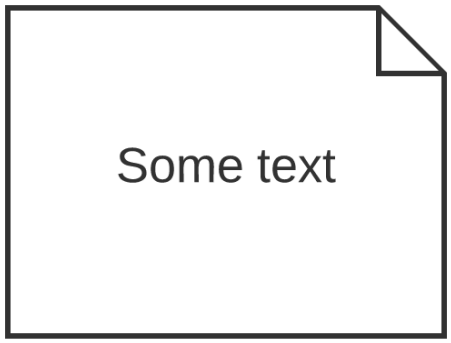


Figure 4.2 *Note*

Note. Notes are for placing comments on class diagrams.

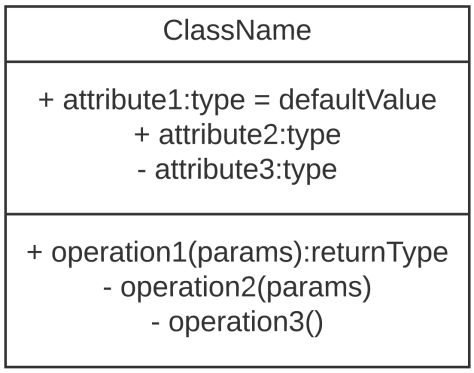


Figure 4.3 *Class*

Note. Attributes are properties that are listed within a class box and the operations are methods. The + indicates a public method, – is private, and # is protected. The notation includes attribute types (e.g., int, Token, etc.), method parameters and return types, and default values for attributes.

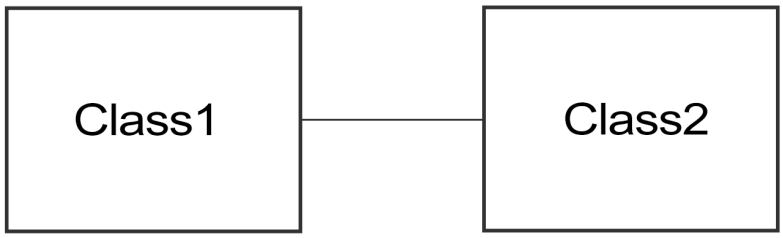


Figure 4.4 *Association*

Note. Association means that a class contains a reference to an object(s) of the other class in the form of a property. In this example, we aren't told whether Class1 references Class2 or vice versa.

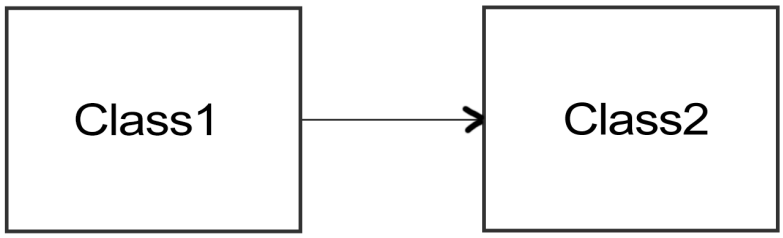


Figure 4.5 *Unidirectional Association*

Note. The arrow indicates that Class1 **has a** Class2.

Figure 4.6 *Unidirectional Association with Property Name*

Note. Class1 has a property named instance1. It is an instance of Class2.

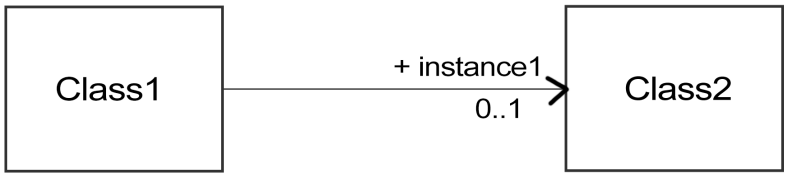


Figure 4.7 *Unidirectional Association with Property Name and Target Multiplicity*

Note. Class1 has a property named instance1 containing zero or one instances of Class2.

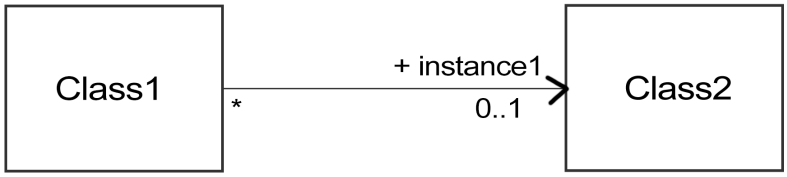


Figure 4.8 *Unidirectional Association with Property Name, Target Multiplicity, and Source Multiplicity*

Note. Zero or more instances of Class1 have properties named instance1 containing zero or one instance of Class2.

Figure 4.9 *Bidirectional Association and Target/Source Multiplicity*

Note. Class1 has zero or more instances of Class2. Class2 has exactly one instance of Class1.

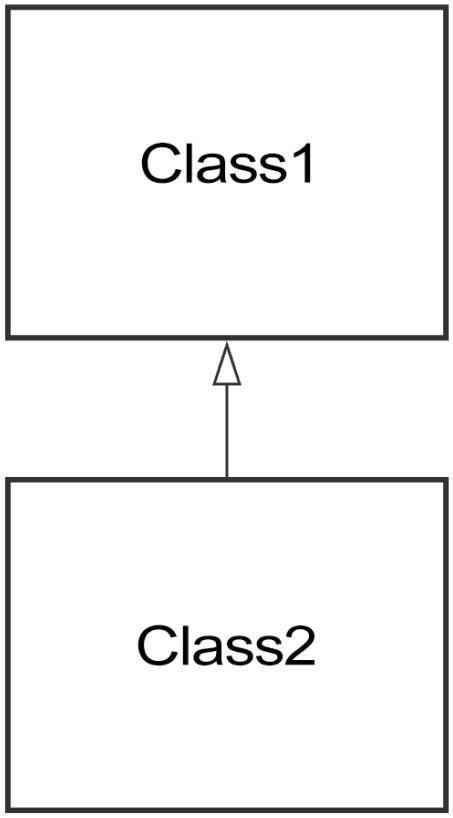


Figure 4.10 *Inheritance*

Note. Class2 is a subclass of Class1; Class2 **is a** Class1.

4.6.2 Real-World Class Diagram Examples

These PDFs contain class diagrams for actual software.

- *EvoloPy: An open-source nature-inspired optimization framework in Python* (Faris et al., 2016).
- *Processing: A Python framework for the seamless integration of geoprocessing tools in QGIS* (Graser & Olaya, 2015).
- *Java-based graphical user interface for MRUI, a software package for quantitation of in vivo/medical magnetic resonance spectroscopy signals* (Naressi et al., 2001).
- *RepoMiner: A language-agnostic Python framework to mine software repositories for defect prediction* [PDF] (Palma et al., 2021).
- *pyfao56: FAO-56 evapotranspiration in Python* [PDF] (Thorp, 2022).

If any links are broken, try the Wayback Machine.

4.7 Sequence Diagrams

> When making any diagram, know your audience and what you're trying to communicate. If your audience is a human, they have limited capacity for absorbing tiny details (and probably limited time). Focus on showing them what's most important in a way they will understand.

A sequence diagram describes interactions between objects. Usually, the diagram shows a single use case or scenario. Sequence diagrams are a type of interaction diagram and are not as good for showing object implementation details.

This section provides an example sequence diagram and commonly used sequence diagram notation. For more detailed information, see Fowler (2004).

In Figure 4.10, the example sequence diagram represents interactions between instances of the Manager, Employee, and Order classes. Manager asks the Employee for a status update, Employee complies, Employee creates an Order, Manager asks Employee to close the shop, Employee closes the Order.

In the example, the columns (called participants) are objects, but this is not always the case. For example, a participant can be a user. Users, if they are human, are sometimes represented as stick figures (without the box). Another possible non-object participant could be a database (although in some cases, a database is considered an object). What's most important when creating diagrams is not following the rules or conventions but communicating with your audience.

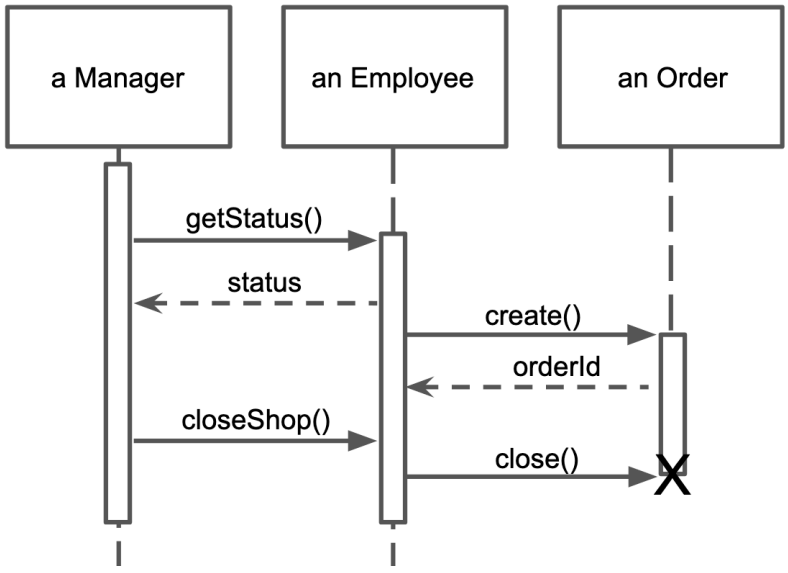


Figure 4.11 *Simple Sequence Diagram*

4.7.1 UML Sequence Diagram Notation

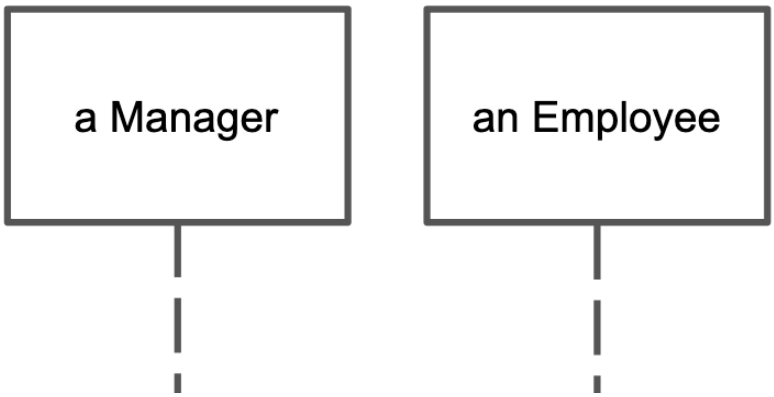


Figure 4.12 *Participant*

Note. The "columns" of a sequence diagram are each participants. Participants are often objects. The name of the participant goes in the box.

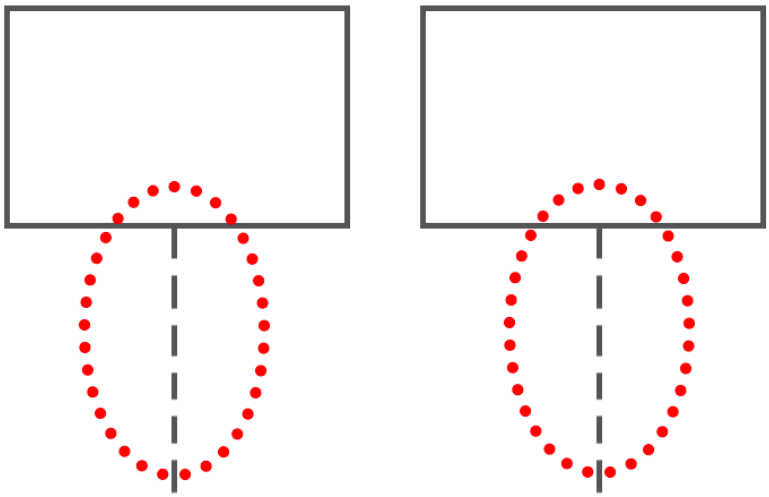


Figure 4.13 *Lifeline*

Note. Vertical dashed line represents the life span of the participant. Top is beginning of life, and bottom is the end. Life ends when the participant is deleted.

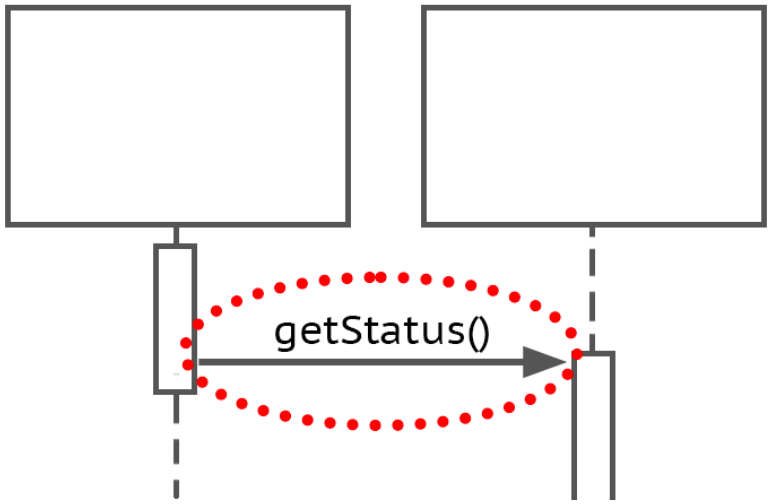


Figure 4.14 *Message*

Note. Interaction from one participant to another is shown by the solid line with arrow. Often a method call.

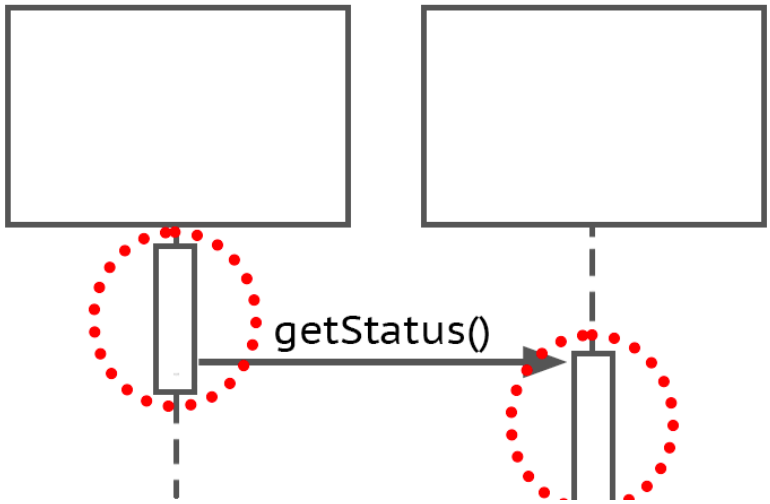


Figure 4.15 *Activation Bar*

Note. Box on lifeline indicates when the participant is active. Indicates method is on call stack.

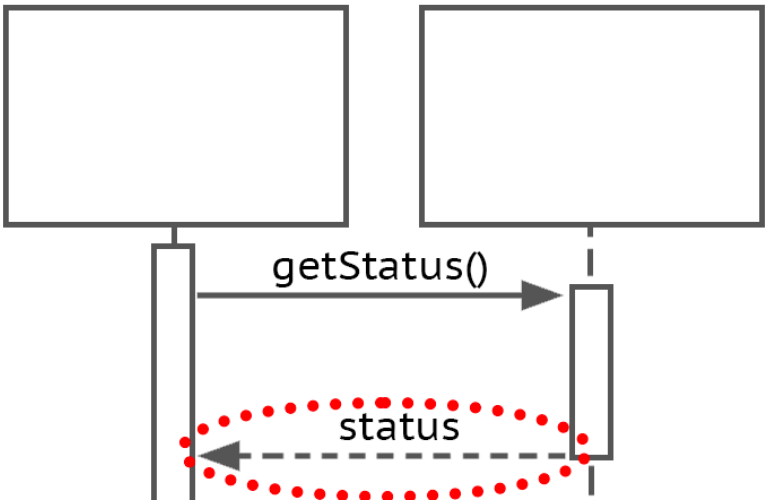


Figure 4.16 *Return*

Note. Dashed line with arrow indicates method return. Use only when it helps communicate something important about the interaction.

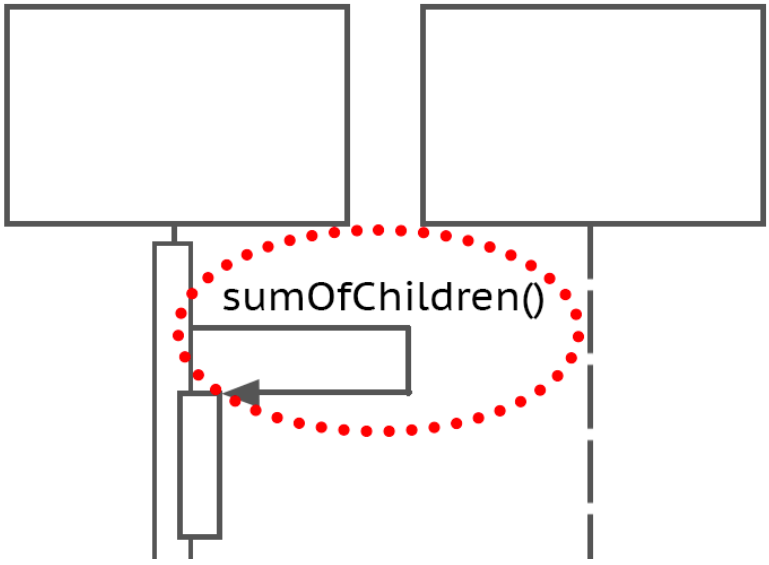


Figure 4.17 *Self-Call*

Note. Method calling self. Solid line with arrow points back to participant's own lifeline.

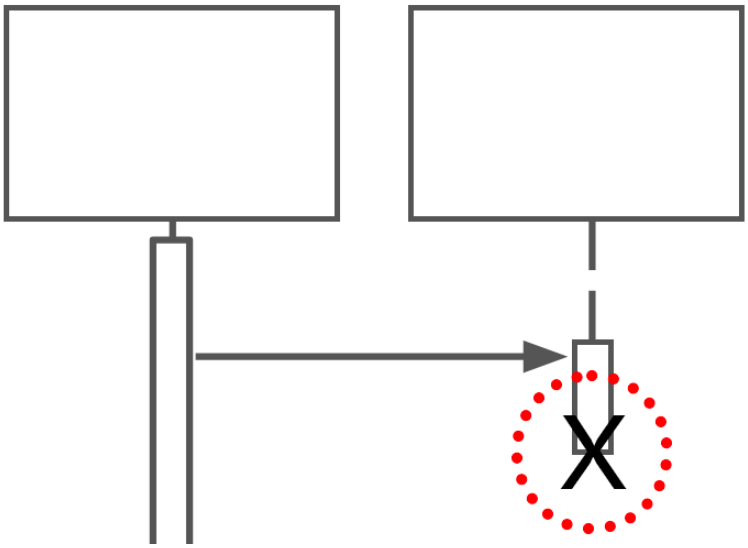


Figure 4.18 *Deletion*

Note. End of participant's life. Indicated by an X on the lifeline.

4.7.2 Real-World Sequence Diagram Examples

These PDFs contain sequence diagrams for actual software.

- *Py4JFML: A Python wrapper for using the IEEE Std 1855-2016 through JFML* [PDF] (Alcalá-Fdez et al., 2019).
- *COFFEE—An MPI-parallelized Python package for the numerical evolution of differential equations* (Doulis et al., 2019).
- *Teetool—A probabilistic trajectory analysis tool* (Eerland et al., 2017).
- *GEMS: A Python library for automation of multidisciplinary design optimization process generation* [PDF] (Gallard et al., 2018).

If any links are broken, try the Wayback Machine.

4.8 Summary

UML diagrams can be helpful for communicating how your code works. Class diagrams and sequence diagrams are two commonly used types of UML diagrams. Each type of diagram emphasizes some part of the code design while leaving out other parts. UML diagrams are for communicating with humans—not computers.

References

Alcalá-Fdez, J., Alonso, J. M., Castiello, C., Mencar, C., & Soto-Hidalgo, J. M. (2019, June). *Py4JFML: A Python wrapper for using the IEEE Std 1855-2016 through JFML*. Paper presented at the 2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), New Orleans, LA, USA. https://ricerca.uniba.it/bitstream/11586/256332/9/PID5822281-PrePrint%28con-DOI%29.pdf

Doulis, G., Frauendiener, J., Stevens, C., & Whale, B. (2019). COFFEE—An MPI-parallelized Python package for the numerical evolution of differential equations. *SoftwareX*, 10, 100283. https://www.sciencedirect.com/science/article/pii/S2352711019300950

Eerland, W., Box, S., Fangohr, H., & Sóbester, A. (2017). Teetool—A probabilistic trajectory analysis tool. *Journal of Open Research Software*, *5*(1). https://openresearchsoftware.metajnl.com/articles/10.5334/jors.163

Faris, H., Aljarah, I., Mirjalili, S., Castillo, P. A., & Guervós, J. J. M. (2016). EvoloPy: An open-source nature-inspired optimization framework in python. In *Proceedings of the 8th International Joint Conference on Computational Intelligence (IJCCI): Evolutional Computational Theory and Applications (ECTA), 1*, 171-177. https://research-repository.griffith.edu.au/bitstream/handle/10072/401215/Estivill-Castro165057-Published.pdf?sequence=2

Fowler, M. (2004). *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley Professional.

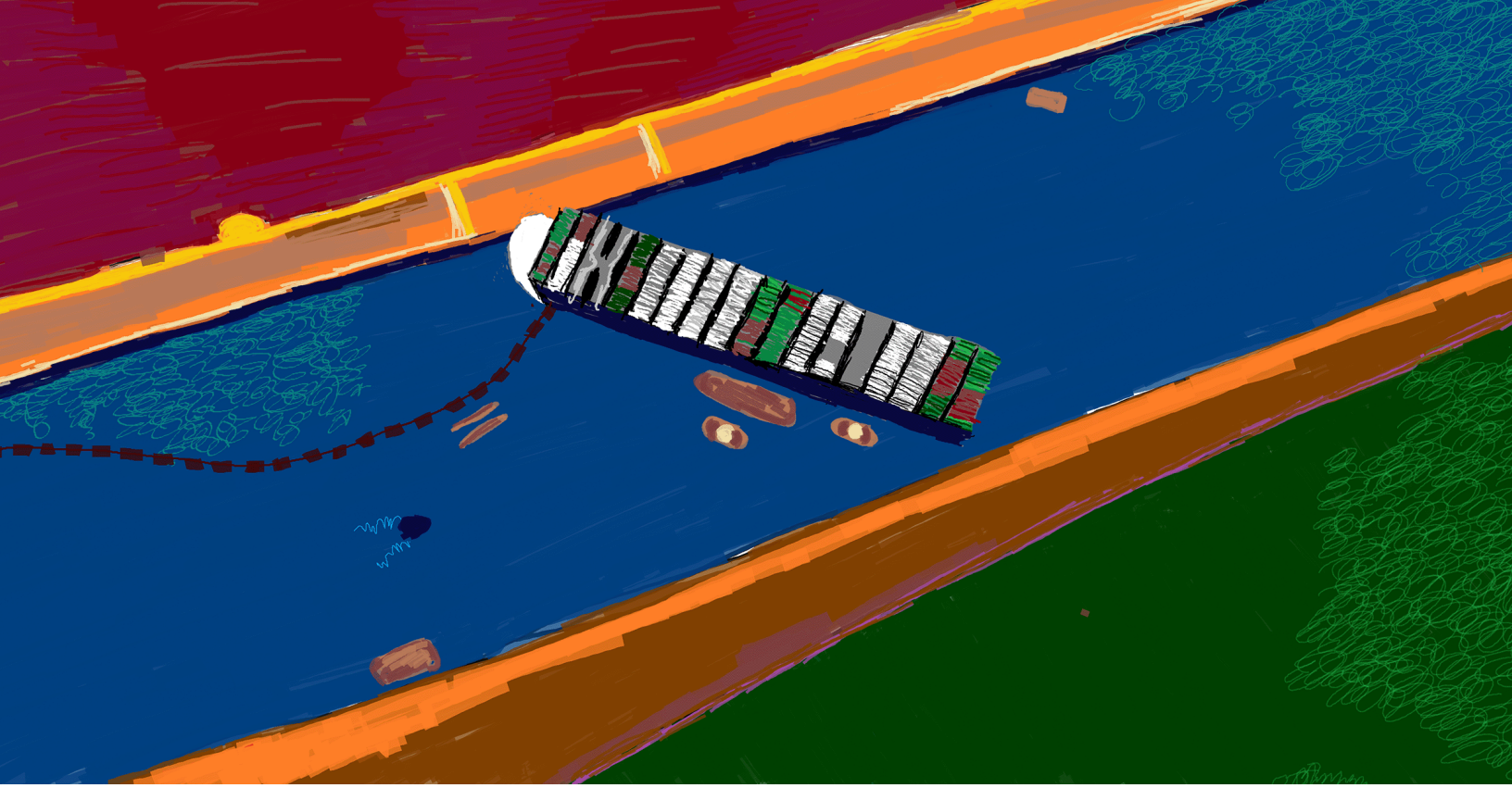
Gallard, F., Vanaret, C., Guénot, D., Gachelin, V., Lafage, R., Pauwels, B., Barjhoux, P.-J., & Gazaix, A. (2018). *GEMS: A Python library for automation of multidisciplinary design optimization process generation*. Paper presented at the 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Kissimmee, FL, USA.. https://hal.science/hal-02335530/file/DTIS19188.1570026732_preprint.pdf

Graser, A., & Olaya, V. (2015). Processing: A python framework for the seamless integration of geoprocessing tools in QGIS. *ISPRS International Journal of Geo-Information*, *4*(4), 2219-2245. https://www.mdpi.com/2220-9964/4/4/2219/pdf

Naressi, A., Couturier, C., Castang, I., De Beer, R., & Graveron-Demilly, D. (2001). Java-based graphical user interface for MRUI, a software package for quantitation of in vivo/medical magnetic resonance spectroscopy signals. *Computers in Biology and Medicine*, *31*(4), 269-286. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=cdb5e5d28a9bd6a04f969d6465110f875e706e71

Palma, S. D., Di Nucci, D., & Tamburri, D. (2021). RepoMiner: A language-agnostic Python framework to mine software repositories for defect prediction. *arXiv preprint arXiv:2111.11807*. https://arxiv.org/pdf/2111.11807.pdf

Thorp, K. R. (2022). pyfao56: FAO-56 evapotranspiration in Python. *SoftwareX*, *19*, 101208. https://www.ars.usda.gov/ARSUserFiles/40820/Thorp2022%20-%20pyfao56.pdf



Chapter 5

Monolith versus Microservice Architectures

High-level architecture is the software's all-encompassing code design. When described with a diagram, a high-level architecture usually looks like a few to dozens of interconnected shapes with short labels, an abstraction that usually represents the entire codebase. In this chapter, we'll use "architecture" interchangeably with "high-level architecture" (in other contexts, software architecture can refer to code design at lower levels).

In this chapter, **I won't be covering every high-level architecture**. Instead, I'll concentrate on two distinct high-level architectures: monolith and microservices. Talking about the ways they're different will lead us through concepts applicable to high-level architecture in general.

5.1 Monolith Architecture

Monolith software is one interconnected codebase that **cannot easily be divided** into multiple independent components that run separately and are individually useful.

If you're trying to think of an example of a monolith and nothing is coming to mind, that's probably because this architecture is so common that it can arise without having to plan. Your first computer program was probably a small monolith. If you keep adding more code/files/classes/components, the software becomes a bigger monolith—unless you change the architecture.

5.2 Microservice Architecture

Microservices are **separate applications**, each of which runs in a **separate process** and could be **individually useful**. This section describes core characteristics of software that uses the microservice architecture. The subheadings are borrowed from Lewis & Fowler (2014). Martin Fowler's Microservices Guide (Fowler, 2019) provides additional discussion.

5.2.1 "Smart End Points and Dumb Pipes"

> "Dumb pipes" does not imply simple message contents.

The communication pipe within a microservice architecture is **simple,** and the services themselves take care of translating and otherwise processing messages. For example, microservices commonly communicate through a REST API, which allows these kinds of messages: GET, POST (create), PUT (update), or DELETE. The contents of the messages can be complex, but it's the job of the services to deal with that.

5.2.2 "Componentization via Services"

> Even though it provides a service, a library is not a service if you're including its code in your code.

In a microservice architecture, **components are services**. The Lewis and Fowler (2014) definition of a component is "a unit of software that is independently replaceable and upgradeable." A service provides functionality while running in its own process. A monolith typically has code with tight coupling and components that run in the same process.

Advantages of splitting components into services:

- **Independence**: Each individual service can be updated, tested, launched, and stopped without requiring the same from other components of the software. In contrast, with some monolithic software, all tests must be run each time a developer commits to a change, which can make for a long wait. If a service fails, any software depending on it will be without that service, but the rest of the software needn't be affected.
- **Standardized component communication**: Service communication pipes can be simple and the same each time. This can make for less thinking, fewer mistakes, and less violation of encapsulation when connecting two components—just use the pipe.

Disadvantages of splitting components into services:

- **More expensive communication**: Components in a monolith can communicate via direct calls (fast, lightweight); in contrast, microservices **often communicate over a network**. Microservice requests typically need to include request metadata, and because the pipes are "dumb," responses might contain extra data (slower, heavier).
- **Potentially less secure communication**: Communication over a network can be more prone to interception and alteration.

5.2.3 "Organized around Business Capabilities"

You may have heard of the client-server architecture, in which multiple instances of client-side software communicate with server-side software, which communicates with a database. That architecture is organized around technology. Another way to put that: someone unfamiliar with the differences between client-side software, server-side software, and a database would not get much out of seeing a diagram of this architecture.

In contrast, microservices are organized around business capabilities. This term has multiple definitions. Michell's (2011) integrated definition of a business capability fits what we're talking about: "the potential of a business resource (or groups of resources) to produce customer value by acting on their environment via a process using other tangible and intangible resources."

Examples of business capabilities:

- The manufacturer can slice a 20-foot by 40-foot rectangle of wheat dough into 0.5-cm strips in 1.2 seconds, which will later become packaged noodles someone can buy for lunch in a grocery store.
- A loan officer can lead a customer through the process of securing a loan, enabling the customer to start a small business.
- A pet food distributor can regularly ship nutritionally balanced cat food to stores around the country.
- The software can make a video file compatible with mobile devices.

One implication of being focused on business capabilities is that each microservice can have its own tech stack (including its own database).

5.2.4 "Decentralized Data Management"

In a microservice architecture, **each service typically has its own database** instead of sharing a centralized database. This is part of decoupling the software's components, which has many benefits including **failure containment**. A disadvantage is that if two microservices need to share data, the two copies of that data can become inconsistent (e.g., because one database has not yet received the update). Microservice databases are said to have eventual consistency, which means that, with time, each microservice will have the most up-to-date information, but meanwhile, there could be a mismatch (perhaps one that will annoy or mislead human users).

5.2.5 "Decentralized Governance"

Microservices need only be compatible at their interfaces (communication pipe), leaving **flexibility in how each is implemented**. For example, each service can be written in a different language, reducing the weight of tech stack decisions and decreasing the need to compromise on those decisions. For each service, teams can choose the optimal programming language, framework, architecture, and more. The technologies of each microservice can be independently changed. Conversely, in a monolith, teams might only need to maintain a small set of technologies (e.g., if there's only one framework, only one framework will need updates installed) and might not need as broad of expertise (e.g., having working knowledge of five programming languages). Also, when code is more or less part of the same codebase, it might be easier to maintain the same standards across the code.

5.2.6 "Design for Failure"

When services run in different processes on different machines and were created by different teams using different technologies and standards, that **can change how developers think**. Instead of keeping the whole ship afloat, thinking can shift toward service-specific **monitoring, logging**, and design decisions about **what to do when a service fails**—including what to tell the user. In contrast, with a monolith, more thought might be put into how to revert quickly if a deployment fails (because failure might mean no part of the monolith works). Monoliths can also be designed for failure, but that's not as natural a tendency as with microservices.

5.3 Monolith Compared to Microservices

This section recaps and expands upon differences between monolith and microservice architectures (Fowler, 2015; Lewis & Fowler, 2014).

5.3.1 How Does Communication Happen within a Monolith versus between Microservices?

In a monolith, communication (e.g., between classes and components) can happen in many ways, including through direct calls and over a network. With microservices, communication typically happens over a network such as through HTTP requests/responses, through "dumb," standardized communication pipes. While microservices communication pipes are less complex, that means the end points need to be smarter. Also, communication over a network can be less reliable and less secure.

5.3.2 How Is a Monolith Deployed versus Microservices?

Monolithic software often needs to be deployed all at once. Microservices can be independently deployed and can potentially be stopped without stopping connected services.

5.3.3 How Is a Monolith Scaled versus Microservices?

If your monolithic software needs more resources to be able to support how much it's being used, it can be copied onto multiple machines. Each machine must have enough space, memory, processing speed, and the like to support the entire monolith.

If your microservices software needs more resources, you have more options. For example, the services that are used more can be replicated more times.

5.3.4 How Is a Monolith Tested versus Microservices?

In microservice software, each service can be independently tested. In a monolith, the way you test is influenced by dependencies within the code, which could reach broadly across the software (and make for slow tests).

5.3.5 How Is a Monolith Upgraded versus Microservices?

Each microservice can be written in a different language (e.g., one in Python, another in Java, another in C++, etc.) and can run in different contexts (e.g., machines with different operating systems, libraries, versions of libraries, and so on). In theory, this means they can be independently upgraded.

With a monolith, upgrading may require more care. Each component must be compatible with the new context (but this is also sometimes true with microservices).

5.3.6 How Is the Database Used in a Monolith versus Microservices?

Monolithic software might have just one database, potentially a very large one. This can create a bottleneck if multiple parts of the software need to access the database in parallel and can make for slow database backups/restores, among other drawbacks. If you only have one database, however, that's just one place for managing database access accounts and one database to maintain/back up/restore/etcetera. In contrast, each microservice typically has its own data storage.

5.4 Summary

Monolith and microservice architectures have different advantages and disadvantages. In a microservice architecture, each service is its own application and can be independently managed. Communication mechanisms between modules can be standardized. In a monolith, however, the codebase can be deployed all at once and components can communicate directly, which can be more reliable, less expensive, and provide better consistency than communicating between multiple applications over a network.

5.5 Case Study: Microservice Architecture

The Oregon State University (OSU) Center for Applied Systems and Software (CASS) is a nonprofit that gives students real-world software development experience through its work with clients such as the Oregon Department of Transportation (ODOT).

CASS and ODOT decided to convert ODOT's statewide computer-aided dispatch software, Transportation Operation Center System (TOCS), from a monolith to microservices. TOCS helps dispatchers share road emergency information with responders and the public. The part of TOCS that CASS started with was the outdated home screen.

From a user perspective, the main problem with the TOCS home screen was inflexibility. Dispatcher centers in different parts of Oregon had different needs (e.g., some centers dealt with more icy roads, others withs more fender-benders) but had to use the same home screen, which could not be easily configured.

From a developer perspective, the monolith had multiple technological drawbacks that made it difficult to respond to TOCS users' needs:

- It was **difficult to keep software components decoupled**, especially since many different developers worked on the software. They were building up technical debt, which meant that developers might need to focus on clearing that debt instead of implementing new TOCS features.
- CASS **could only deploy TOCS a few times a year** because the software had to be tested and deployed in its entirety (a long process) and it was essential for the software to remain stable, especially during times of year with more weather and road hazards. This meant dispatch centers had to wait a long time for new features (e.g., individualized home screens).
- There was a lot of **pressure on the database** because the TOCS software at all the dispatch centers was transacting with the same database and causing performance issues.
- **Technology choices were limited** because every part of the software had to be compatible with the .NET Framework. Even worse, their **technology stack was becoming deprecated** because Microsoft stopped releasing updates to the .NET Framework after version 4.8. CASS chose the microservice architecture as a solution to all these problems.

Figure 5.1 depicts the new architecture of the TOCS homepage, which integrates with the monolith. The WinGui Gateway application is responsible for preparing data from the services so it can be used by the New Home Screen UI. It uses the .NET 6 stack, which gives developers access to modern features. The Message Broker (Apache ActiveMQ) application talks to the services and the Gateway. Because the

Message Broker uses a standard protocol, AMQP, it would be feasible to change the Message Broker technology in the future. Each service is also a separate application and has its own database. CASS found that one advantage of a dedicated database was that they could use JSON for the Profile Service, which was more appropriate than the relational database used within the monolith.

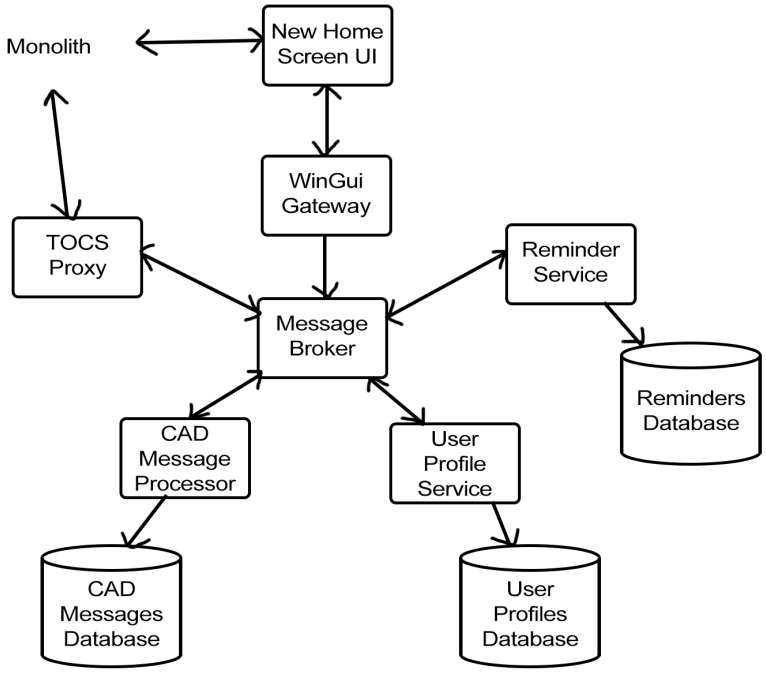


Figure 5.1 *Microservice Architecture of ODOT's TOCS Home Screen*

For more information about this project, see Fern (2022) for a video that describes it in detail.

References

Fern, A. (2022). *Tech Talk Tuesday: Lessons in real-world software: going from monolith to microservices*. OSU MediaSpace. https://media.oregonstate.edu/media/t/1_ls3xsa6r

Fowler, M. (2015, July 1). *Microservice trade-offs*. martinfowler.com. https://martinfowler.com/articles/microservice-trade-offs.html

Fowler, M. (2019, August 21). *Microservices guide*. martinfowler.com. https://martinfowler.com/microservices/

Lewis, J., & Fowler, M. (2014, March 25). *Microservices*. martinfowler.com. https://martinfowler.com/articles/microservices.html

Michell, V. (2011). A focused approach to business capability. In B. Shishkov (Ed.), *Proceedings of the First International Symposium on Business Modeling and Software Design*, 105–113. Springer. https://doi.org/10.5220/0004459101050113

Chapter 6

Paper Prototyping

User interface (UI) design often involves prototyping: iteratively creating depictions of what you think the UI should look like, and how users should interact with it, based on the software's requirements. Prototyping gives you a way to try out a UI design and find problems early. Changing a drawing (digital or physical) is often easier and faster than changing its code implementation.

There are multiple levels—or "fidelities"—of UI design prototypes (low fidelity, medium fidelity, and high fidelity). If you look around, you'll find disagreement on the definitions (Snyder, 2011). **I use the following definitions**:

- Low fidelity (Figure 6.1): A **rough sketch** that is often drawn by hand, drawn using an app and stylus, or made using software specifically for creating low-fidelity prototypes. At this fidelity, you can **gather feedback on higher-level features** and have the **flexibility** to make large, low-cost changes.

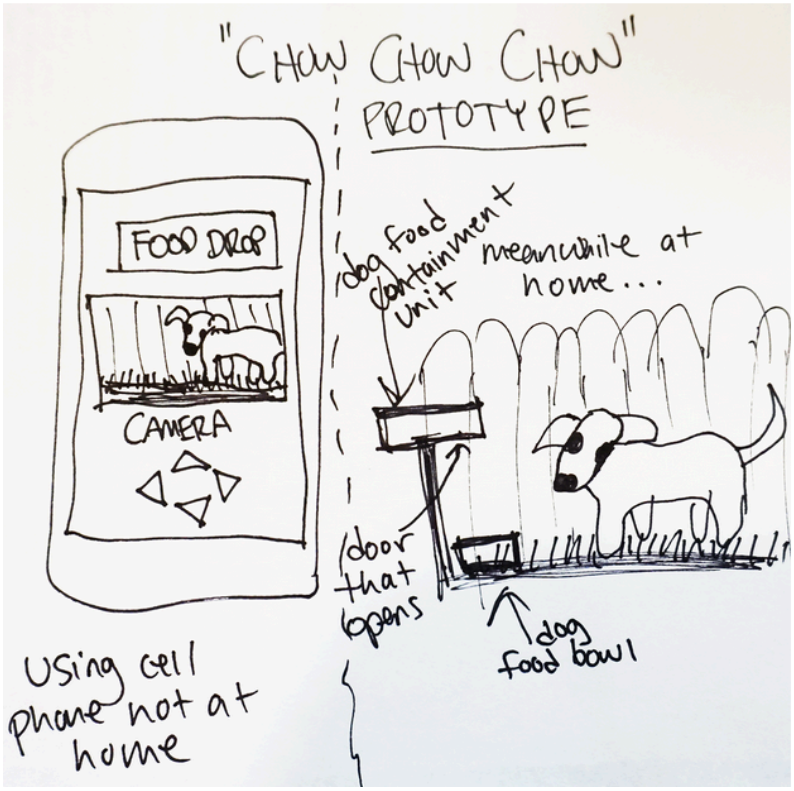


Figure 6.1 *Low-Fidelity Prototype Example*

- Medium fidelity (Figure 6.2): A **detailed** illustration often created using a professional drawing or presentation tool (e.g., Visio, PowerPoint, and the like), or perhaps a careful and detailed hand drawing. At this fidelity, to keep costs low, you can **gather feedback on small changes** to defined and accepted features that you plan to keep but might change the look of.

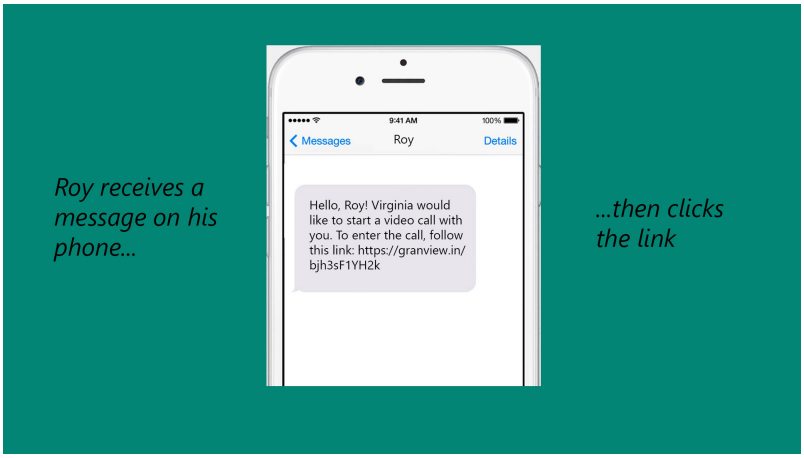


Figure 6.2 *Medium-Fidelity Prototype Example*

- High fidelity (Figure 6.3): A **polished**, detailed illustration that looks like a finished UI. These designs might be created in a full-featured graphics editor (such as Photoshop, Illustrator, etc.) or a GUI builder. At this fidelity, to keep costs low, you can **gather feedback about detailed tweaks** to specific features to make focused and incremental improvements.

Figure 6.3 *High-Fidelity Prototype Example*

A quick and low-cost way to begin prototyping (and begin getting feedback on your UI design) is to create a low-fidelity paper prototype.

A paper prototype is a hand-drawn sketch of a UI design that's based on the software's requirements. It **doesn't need to be pretty** or artistic. It can be simple and reduce the UI to only the most important elements (i.e., it is often low fidelity).

6.1 Showing Interaction

A paper prototype **needn't be static** or limited to one sheet of paper. With some craftiness and creativity, paper prototypes can communicate elements of interaction design by indicating what users can interact with (e.g., a slider), how they can interact (e.g., by dragging), and what happens when they interact (e.g., an overlay appears, showing the elevations of each mountain in the photo). To show interaction design

through a paper prototype, you can, for example, cut out small paper shapes you can easily move around (e.g., a small rectangle showing the submenu items that appear when a user clicks), place arrows and annotations on your prototype, and even add strings to show how UI elements may move. I've even seen people use brass brads for spinnable elements. But keep in mind that if your client doesn't like your design, you might have saved time and communicated your concept just as well with a less elaborate paper prototype.

6.2 Showing Your Concept to Others

Once you have a paper prototype, you can **use it to harvest feedback**. Here's one way: if each of your screen designs is on one piece of paper, give your user the entry screen drawing, then either give them a task (e.g., submit data report) or let them explore on their own. Watch as they tap buttons or otherwise interact. Be ready to quickly swap in other drawings to respond to their interactions (e.g., if they tap the gear icon, give them a sketch of the settings screen). If you're fast and brought extra supplies, you can construct new designs on the fly or (if they're interested) let your user participate.

You can ask your user to provide feedback about the design after they're done using it or as they go, using a think-aloud protocol. Ask your user to tell you what they're doing, what they're trying to do, what questions they have at that moment, what they don't like, and so on.

6.3 Summary

Paper prototyping can help reduce project costs by giving a way to detect user interface design flaws before they are implemented. It can also help teams communicate about the software with each other, clients, and users.

Reference

Snyder, C. (2011). *Paper prototyping: The fast and easy way to design and refine user interfaces*. Morgan Kaufmann.

Chapter 7

Inclusivity Heuristics

The Inclusivity Heuristics are **guidelines for designing technology to work well for a diversity of users**. Using the heuristics to build inclusive technology is a way to practice inclusive design: it is "a methodology . . . that enables and draws on the full range of human diversity. Most importantly, this means including and learning from people with a range of perspectives" (Microsoft).

The Inclusivity Heuristics, in their current form, give advice for how to support five cognitive facets involved in how people interact with technology for the first time (Burnett et al., 2016).

> The full definitions of these facets are available in GenderMag Project et al. (2021).

1. Attitude toward risk (risk-averse to risk-tolerant).
2. Computer self-efficacy (low to high).
3. Information processing style (comprehensive to selective).
4. Learning style (process-oriented to mindful tinkering to tinkering).
5. Motivations (task-motivated to motivated by tech interest).

A cognitive style is a cognitive facet value. For example, my cognitive styles are medium attitude toward risk, high computer self-efficacy, selective information processing style, a highly variable learning style, and task motivation.

The Inclusivity Heuristics help software practitioners support the full range of cognitive styles for each cognitive facet.

7.1 Background

The Inclusivity Heuristics, also called the Cognitive Style Heuristics or the GenderMag Heuristics (Burnett et al., 2021), were developed by human-computer interaction researchers at Oregon State University as part of the GenderMag Project. The research behind the heuristics is more than 40 publications about gender differences in how people use technology. In the future, the heuristics will potentially expand to include research about other diversity dimensions, such socioeconomic diversity (Hu et al., 2021) and age diversity (McIntosh et al., 2021).

Heuristics, such as the Inclusivity Heuristics and Nielsen's Heuristics (Nielsen, 1994), are meant to be used within a usability inspection method called heuristic evaluation (Nielsen & Molich, 1990). In a heuristic evaluation, multiple evaluators independently check whether a technology design follows the heuristics. They make note of any issues and compare results. The output is a combined set of usability issues.

7.2 Inclusivity Heuristics Personas

> You can find the full versions of the Abi, Pat, and Tim personas and the GenderMag Method at GenderMag.org.

A unique characteristic of the Inclusivity Heuristics is they are framed from the perspective of supporting three personas: Abi, Pat, and Tim. A persona is a representation of a user or a group of users. Abi, Pat, and Tim each have different cognitive styles. Figure 7.1 lists each persona's cognitive styles.

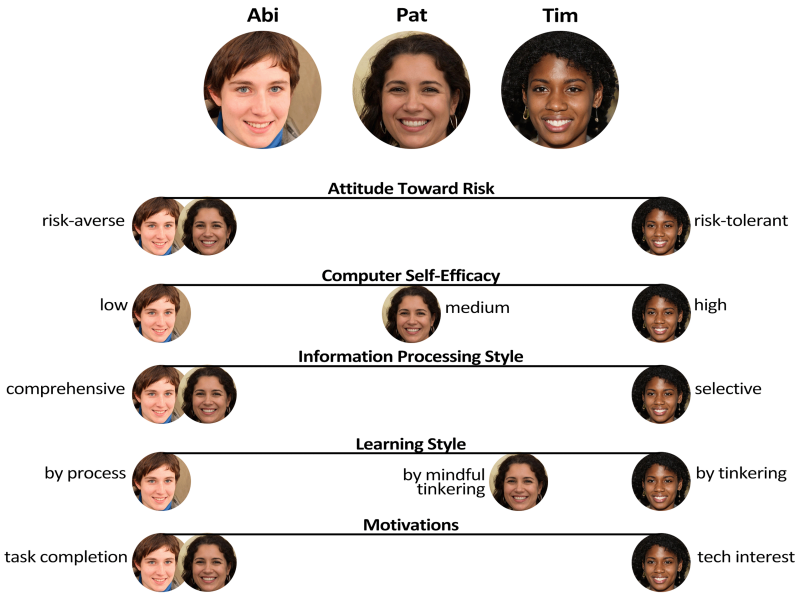


Figure 7.1 *Cognitive Styles of Abi, Pat, and Tim*

Note. The personas can have any gender and picture.

7.3 The Inclusivity Heuristics

Each of the eight heuristics are listed and described below, with examples.

7.3.1 Heuristic #1 (of 8) Explain (to Users) the Benefits of Using New and Existing Features

Abi and Pat have a pragmatic approach toward technology, using it only when necessary for their specific tasks. They have limited spare time and prefer to stick to familiar features, enabling them to maintain focus on the task at hand. Unless they can clearly understand how certain features will help them complete their tasks, they might not use them.

Abi is risk-averse toward technology. Abi tends to avoid features with unknown time costs and other risks.

Similarly, Pat is also cautious about using new features, but open to trying out features to determine whether they're relevant to completing their task.

In contrast, Tim is enthusiastic about discovering and exploring new, cutting-edge features. Moreover, Tim is willing to take risks and may use features without prior knowledge of their costs or even their exact functionality.

Figure 7.2 provides an example design that reflects this heuristic and how Abi, Pat, and Tim might react to it.

To support the personas' **motivations** and **risk tolerance**, make it easy for Abi and Pat to quickly see the benefits of features and decide if they want to use them, and give Tim the ability to quickly figure out what new and unique features do, so these users can explore those features if they're interested.

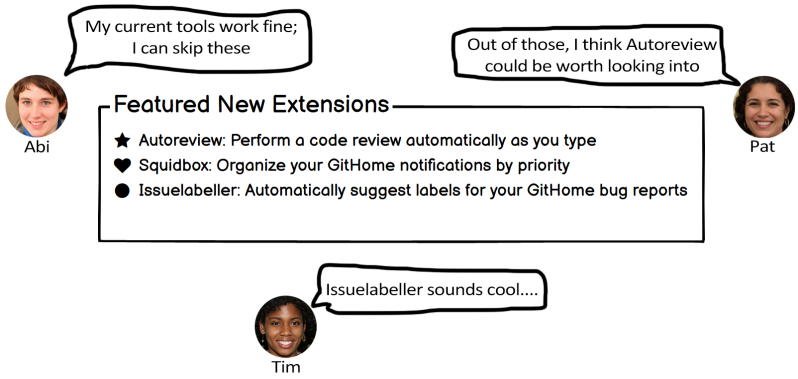


Figure 7.2 *Inclusivity Heuristic #1 Design Example*

Note. The designs help Abi, Pat, and Tim decide whether they want to use the features. Abi and Pat seek features that help them with their task. Tim seeks features that are interesting.

7.3.2 Heuristic #2 (of 8)
Explain (to Users) the Costs of Using New and Existing Features

Abi and Pat prefer to reduce risk by avoiding features that might require significant time and effort.

Tim is more open to taking risks and may be willing to invest additional time and effort into using features, even if they aren't directly related to the current task.

Figure 7.3 provides an example design that reflects this heuristic and how Abi, Pat, and Tim might react to it.

To support the personas' **attitudes toward risk**, give them the ability to assess whether a feature might require excessive time and effort so that Abi and Pat can avoid it or proceed with caution, and so Tim understands the relative amount of risk a feature comes with.

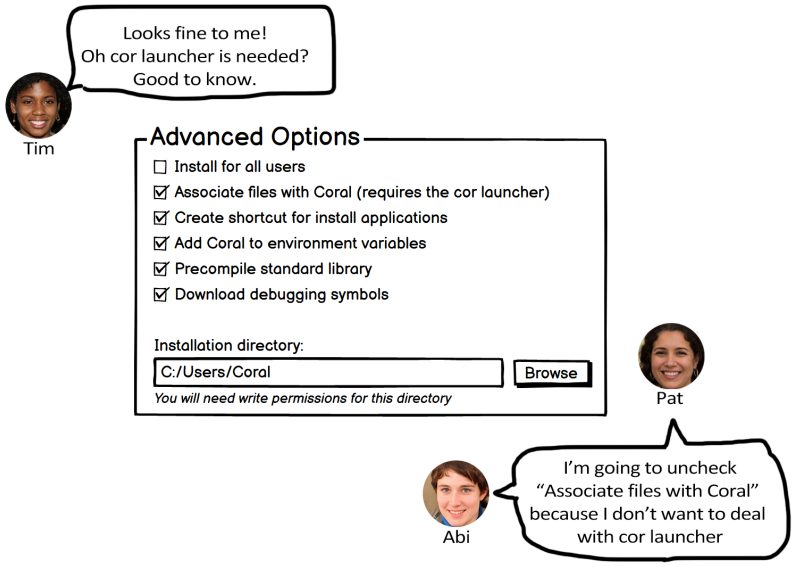


Figure 7.3 *Inclusivity Heuristic #2 Design Example*

Note. Indicating that "cor launcher" is required helps Abi and Pat decide whether they want to proceed or quit, and helps Tim understand what other technical configuration might be required.

7.3.3 Heuristic #3 (of 8)
Let Users Gather as Much Information as They Want, and No More Than They Want

Abi and Pat approach decision-making by diligently gathering and thoroughly reviewing relevant information before acting.

Tim prefers to dive right into the first option that catches their interest and pursue it. They will backtrack if necessary.

Figure 7.4 provides an example design that reflects this heuristic and how Abi, Pat, and Tim might react to it.

To support the personas' **information processing styles**, make it easy for Abi and Pat to gather as much information as they want, and give Tim the ability to quickly gather the useful information they need without having to process a lot of information they don't care about.

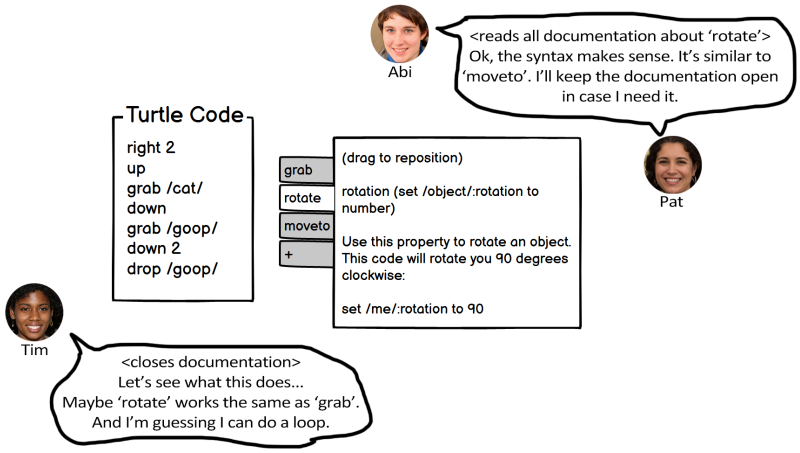


Figure 7.4 *Inclusivity Heuristic #3 Design Example*

Note. The design allows users to access documentation, and keep it open, while coding. This helps Abi and Pat fully understand the syntax before using it. Tim can choose to close the documentation.

7.3.4 Heuristic #4 (of 8)
Keep Familiar Features Available

Abi, who has lower computer self-efficacy and is more risk-averse than Tim, tends toward self-blame and will stop using unfamiliar features if problems arise. Abi prefers to avoid potentially wasting time trying to make unfamiliar features work.

Pat, with moderate technological self-efficacy, adopts a different approach. When faced with problems while using unfamiliar features, Pat will attempt alternative methods to succeed for a while. Being risk-averse, however, Pat prefers to rely on familiar features, which are more predictable in terms of expected outcomes and time required.

In contrast, Tim has higher computer self-efficacy and is more risk-tolerant compared to Abi. If problems arise with unfamiliar features, Tim tends to blame the technology itself and may invest considerable extra time exploring various workarounds to overcome the problem.

Figure 7.5 provides an example design that reflects this heuristic and how Abi, Pat, and Tim might react to it.

To support the personas' **computer self-efficacies** and **attitudes toward risk** while also promoting continued use of the technology without unnecessary time wastage, allow Abi, Pat, and Tim to engage with familiar features that they have previously used.

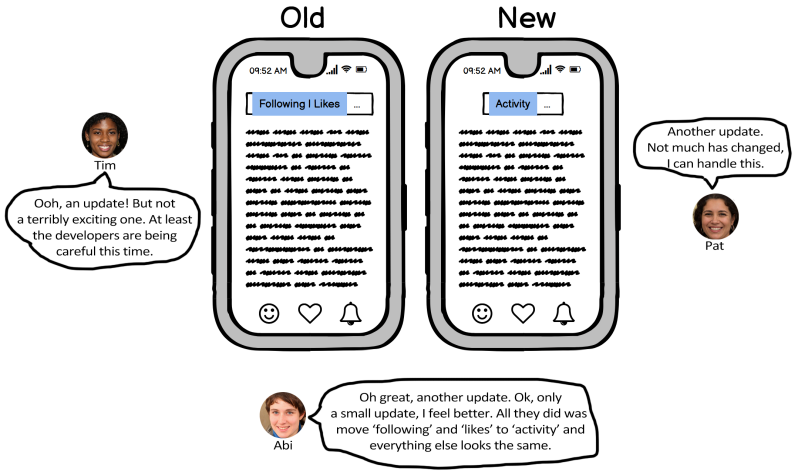


Figure 7.5 *Inclusivity Heuristic #4 Design Example*

Note. The design update is minimal, keeping most features the same. This helps Abi and Pat detect the familiar features with which they're comfortable, and helps Tim detect which features they have already explored.

7.3.5 Heuristic #5 (of 8)
Make Undo/Redo and Backtracking Available

Abi and Pat, being risk-averse, tend to avoid taking actions in technology that may be difficult to undo or reverse. In contrast, Tim, who is risk-tolerant, is willing to take actions in technology that might be incorrect or require reversal.

Figure 7.6 provides an example design that reflects this heuristic and how Abi, Pat, and Tim might react to it.

To support the personas' **attitudes toward risk**, offer Abi and Pat the option to undo/redo actions and backtrack, ensuring they feel at ease when taking actions that have uncertain consequences. This way, they can be confident knowing they can easily reverse these actions if needed. In addition, these features allow Tim to recover from mistakes.

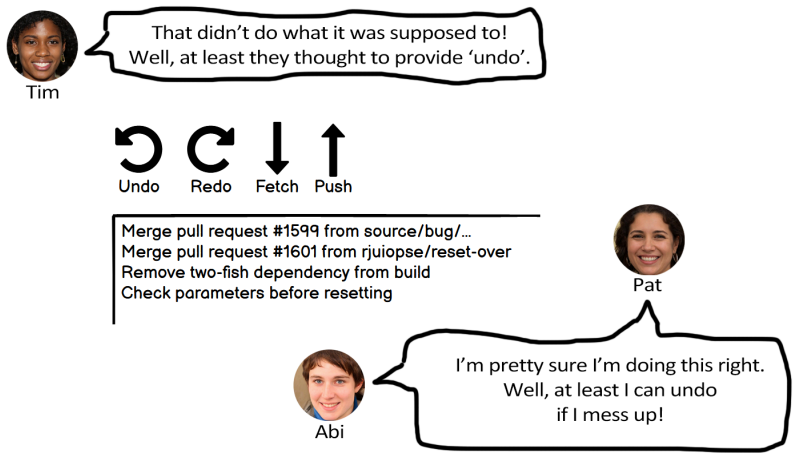


Figure 7.6 *Inclusivity Heuristic #5 Design Example*

Note. The design allows users to undo or redo their last action. This helps Abi and Pat feel assured that using the functionality is safe, and helps Tim backtrack in case they make a mistake.

7.3.6 Heuristic #6 (of 8)
Provide an Explicit Path through the Task

Abi, as a process-oriented learner, prefers to approach tasks in a systematic and step-by-step way.

Tim and Pat, however, who are more inclined toward tinkering as their learning style, prefer not to be confined by strict and predetermined processes. They thrive when they have the freedom to explore and experiment without rigid constraints.

Figure 7.7 provides an example design that reflects this heuristic and how Abi, Pat, and Tim might react to it.

> To support the personas' **learning styles**, offer Abi a well-defined and explicit task process that provides clarity and structure. For Tim and Pat, provide them with the flexibility to bypass step-by-step processes and tutorials that are not necessary for learning the technology. This allows them to explore and learn in a way that suits their preferred approach.

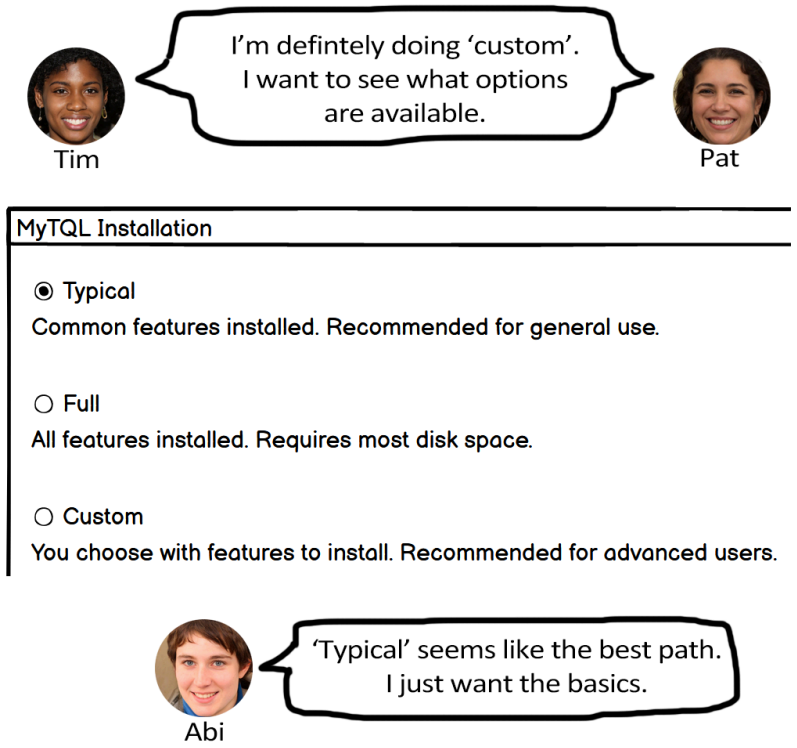


Figure 7.7 *Inclusivity Heuristic #6 Design Example*

Note. The design gives users a clear choice between three paths. A structured process helps Abi feel comfortable. Pat and Tim can select "custom" if they'd like to tinker.

7.3.7 Heuristic #7 (of 8)
Provide Ways to Try Out Different Approaches

Abi, with lower computer self-efficacy compared to Tim, tends toward self-blame when problems arise in technology. As a result, Abi may stop using the tech altogether.

Pat, with moderate self-efficacy in technology, takes a different approach. When faced with problems while using technology, Pat will attempt alternative methods to succeed for a period.

In contrast, Tim, with higher computer self-efficacy than Abi, tends to blame the technology itself if a problem arises. Abi will then explore numerous workarounds in order to overcome the issue.

Figure 7.8 provides an example design that reflects this heuristic and how Abi, Pat, and Tim might react to it.

To support the personas' **computer self-efficacies**, provide Abi with alternative approaches when difficulties with the current approach arise. This will also encourage Tim and Pat to explore multiple strategies to solve problems.

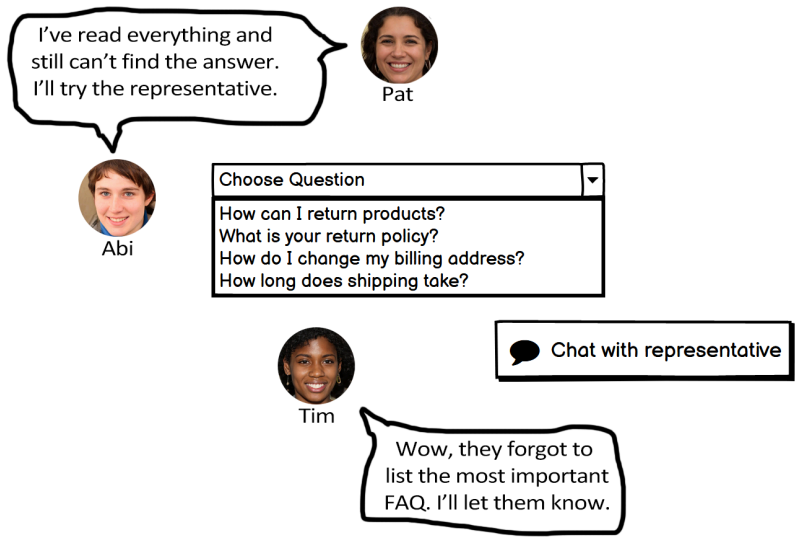


Figure 7.8 *Inclusivity Heuristic #7 Design Example*

Note. The design allows users to chat with a person in case they can't find their question on the list. This helps Abi and Pat because they know they have a backup plan. It also helps Tim, who might want to report the problem.

7.3.8 Heuristic #8 (of 8)
Encourage Tinkerers to Tinker Mindfully

Tim's learning style revolves around tinkering, but at times Tim becomes excessively engrossed in tinkering, leading to long distractions.

Pat, in contrast, embraces a learning approach that involves actively experimenting with new features. Pat does so mindfully, however, taking the time to reflect on each step taken during the learning process.

Figure 7.9 provides an example design that reflects this heuristic and how Tim might react to it.

To support Tim's **learning style**, encourage Tim to avoid excessive tinkering, such as by adding an extra click. This helps minimize mistakes, allows for better absorption of important information, and helps Tim stay focused on the task at hand.

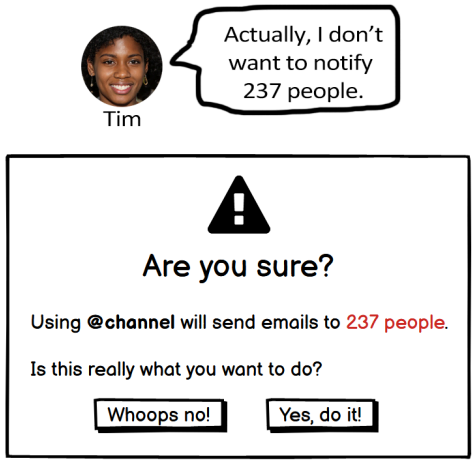


Figure 7.9 *Inclusivity Heuristic #8 Design Example*

Note. The design helps Tim avoid making mistakes while tinkering.

7.4 Summary

The Inclusivity Heuristics are a set of eight software usability heuristics for evaluating and improving the usability of UIs across users with different cognitive styles.

1. Explain (to Users) the Benefits of Using New and Existing Features
2. Explain (to Users) the Costs of Using New and Existing Features
3. Let Users Gather as Much Information as They Want, and No More Than They Want
4. Keep Familiar Features Available
5. Make Undo/Redo and Backtracking Available
6. Provide an Explicit Path through the Task
7. Provide Ways to Try Out Different Approaches
8. Encourage Tinkerers to Tinker Mindfully

References

Burnett, M., Stumpf, S., Macbeth, J., Makri, S., Beckwith, L., Kwan, I., Peters, A., & Jernigan, W. (2016). GenderMag: A method for evaluating software's gender inclusiveness. *Interacting with Computers*, *28*(6), 760–787. https://doi.org/10.1093/iwc/iwv046

Burnett, M., Sarma, A., Hilderbrand, C., Steine-Hanson, Z., Mendez, C., Perdriau, C., Garcia, R., Hu, C., Letaw, L., Vellanki, A., & Garcia, H. (2021, March). *Cognitive style heuristics (from the GenderMag Project)*. GenderMag.org. https://gendermag.org/Docs/Cognitive-Style-Heuristics-from-the-GenderMag-Project-2021-03-07-1537.pdf

GenderMag Project, Di, E., Noe-Guevara, G. J., Letaw, L., Alzugaray, M. J., Madsen, S., & Doddala, S. (2021, June). *GenderMag facet and facet value definitions (cognitive styles)*. OERCommons.org. https://www.oercommons.org/courses/handout-gendermag-facet-and-facet-value-definitions-cognitive-styles

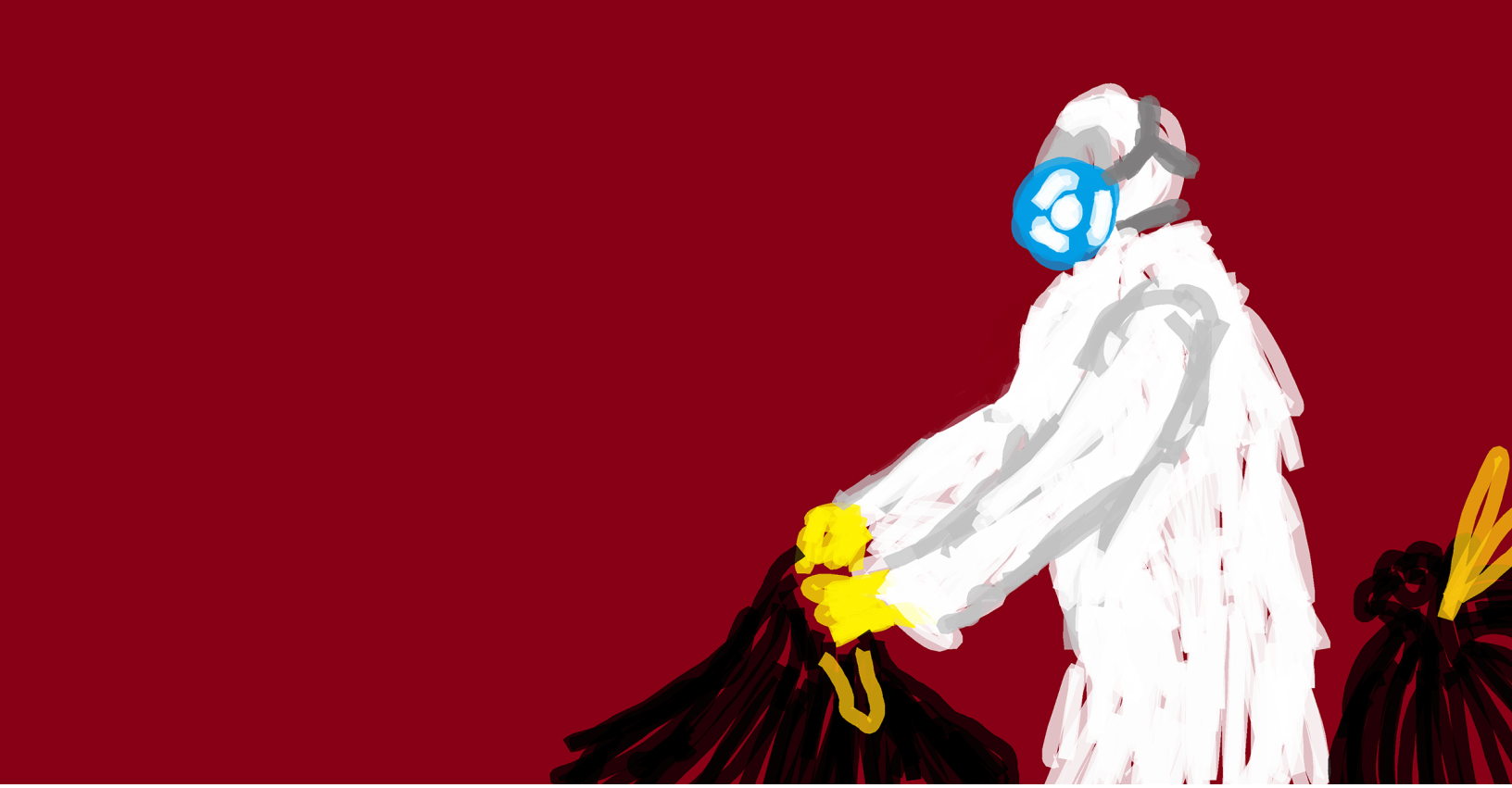
Hu, C., Perdriau, C., Mendez, C., Gao, C., Fallatah, A., & Burnett, M. (2021). Toward a socioeconomic-aware HCI: Five facets. *arXiv preprint arXiv:2108.13477*.

McIntosh, J., Du, X., Wu, Z., Truong, G., Ly, Q., How, R., Viswanathan, S., & Kanij, T. (2021). *Evaluating age bias in e-commerce*. Paper presented at the 2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), Madrid, Spain. https://doi.org/10.1109/chase52884.2021.00012

Microsoft. (n.d.). *Microsoft inclusive design*. https://inclusive.microsoft.design/

Nielsen, J. (1994). Heuristic evaluation. In *Usability inspection methods.* John Wiley & Sons.

Nielsen, J., & Molich, R. (1990). Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems Empowering People—CHI '90*. Association for Computing Machinery. https://doi.org/10.1145/97243.97281



Chapter 8

Code Smells and Refactoring

> If you want to learn more about any of the code smells and refactorings described in this chapter or want to know additional ways your code can smell, Martin (2009), Shvets, and Fowler and Beck (2019) are good resources.

Code smells are indications that the code needs to be reorganized—a sign your software is undergoing code decay. Your code might need attention if you're having thoughts like these:

- "I would **never show this code** during an interview."
- "I'm going to **start over** and rewrite this code from scratch."
- "Every time I look at this code, I have to **re-figure-out** what it does."
- "These **comments don't match the code** . . ."
- "Why is this **code repeated** in three different places?"
- "I want to switch out this component, but **that'll break X, Y, and Z** in this other place, and I don't want to deal with that."

Types of codes smells we'll cover (including how to fix them):

- Code smells about **comments**.
- Code smells about **functions**.
- **General code** smells (e.g., about the code within functions).

8.1 Why Care about Code Smells?

Reasons to pay attention to and fix code smells:

- Smelly code can be **harder for you and others to maintain** because the code is unclear. When code is hard to maintain, developers tend to work around it or re-create the same functionality elsewhere.
- Smelly code **leads to smellier code**. When you let your code become disorganized, you are giving yourself and others the message that smelly code is acceptable. Disorganized code also tends to give us an excuse to be lazy coders. A web development example: if you've used CSS, you may have encountered frustrating situations where the style you're trying to apply is not working—somewhere in the code (e.g., other CSS, HTML, or JS), your style is being overridden. Instead of tracking down the competing code or markup, you use the "!important" property, which forces the style to be applied. The codebase is a mess anyway, so who cares? Your future self.
- Smelly code builds up technical debt. If the code is working, there's never a reason to change it, right? Wrong. Each time you write sloppy code, you are contributing to your project's technical debt. Maybe it works now, but as sloppy software grows, it will get more difficult to deal with. That can mean your company needing to hire more developers to keep productivity up. Instead, productivity can go down because now the old developers are struggling to teach the new developers, and everyone is continuing to write sloppy code (Martin, 2009). Ultimately, the software may have to be redeveloped entirely (which doesn't always solve the problem). Or the project could fail.

8.2 Your Code Stinks—Now What?

If you can (e.g., your manager allows it), strongly consider refactoring. Refactoring is when you improve your code without changing what the code does. Refactoring is a way to pay down technical debt.

The remainder of this chapter is about code smells and how to clean them up. This is not an exhaustive list. You can find more advice in the references at the end of the chapter.

8.3 Comments

When we first learned to code, many of us didn't write comments: solving problems and coding is fun; no time for boring comments! Then, we got more experience, started coding with others, were formally trained to code, or attempted to continue an old project, and we saw why comments are useful—and then some of us jumped to the other extreme: too many comments. We explained functions with paragraphs of prose, or even commented each line. It's tedious, but it's the right thing to do, right? Unfortunately (and fortunately), **too many comments can be as bad as none**.

8.3.1 Drawbacks of Having Many Comments

> Don't fall into the trap of adding excessive comments to your code before an interview! Some prospective employers specifically look for over-commented code (or can't help but see it) as an indicator of poor programming habits.

- Comments **get out of date quickly**. If we update the code, then procrastinate on the comments, what we leave can be misleading (to others and our future selves). Also, more comments mean greater likelihood some will be ignored, giving us the smelly situation of some accurate and some inaccurate comments. In that case, why would we trust any of the comments?
- Writing comments for straightforward code **can distract from the important comments**. If the code was difficult to write, is long, is unique, is complex, or has a "gotcha," comments can help call attention to idiosyncrasies of the code.
- Writing lots of comments could **indicate the code needs to be simplified**. Ideally, most of the code you write will be self-explanatory, so frequent comments are not needed.

8.3.2 Code Smells about Comments

Below is a concise **list of common code smells** about comments and what to do about them (how to refactor).

- **Obsolete Comment** (no longer describes the code). Remove or update.

```
# SMELLY
"""
Uses the TwoFish block cipher with 256 bit key size
"""
ThreeFish(512,data)
```

- **Commented-Out Code** (somebody thought they'd need that code later, but the commented-out block is now getting out of date and in the way). Remove. If you're feeling risk-averse, save a backup or use a version-control system.

> Commenting out code often comes with poor assumptions (e.g., you'll need the code later, others will understand why you commented it out, the surrounding code will continue having the same purpose, and so on).

```
# SMELLY
def updateWorldState():
    """
    updateTime() # might need later
    updatePlayers()
    updatePoints()
    """
    for p in players:
        p.updateState()
```

- **Redundant Comment** (states what would already be immediately apparent to a programmer of any level). Remove. Less is more.

```
# SMELLY
getLength() # gets the length
```

- **Long Comment** (multiple sentences, complicated, goes into a lot of detail). Simplify the code to make it more self-explanatory; shorten or remove comment.

```
# SMELLY
"""
This is the first function I made in this module, and it
    takes the user's Unicode text input, converts it to
    ASCII, then that creates a visualization of a type-
```

```
    writer typing the input. Problem is, as you might
    imagine, sometimes there's no good conversion to
    ASCII, so some meaning is lost.
"""
```

8.4 Functions

> If you're only writing a short program, does coding style matter? Treating code as disposable is a self-fulfilling prophecy.

A natural way to code is to start writing a function and then, as the program gets more complicated, keep adding to it. For example, if your program's GUI only has a start and a stop button, the function for populating the screen with UI elements only needs to draw those two buttons. Then, when you add a menu and a settings button, you could update the function to draw those elements, too. You then add user accounts and decide that function is a fine place to check if the user is logged in, their level of inactivity, show a pop-up about cool new features . . . and your function balloons. Understanding the small details of how the function works can even make one feel proud—until the **code becomes unmaintainable and bug-ridden**.

8.4.1 Code Smells about Functions

> Software made of three to four line functions is amazing to behold!

Follow these **refactoring suggestions** to increase code readability, maintainability, and modularity.

- **Long Function** (more than 10 lines or so). Break into multiple functions. Aim for five lines or fewer.
- **Function with Many Jobs** (doing more than what its name suggests, doing things that aren't closely related, doing many things). Break into multiple functions.

```
# BEFORE
def updateGUI():
    updateTime()
    updateTimeDisplay()
    updateScores()
    updateScoreDisplay()
    refreshWindow()
```

```
8
9  # AFTER
10 def updateState() :
11     updateTime()
12     updateScores()
13
14 def updateGUI():
15     updateTimeDisplay()
16     updateScoreDisplay()
17     refreshWindow()
```

- **Function with Many Parameters** (more than four, some say more than three). As appropriate, pass an object that combines the parameters, make calls within the function to get the parameter data, break into multiple functions, or find another way of reducing the number of parameters.

> Zero function parameters is even better than four!

```
1  # BEFORE
2  initOutdoorPlace(floraList, faunaList, temperature, wind-
       Speed, cloudiness, rockiness, birdNoises,
       grassLength)
3
4  # AFTER
5  initOutdoorPlace(world1data)
```

8.5 Code

Code **gets messy fast** if you're not paying attention. One reason is because many of us weren't trained to be neat with code when we first learned it. To write tidy code, you may have to frequently **stop and think** about its design or be strict with yourself about **refactoring regularly**. Over time, you might adopt better habits.

8.5.1 Code Smells about Code in General

- **Duplicate Code** (same code in multiple places). Consolidate into one place, but watch out for creating unwanted dependencies.

```
# BEFORE
def updateLevelOfAlarm(npc):
    if (npc.isWalking() && npc.isAlive() &&
     npc.isFriendly())
        setLevelOfAlarm(0)
    else
        setLevelOfAlarm(500)
        react(npc)

def react(npc):
     if (npc.isWalking() && npc.isAlive() &&
     npc.isFriendly())
         keepWalking()
     else
         runAway()

 # AFTER
 def react(npc):
     if (npc.isHarmless())
         setLevelOfAlarm(0)
         keepWalking()
     else
         setLevelOfAlarm(500)
         runAway()

 def setLevelOfAlarm(level):
     alarmLevel = level

 def isHarmless(npc):
     return (npc.isWalking() && npc.isAlive() &&
     npc.isFriendly())
```

- **Long Lines** (more than 100 characters or so). Shorten by breaking into multiple lines, converting to a function call, defining new variables, and so on.

> Thresholds like “100 characters” or “five lines” are arbitrary. Generally, shorter is better, but not even that rule can be applied everywhere. For example, “syntactic sugar” is the term for concise and elegant code syntax, usually built into the programming language. It can make your code shorter, but what’s the point if nobody can understand it!

```
# BEFORE
if (rectangle.coordinate[1][0] - rectangle.coordinate
     [2][0] > 500 && rectangle.coordinate[2][1] - rectan-
     gle.coordinate[3][1] > 500 && rectangle.isSquare()):

# AFTER
if (rectangle.isSquare() && rectangle.width > 500):
```

- **Inconsistent Conventions** (formatting code differently in different places, or untidily). Follow whatever style conventions the code is already using. If it's a new project, plan to be self-consistent or follow accepted conventions for the language you're using.

> When adding to another person's code, it's best to follow their coding style conventions even if you prefer a different way. If their code style is sloppy and inconsistent, however, consider whether there's a polite way to fix the problem.

```
# BEFORE
if (whale.isSinging) {
    activateAudioRecordingDevice();
} else {
    recording_device_off_confirmation_check();
}

if (starfish.blockingCamera)
{
    AirCannon.Spray(camera.coordinates);
 }

 # AFTER
 if (Whale.isSinging) {
     activateAudioRecordingDevice();
 } else {
     confirmRecordingDeviceOff();
 }

 if (Starfish.isBlockingCamera) {
 AirCannon.spray(Camera.coordinates);
 }
```

- **Vague Naming** (does not communicate what the function, variable, etc. is for). Rename it, even if the name is long. Long names can sometimes replace comments.

> Wouldn't it be nice if code read like a book?

```
# BEFORE
a = 100
b = 2

# AFTER
retail_price = 100
wholesale_multiplier = 2
```

8.6 Summary

Cleaning up your code can help make your software sustainable and extensible and can make your teammates happier, too.

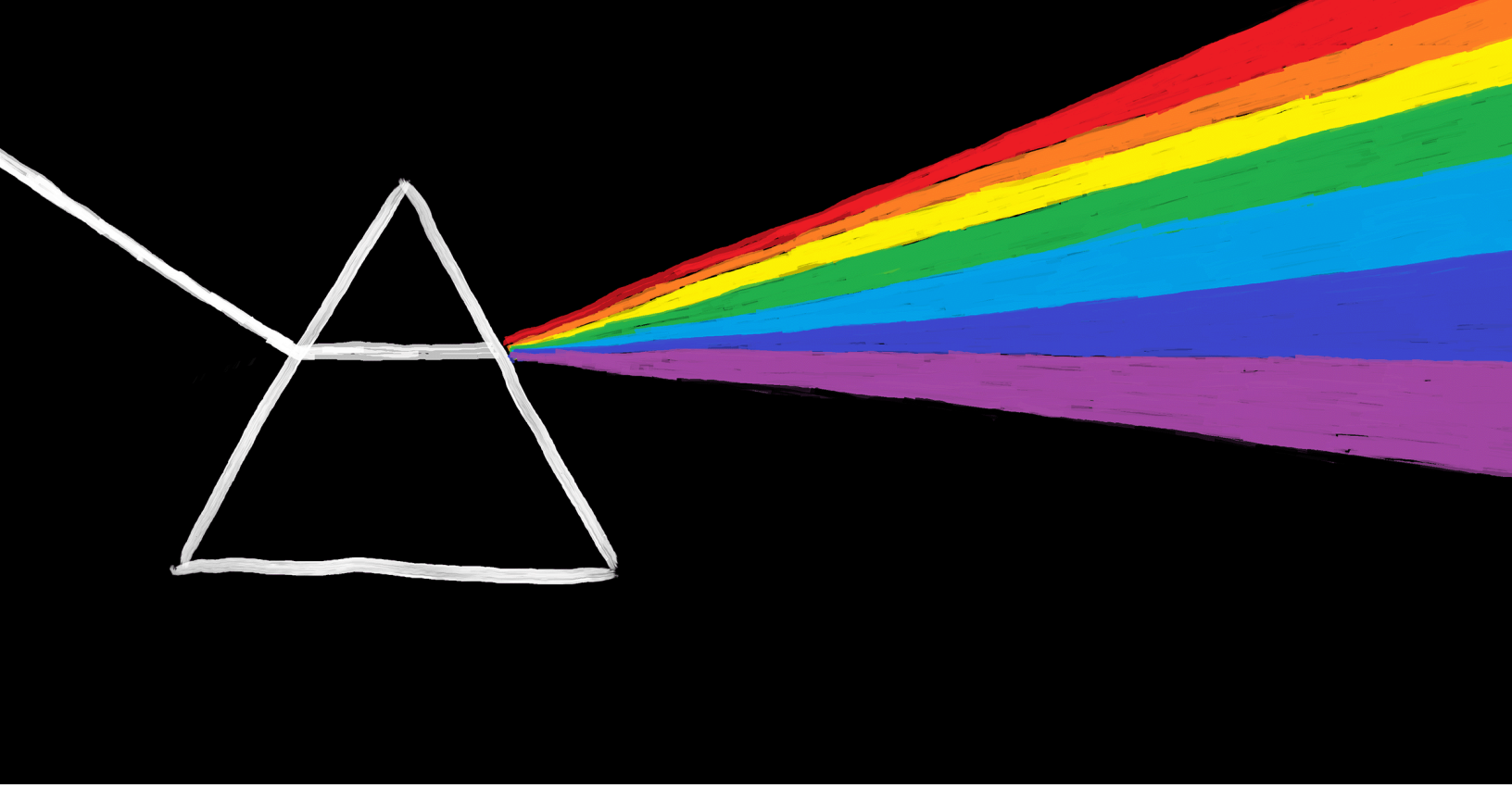
- Obsolete comment? Remove or update.
- Commented-out code? Remove.
- Long comment? Simplify, shorten, or remove.
- Long function (more than ~10 lines)? Split.
- Function with many jobs? Split.
- Function with many parameters? Pass an object, make calls to get the parameter data, or split.
- Duplicate code? Consolidate.
- Long lines (more than ~100 characters)? Shorten, convert to function, or define new variables.
- Inconsistent conventions? Follow existing conventions.
- Vague naming? Rename.

References

Fowler, M., & Beck, K. (2019). *Refactoring: Improving the design of existing code*. Addison-Wesley.

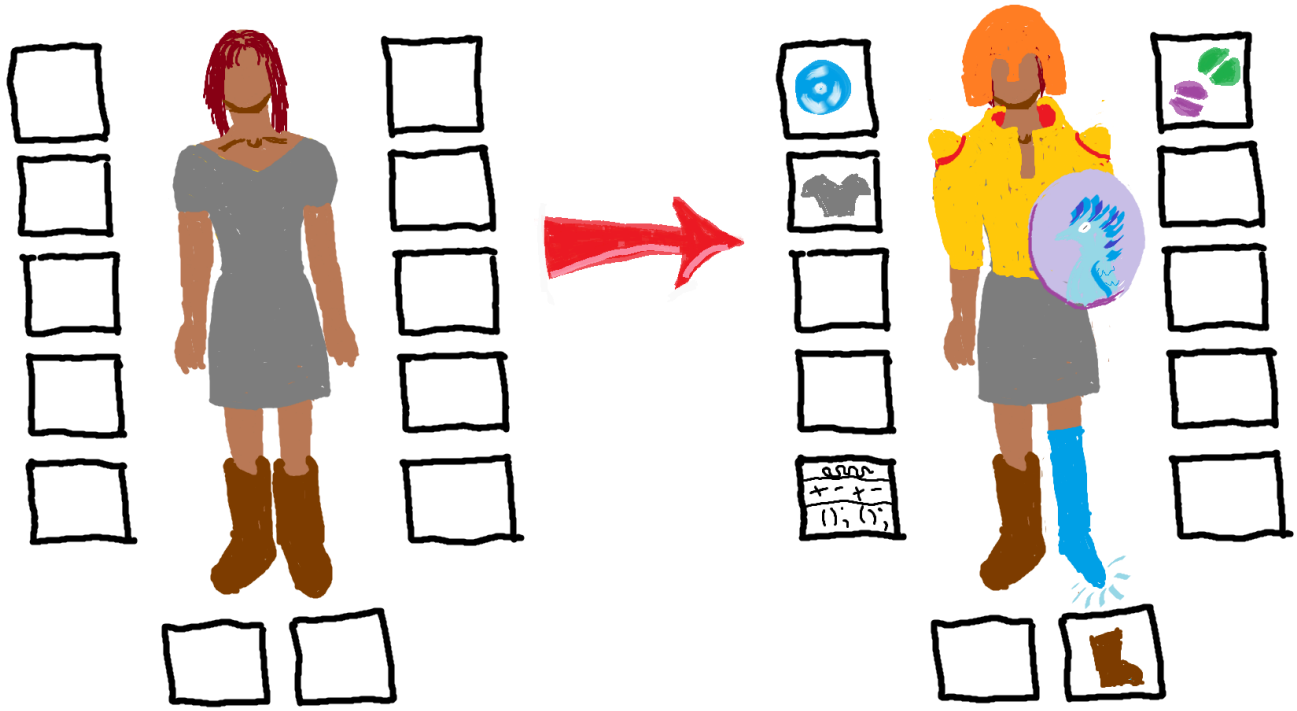
Martin, R. C. (2009). *Clean code: A handbook of Agile Software craftsmanship*. Prentice Hall.

Shvets, A. (n.d.). *Refactoring and Design Patterns*. https://refactoring.guru/



Conclusion

I hope you're now better equipped for your next software project.



The Reader Becoming Better Equipped

Glossary

acceptance criteria
Statements about functionality that, when satisfied, mean the functionality has been satisfactorily implemented.

Agile
A software process model and philosophy for managing and developing software projects. Agile values include individuals and interactions, working software, customer collaboration, and responding to change.

attitude toward risk
(Cognitive facet.) How willing a person is to take chances while using technology (risk-tolerant vs. risk-averse).

business capability
The capacity of a business resource or a combination of resources to generate value for customers by leveraging their environment through a process that involves both tangible and intangible resources. *Source*: Michell, V. (2011). A focused approach to business capability. In *Proceedings of the First International Symposium on Business Modeling and Software Design*, 105–113. University of Reading. https://doi.org/10.5220/0004459101050113

class diagrams
In Unified Modeling Language (UML), a visualization of how classes are built in relation to other classes in object-oriented software. Includes properties and methods of individual classes and "has a" and "is a" relationships between classes.

client-server architecture
High-level architecture characterized by one component (the server) responding to requests and providing resources while other components (clients) request those resources.

clients
One or more people or organizations who are requesting the software be made and have decision-making authority about the software (e.g., because they are paying for it or otherwise providing resources).

code decay
Reduction of code quality over time. Can result in decreased maintainability, more bugs, and irretrievable failure.

code smell
Aspect of code indicating the code is of poor quality (e.g., has detriments to readability and maintainability).

cognitive facet value
A position on the spectrum of a cognitive facet. Also called a "cognitive style."

cognitive facets
Five aspects of users that affect how they solve problems in software: motivations, information processing style, computer self-efficacy, attitude toward risk, learning style.

cognitive style
A person's preferred way of processing (perceiving, organizing and analyzing) information using cognitive mechanisms and structures. They are assumed to be relatively stable. Whilst cognitive styles can influence a person's behavior, depending on task demands, other processing strategies may at times be employed – this is because they are only preferences. *Source*: Armstrong, S.J., Peterson, E.R., & Rayner, S. G. (2012). Understanding and defining cognitive style and learning style: A Delphi study in the context of educational psychology. *Educational Studies, 4*, 449-455. https://doi.org/10.1080/03055698.2011.643110

communication pipe
Technology and/or approach used for sending and receiving messages between processes.

component
Within a codebase, a unit of the code containing related functionality. Ideally, a component is both replaceable and reusable.

computer self-efficacy
(Cognitive facet.) A person's confidence in their ability to use technology (low vs. medium vs. high).

constraint
A restriction; what must be done or not done.

contingency
A future event or circumstance that may occur but depends

on known and unknown factors. Can be difficult to predict far ahead of time.

coupling
The degree to which one unit of code is dependent on another.

Daily Scrum
In Agile Scrum, a 15-minute meeting during which developers discuss what has been done since the last Daily Scrum, what will be done before the next Daily Scrum, and whether there are any blockers.

Definition of Done (DoD)
A set of acceptance criteria that, once satisfied, means a user story has been satisfactorily implemented.

Eisenhower matrix
A grid for helping decide whether to do, delegate, schedule, or eliminate a task based on its urgency and importance.

encapsulation
In object-oriented programming, (1) combining data and the methods that act upon those data into one unit of code or (2) preventing external direct access to data within a unit of code.

estimation
Figuring out ahead of time how long a task is likely to take.

eventual consistency
Characteristic of software systems where different parts of the system can have less up-to-date information (e.g., state, data) than other parts, but the inconsistencies are temporary.

extensibility
Degree to which software supports adding functionality later.

Extreme Programming (XP)
Agile methodology that prioritizes customer satisfaction and communication, short development cycles, iteration, frequent releases, code review, teamwork, pair programming, required unit testing, and implementing only the functionality that's needed.

fist of five
A method for gauging and building group consensus that uses a six-level voting system (zero to five fingers).

focus groups
A structured conversation facilitated by a researcher with a small group of prospective users (typically 6-12 individuals). The aim of this session is to gather insights about the participants' attitudes, opinions, motivations, concerns, and challenges concerning a specific product or topic.

functional requirement
Description of what functionality the software needs to have.

Gantt chart
Horizontal bar chart showing start and end times of activities within a project schedule, along a time line.

GenderMag Method
A method that involves utilizing a specialized cognitive walkthrough and customizable personas (Abi, Pat, and Tim) to identify and address gender-inclusivity issues in software, thus improving its overall gender inclusiveness.

given-when-then
A format for writing an acceptance criterion: Given <context>, when <action>, then <result>. *Source*: Agile Alliance. (n.d.). *What is "given – when – then"?* https://www.agilealliance.org/glossary/gwt/

graphical user interface (GUI)
A user interface with interactive graphics, in contrast to a text-based user interface.

ground rules
A set of statements about the team, agreed to by each team member, for avoiding team conflict and dysfunction.

heuristic evaluation
A usability inspection method in which evaluators independently examine a design to ensure it aligns with a predetermined set of heuristics, and then compare their findings. *Source*: Nielsen, J., & Molich, R. (1990). Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems Empowering People—CHI '90*. Association for Computing Machinery. https://doi.org/10.1145/97243.97281

high-fidelity prototype
A polished illustration that looks like a finished, publishable user interface design (especially a GUI). Almost always digital.

high-level architecture
Abstract representation of overall code design; covers all parts of the software.

ideal days
The number of days it would take to complete the work if the work could be 100% focused on.

implementation
(Software development life cycle phase.) Using the requirements and design to code the software.

inclusive design
Designing with the goal of increasing usability for traditionally underserved user populations while also increasing usability for mainstream users.

Inclusivity Heuristics
Guidelines for making software inclusive to diverse users.

Increment
In Agile Scrum, a measurable increase in functionality toward completing the Product Goal.

information processing style
(Cognitive facet.) How a person gathers data in relation to acting on those data (comprehensive vs. selective).

inspection method
Any approach in which an evaluator examines a user interface.

integrated development environment (IDE)
Software for developing software.

interaction design
A method of designing technology that focuses on aiding users in comprehending the operations and events occurring within the technology, as well as guiding them on the available actions they can take.

interaction diagram
Visualization of collaboration between different parts of software.

INVEST
Characteristics of good user stories (independent, negotiable, valuable, estimable, small, testable). *Source*: Wake, B. (2003, August 17). *Invest in good stories, and Smart Tasks*. XP123 Exploring Extreme Programming. https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

iteration
Verb: Revision. Noun (in Agile): A time-boxed software development cycle.

iteration plan
In Agile, establishing what will be done during a development cycle.

JavaScript Object Notation (JSON)
A lightweight data-interchange format that is easy for humans to read and write. *Source*: JSON.org. (n.d.). *Introducing JSON*. https://www.json.org/json-en.html

learning style
(Cognitive facet.) How a person prefers to move through software (by tinkering vs. by mindful tinkering vs. by process).

low-fidelity prototype
A rough sketch of a user interface design (especially a GUI). Can be hand-drawn or digital.

maintenance
Development activities that improve software but that are unrelated to implementing new features (e.g., correcting bugs, improving organization of code, and the like).

managerial skill mix (MSM)
Three categories of skills used by managers: (1) interpersonal, (2) technical, (3) administrative/conceptual. *Source*: Badawy, M. K. (1995). *Developing managerial skills in engineers and scientists: Succeeding as a technical manager*. Van Nostrand Reinhold.

medium-fidelity prototype
A careful and detailed illustration of a user interface design

(especially a GUI). Can be hand-drawn, but digital is more common.

method
A preestablished way of achieving a specific outcome.

microservices architecture
High-level architecture characterized by multiple independent components that each run in their own process and communicate between one another without direct access.

minimum viable product (MVP)
A cost-effective and efficient approach that allows for improved evaluation of potential user interest in a product before it is fully developed. *Source*: Olsen, D. (2015). *The lean product playbook: How to innovate with minimum viable products and rapid customer feedback*. Wiley.

mitigation plan
What will be done if a contingency happens.

monolith architecture
High-level architecture characterized by being in one or few pieces; cannot be easily divided into components that run separately and are independently useful.

motivations
(Cognitive facet.) What keeps someone using technology (task completion vs. tech interest).

nonfunctional requirement
Description of how well software is expected to perform or what constraints or limitations it must respect.

pairwise comparison
A process in which entities are compared in order to determine which is preferred.

paper prototype
A manually created drawing utilized to convey a prospective user interface design that is intended for implementation, particularly a design focused on graphical user interface.

participant
In a Unified Modeling Language sequence diagram, the columns. They can represent objects, users, or other entities involved in a program's execution.

persona
Fictitious character created to represent specific user subsets within a target audience. They are commonly used in marketing and user interface design to aid in the concentration on particular groups of users and customers.

planning fallacy
An optimism bias in which predictions regarding the time required to complete a future task tend to underestimate the actual time needed.

planning poker
In Agile, a consensus-based method of assigning estimates to a task that involves individuals on a team each making their own estimate privately, then sharing with the team, discussing, and re-estimating as needed.

prioritization
Deciding which units of work to complete before others.

Product Backlog
In Agile Scrum, an ordered list of all that is known to be needed to improve a product.

Product Owner
In Agile Scrum, the person who is responsible for guiding the Scrum Team on making the most valuable software possible.

project management
The process of planning and executing a project while balancing the time, cost, and scope constraints.

project management system
Software for planning, organizing, and otherwise carrying out a project.

project network diagram
Graph showing the order in which a project's activities are to be completed.

project priority matrix
A 3 × 3 grid for documenting how to respond when there are potential changes to a project's time, cost, or scope. Options include allowing only positive change (constrain), allowing negative change (accept), or seeking positive change (enhance).

pseudocode
Fake code. Pseudocode looks like code but doesn't follow the rules of a particular programming language. Used to communicate programming concepts.

quality attribute
A characteristic of software used to describe how good it is.

RACI matrix
In project management, a chart for defining which roles are responsible (R) and accountable (A) for a task or deliverable, and which roles should be consulted (C) or informed (I) about the status of the task or deliverable.

refactoring
Improving code design without changing what the code does.

release plan
What will be completed for a specific software release and when the release will occur.

requirement
A rule the software must conform to, including what the software must do, how well it must do what it does, or the software's limitations or constraints.

requirements elicitation
The process of gathering requirements from project stakeholders.

requirements specification
Converting stakeholder requests into written requirements.

risk
Estimated probability of a negative contingency given known and unknown factors.

risk mitigation
An action taken in order to avoid a contingency.

scheduling
Deciding when project activities are to be completed, how long they will take, and what resources are needed to complete them.

scope
The boundaries and deliverables of a project.

Scrum
An Agile framework designed for the development and maintenance of complex software.

Scrum board
A way to organize and visualize tasks or work as cards on a board. The board has columns for different categories, and each card is placed within a column. A Scrum board could be a physical bulletin board with sticky notes or index cards. It is also a common feature of task management software.

Scrum Master
In Agile Scrum, the person who is responsible for making sure the Scrum Team is following Scrum.

sequence diagram
In Unified Modeling Language, an interaction diagram showing how different participants (e.g., users, software components, classes, etc.) collaborate during a single use case.

service
A unit of software that receives and fulfills requests.

software architecture
Code design. Can be shown at different levels of abstraction and detail.

software development life cycle (SDLC)
Phases through which a software's development proceeds: requirements, design, implementation, testing, maintenance.

software engineering
Systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software. *Source*: International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers. (2017). *Systems and software engineering — Vocabulary* (ISO/IEC/IEEE Standard No. 24765:2017). https://www.iso.org/standard/71952.html

software process model
A philosophy and/or set of approaches for software development and/or software project management.

software requirements specification (SRS)
A document that contains software requirements.

spike
A quick and to-the-point investigation for gathering information to help the team answer a question or choose a development path.

Sprint
In Agile Scrum, a development period (a month or less).

Sprint Backlog
In Agile Scrum, the set of activities to be completed during a Sprint (from Product Backlog), the associated Sprint Goal, and a plan for completing the activities.

Sprint Goal
In Agile Scrum, the overall objective of the Sprint.

Sprint Planning
In Agile Scrum, the activity of decided what work will be done during the Sprint.

Sprint Retrospective
In Agile Scrum, a meeting during which the Scrum Team discusses how the last Sprint went in terms of individuals, interactions, processes, tools, and the Sprint Definition of Done.

Sprint Review
In Agile Scrum, a meeting during which the Scrum Team and other stakeholders discuss what happened during the Sprint and what to do during future Sprints.

stakeholders
Anyone who is or will be affected by the software or its development (e.g., clients, companies, users, developers, managers, politicians, and so on).

story points
A method for estimating an activity based on its size relative to other activities. Scale established by team.

sustainability
Degree to which software can continue to function over time (e.g., measured in time and how well the software is functioning).

task management
The collection, assignment, sharing, tracking, and scheduling of tasks. *Source*: Gil, Y., Groth, P., & Ratnakar, V. (2009). Leveraging social networking sites to acquire rich task structure. In *Proceedings of the Workshop on User-Contributed Knowledge and Artificial Intelligence: An Evolving Synergy (WikiAI)*.

task management system
Software for planning and organizing project activities.

tech stack
The set of programming languages, frameworks, and other technologies chosen or needed for implementing a piece of software.

technical debt
Time and resources you (or someone else) will need to spend on modifying your software in the future because of the poor decisions you're making in the present.

think-aloud protocol
A feedback-gathering method to assess the usability of a design, wherein a test user verbalizes their thoughts and impressions while interacting with the design.

triple constraint
In project management, the three limiting factors that govern project execution: time, cost, and scope. Scope includes quality. Cost includes spending money and resources.

Tuckman's model of team development
A five-stage model of how a team develops over time: (1) forming, (2) storming, (3) norming, (4) performing, (5) adjourning.

Unified Modeling Language (UML)
A set of notation and methods for describing and designing software.

usability testing
The act of observing individuals as they attempt to interact with your software.

use case
A formal agreement outlining the expected behavior of a system.

user acceptance testing (UAT)
Formally testing software with end users to check not only whether it performs as expected but also whether end users will use it. Typically performed before the software is released.

user interface (UI)
What a user interacts with to operate a system (e.g., a graphical user interface, a command-line interface, a virtual or augmented reality interface, and the like).

user story
A concise and straightforward explanation of a feature presented from the viewpoint of the individual seeking the new functionality, typically a user or customer of the system.

validation
Confirming that software meets users' needs ("Did we build the right software?").

velocity
In Agile, a measure of how much work is being completed.

verification
Confirming that software satisfied its requirements ("Did we build the software right?").

Waterfall
Way of going about software development and management that is characterized by extensive planning, comprehensive documentation, and moving linearly through stages of the software development life cycle (SDLC).

References

@ShitUserStory. (n.d.). *Shit User Story*. Twitter. https://twitter.com/shituserstory

Agile Alliance. (n.d.). *What is "given – when – then"?* https://www.agilealliance.org/glossary/gwt/

Alcalá-Fdez, J., Alonso, J. M., Castiello, C., Mencar, C., & Soto-Hidalgo, J. M. (2019, June). *Py4JFML: A Python wrapper for using the IEEE Std 1855-2016 through JFML*. Paper presented at the 2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), New Orleans, LA, USA. https://ricerca.uniba.it/bitstream/11586/256332/9/PID5822281-PrePrint%28con-DOI%29.pdf

Badawy, M. K. (1995). *Developing managerial skills in engineers and scientists: Succeeding as a technical manager*. Van Nostrand Reinhold.

Beck, K., & Andres, C. (2004). *Extreme programming explained: Embrace change* (2nd ed.). Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile software development*. https://agilemanifesto.org/

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile software development*. https://agilemanifesto.org/

Belling, S. (2020). Agile values and practices. In *Succeeding with Agile Hybrids*, 47–61. Springer.

Burnett, M., Sarma, A., Hilderbrand, C., Steine-Hanson, Z., Mendez, C., Perdriau, C., Garcia, R., Hu, C., Letaw, L., Vellanki, A., & Garcia, H. (2021, March). *Cognitive style heuristics (from the GenderMag Project)*. GenderMag.org. https://gendermag.org/Docs/Cognitive-Style-Heuristics-from-the-GenderMag-Project-2021-03-07-1537.pdf

Burnett, M., Stumpf, S., Macbeth, J., Makri, S., Beckwith, L., Kwan, I., Peters, A., & Jernigan, W. (2016). GenderMag: A method for evaluating software's gender inclusiveness. *Interacting with Computers*, *28*(6), 760–787. https://doi.org/10.1093/iwc/iwv046

CEAP. Conservation Effects Assessment Project. (2006). *System requirements specification for STEWARDS.* US Department of Agriculture, Agricultural Research Service. https://www.nrcs.usda.gov/publications/ceap-watershed-2006-stewards-design.pdf

Cohn, M. (2004). *Example user stories*. Mountain Goat Software. https://www.mountaingoatsoftware.com/uploads/documents/example-user-stories.pdf

Cohn, M. (2006). *Agile estimating and planning*. Prentice Hall Professional Technical Reference.

Cotton, G. (2013, August 13). *Gestures to avoid in cross-cultural business: In other words, "keep your fingers to yourself!"* HuffPost. https://www.huffpost.com/entry/cross-cultural-gestures_b_3437653

Doulis, G., Frauendiener, J., Stevens, C., & Whale, B. (2019). COFFEE—An MPI-parallelized Python package for the numerical evolution of differential equations. *SoftwareX*, 10, 100283. https://www.sciencedirect.com/science/article/pii/S2352711019300950

Eaker, F. (2006, November). *Software requirements specification*. Vyasa. https://vyasa.sourceforge.net/vyasa_software_requirements_specification.pdf

Eerland, W., Box, S., Fangohr, H., & Sóbester, A. (2017). Teetool—A probabilistic trajectory analysis tool. *Journal of Open Research Software*, *5*(1). https://openresearchsoftware.metajnl.com/articles/10.5334/jors.163

Faris, H., Aljarah, I., Mirjalili, S., Castillo, P. A., & Guervós, J. J. M. (2016). EvoloPy: An open-source nature-inspired optimization framework in python. In *Proceedings of the 8th International Joint Conference on Computational Intelligence (IJCCI): Evolutional Computational Theory and Applications (ECTA), 1*, 171-177. https://research-repository.griffith.edu.au/bitstream/handle/10072/401215/Estivill-Castro165057-Published.pdf?sequence=2

Fern, A. (2022). *Tech Talk Tuesday: Lessons in real-world software: going from monolith to microservices*. OSU MediaSpace. https://media.oregonstate.edu/media/t/1_ls3xsa6r

Fletcher, A. (2002). *Firestarter Youth Power Curriculum*. Freechild Institute for Youth Engagement. https://freechildinstitute.files.wordpress.com/2023/04/firestarter-participant-guidebook.pdf

Fowler, M. (2004). *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley Professional.

Fowler, M. (2015, July 1). *Microservice trade-offs*. martinfowler.com. https://martinfowler.com/articles/microservice-trade-offs.html

Fowler, M. (2019, August 21). *Microservices guide*. martinfowler.com. https://martinfowler.com/microservices/

Fowler, M., & Beck, K. (2019). *Refactoring: Improving the design of existing code*. Addison-Wesley.

Gallard, F., Vanaret, C., Guénot, D., Gachelin, V., Lafage, R., Pauwels, B., Barjhoux, P.-J., & Gazaix, A. (2018). *GEMS: A Python library for automation of multidisciplinary design optimization process generation*. Paper

presented at the 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Kissimmee, FL, USA.. https://hal.science/hal-02335530/file/DTIS19188.1570026732_preprint.pdf

GenderMag Project, Di, E., Noe-Guevara, G. J., Letaw, L., Alzugaray, M. J., Madsen, S., & Doddala, S. (2021, June). *GenderMag facet and facet value definitions (cognitive styles)*. OERCommons.org. https://www.oercommons.org/courses/handout-gendermag-facet-and-facet-value-definitions-cognitive-styles

Graser, A., & Olaya, V. (2015). Processing: A python framework for the seamless integration of geoprocessing tools in QGIS. *ISPRS International Journal of Geo-Information*, *4*(4), 2219-2245. https://www.mdpi.com/2220-9964/4/4/2219/pdf

Hanington, B. M., & Martin, B. (2019). *Universal Methods of Design: 125 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers.

Hedberg, T., Helu, M., & Newrock, M. (2017, December). *Software requirements specification to distribute manufacturing data*. NIST Advanced Manufacturing Series 300-2. National Institute of Standards and Technology. https://nvlpubs.nist.gov/nistpubs/ams/NIST.AMS.300-2.pdf

Hewlett Packard Enterprise (2017). *Agile is the new normal: Adopting Agile project management*. https://softwaretestinggenius.com/docs/4aa5-7619.pdf

Hu, C., Perdriau, C., Mendez, C., Gao, C., Fallatah, A., & Burnett, M. (2021). Toward a socioeconomic-aware HCI: Five facets. *arXiv preprint arXiv:2108.13477*.

Hulshult, A. R., & Krehbiel, T. C. (2019). Using eight agile practices in an online course to improve student learning and Team Project Quality. *Journal of Higher Education Theory and Practice*, *19*(3). https://doi.org/10.33423/jhetp.v19i3.2116

Institute of Electrical and Electronics Engineers. (2020, June). IEEE code of Ethics. IEEE. https://www.ieee.org/about/corporate/governance/p7-8.html

International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers. (2017). *Systems and software engineering—Vocabulary* (ISO/IEC/IEEE Standard No. 24765:2017). https://www.iso.org/standard/71952.html

Larson, E. W., & Gray, C. F. (2018). *Project management the managerial process*. McGraw-Hill Education.

Lewis, J., & Fowler, M. (2014, March 25). *Microservices*. martinfowler.com. https://martinfowler.com/articles/microservices.html

Mahnič, V., & Hovelja, T. (2012). On using planning poker for estimating user stories. *Journal of Systems and Software*, *85*(9), 2086–2095. https://doi.org/10.1016/j.jss.2012.04.005

Martin, R. C. (2009). *Clean code: A handbook of Agile Software craftsmanship*. Prentice Hall.

McIntosh, J., Du, X., Wu, Z., Truong, G., Ly, Q., How, R., Viswanathan, S., & Kanij, T. (2021). *Evaluating age bias in e-commerce*. Paper presented at the 2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), Madrid, Spain. https://doi.org/10.1109/chase52884.2021.00012

Michell, V. (2011). A focused approach to business capability. In B. Shishkov (Ed.), *Proceedings of the First International Symposium on Business Modeling and Software Design*, 105–113. Springer. https://doi.org/10.5220/0004459101050113

Microsoft. (n.d.). *Microsoft inclusive design*. https://inclusive.microsoft.design/

Naressi, A., Couturier, C., Castang, I., De Beer, R., & Graveron-Demilly, D. (2001). Java-based graphical user interface for MRUI, a software package for quantitation of in vivo/medical magnetic resonance spectroscopy signals. *Computers in Biology and Medicine*, *31*(4), 269-286. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=cdb5e5d28a9bd6a04f969d6465110f875e706e71

Nielsen, J. (1994). Heuristic evaluation. In *Usability inspection methods.* John Wiley & Sons.

Nielsen, J., & Molich, R. (1990). Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems Empowering People—CHI '90*. Association for Computing Machinery. https://doi.org/10.1145/97243.97281

OpenVIBE. (2018, April). *Inria Innovation Lab Certivibe v 1.0 software requirement specification*. http://openvibe.inria.fr/openvibe/wp-content/uploads/2018/04/CERT-Software-Requirement-Specification.pdf

Palma, S. D., Di Nucci, D., & Tamburri, D. (2021). RepoMiner: A language-agnostic Python framework to mine software repositories for defect prediction. *arXiv preprint arXiv:2111.11807*. https://arxiv.org/pdf/2111.11807.pdf

Royce, W. W. (1970). Managing the development of large software systems. *Proceedings of IEEE WESCON, 26*, 1-9.

Schwaber, K., & Sutherland, J. (2020, November). *The 2020 scrum guide*. https://scrumguides.org/scrum-guide.html

Shvets, A. (n.d.). *Refactoring and Design Patterns*. https://refactoring.guru/

Snyder, C. (2011). *Paper prototyping: The fast and easy way to design and refine user interfaces*. Morgan Kaufmann.

Spyridonos, P. (2010, February 6). *Software requirements specification for PDF split and merge requirements for version 2.1.0*. University of Kentucky Software Verification and Validation Lab. https://selab.netlab.uky.edu/~ashlee/cs617/project2/PDFSam.pdf

Standish Group International, Inc. (2015). *CHAOS report 2015*. https://standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf

Stuart, A. (2014). *Ground rules for a high performing team*. Paper presented at PMI Global Congress 2014—North America, Phoenix, AZ. Project Management Institute.

Texas Department of Information Resources. (2008, January 14). *Software requirements specification instructions*. https://dir.texas.gov/sites/default/files/Requirements%20Traceability%20Matrix%20Instructions.pdf

Thorp, K. R. (2022). pyfao56: FAO-56 evapotranspiration in Python. *SoftwareX*, *19*, 101208. https://www.ars.usda.gov/ARSUserFiles/40820/Thorp2022%20-%20pyfao56.pdf

Tuckman, B. W. (1965). Developmental sequence in small groups. *Psychological Bulletin*, *63*(6), 384–399. https://doi.org/10.1037/h0022100

Tuckman, B. W., & Jensen, M. A. (1977). Stages of small-group development revisited. *Group and Organization Studies*, *2*(4), 419–427. https://doi.org/10.1177/105960117700200404

US General Services Administration. (2014, January). *USDA personas and use cases*. https://s3.amazonaws.com/digitalgov/_legacy-img/2014/01/Marsh-Personas.pdf

van Wyngaard, C. J., Pretorius, J. H., & Pretorius, L. (2012). *Theory of the triple constraint—A conceptual review*. Paper presented at the 2012 IEEE International Conference on Industrial Engineering and Engineering Management, Hong Kong, China. https://doi.org/10.1109/ieem.2012.6838095

Wake, B. (2003, August 17). *Invest in good stories, and Smart Tasks*. XP123 Exploring Extreme Programming. https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

Wells, D. (2013, October 13). *Extreme programming: A gentle introduction.* http://www.extremeprogramming.org/

Wiegers, K., & Beatty, J. (2013). *Software requirements* (3rd ed.). Developer Best Practices Series. Microsoft Press.

Wikimedia Foundation. (2023, March 23). *List of system quality attributes*. https://en.wikipedia.org/wiki/List_of_system_quality_attributes