ITERATION 1.2 (March 2022)

Handbook of

SOFTWARE ENGINEERING METHODS

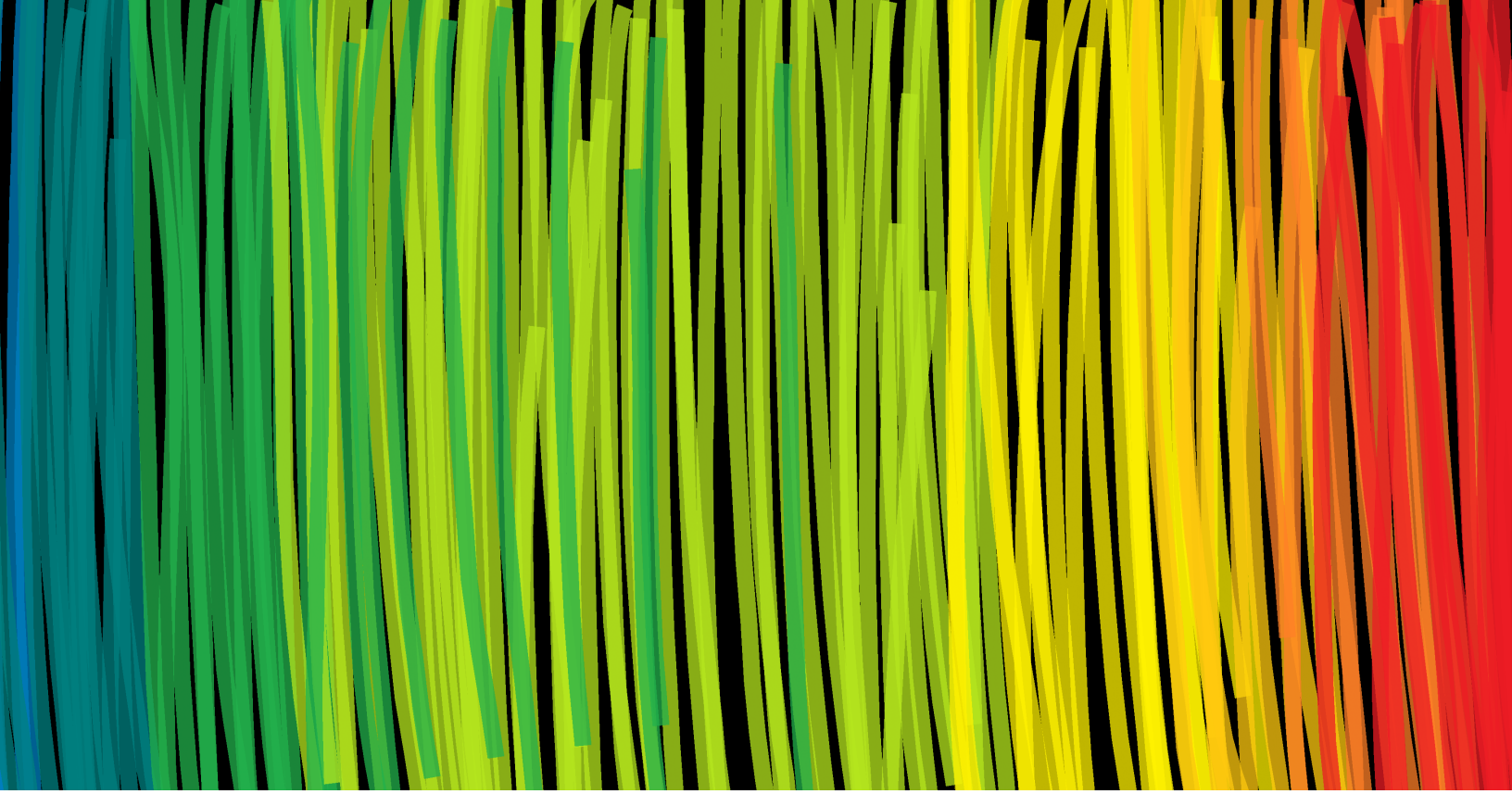
Lara Letaw

Handbook of Software Engineering Methods

Lara Letaw

March 28, 2022

Contents



Chapter 1

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.

1.1 What's software engineering?

Let's build a definition from the bottom up:

- Software engineering is **not** the same as **programming**

- Software engineering involves trying to apply **methods**
- Software engineering involves trying to make programs that have a long lifespan (**sustainable**)
- Software engineering involves trying to make programs that can be added to (**extensible**)
- Software engineering involves trying to balance time, cost, and scope (the **triple constraint**)
- Software engineering often involves **teamwork**
- Software engineering involves trying to solve **problems people care about**
- Software engineering involves both **artistry** and **science**.

Our definition:

> **Software engineering** is the art and science of using different methods to efficiently create extensible, sustainable programs that solve problems people care about.

method: A pre-established way of achieving a specific outcome.

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

extensible: Built in such a way to support adding more functionality later.

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.

software engineering: The art and science of using different methods to efficiently create extensible, sustainable programs that solve problems people care about.

1.2 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.

1.2.1 Software engineering is not black and white

Throughout the book, I've tried to communicate that **software engineering is the gray area of computer science**. "Right" answers can be difficult 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 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?"

1.2.2 Studying every detail of software engineering is a waste of time

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

1.2.3 Agile isn't perfect but I really like it (and other people do too)

This book leans so far toward **Agile** the two are probably in a relationship. 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-end-all, so I'm planning to incorporate more from other **software process models** in the future.

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

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

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

increment: In software, a measurable increase in functionality.

1.3 What's this book like?

It was **written iteratively** ("Do something. Now do it again, but better") **and incrementally** ("Now 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 & 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 (UML) class and sequence diagrams**: A couple types of diagrams useful for communicating how your code works (or should work)

Over here in the margin is where to find **definitions** (also in the Glossary).

This is also where to find **asides**: Comments that are related to the content but don't fit into its flow or seem worth emphasizing.

5. **Monolith vs. microservices architectures**: Two contrasting high-level ways to organize code
6. **Paper prototyping**: Creating a good user interface design before coding it
7. **Cognitive style heuristics**: Making software work well for different kinds of people who are not like you
8. **Code smells & refactoring**: Making your code nicer to work with

This book might get shorter before it gets longer; I've tried to keep chapters concise but informative.

It's **short** and meant to be **readable**:

- Important terms and concepts are **bolded**
- Margins contain **term definitions** and side notes (relevant additional thoughts)
- **Additional resources** are listed at the end of each major chapter

My aim is that you be able 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.

1.4 What's the future of this book?

For **source files, updated versions, or to make suggestions**: `https://github.com/setextbook`

I'll keep iterating and incrementing. If you have content requests, suggestions, or other feedback, you can create an issue or pull request on this book's GitHub repository: `https://github.com/setextbook`.

Potential future additions:

- Debugging
- Deployment
- DevOps
- Ethics
- More software architectures
- More software process models
- Object-oriented design principles
- Professionalism
- Software used by software engineers
- Testing your code (verification)

- More examples, figures, and images *

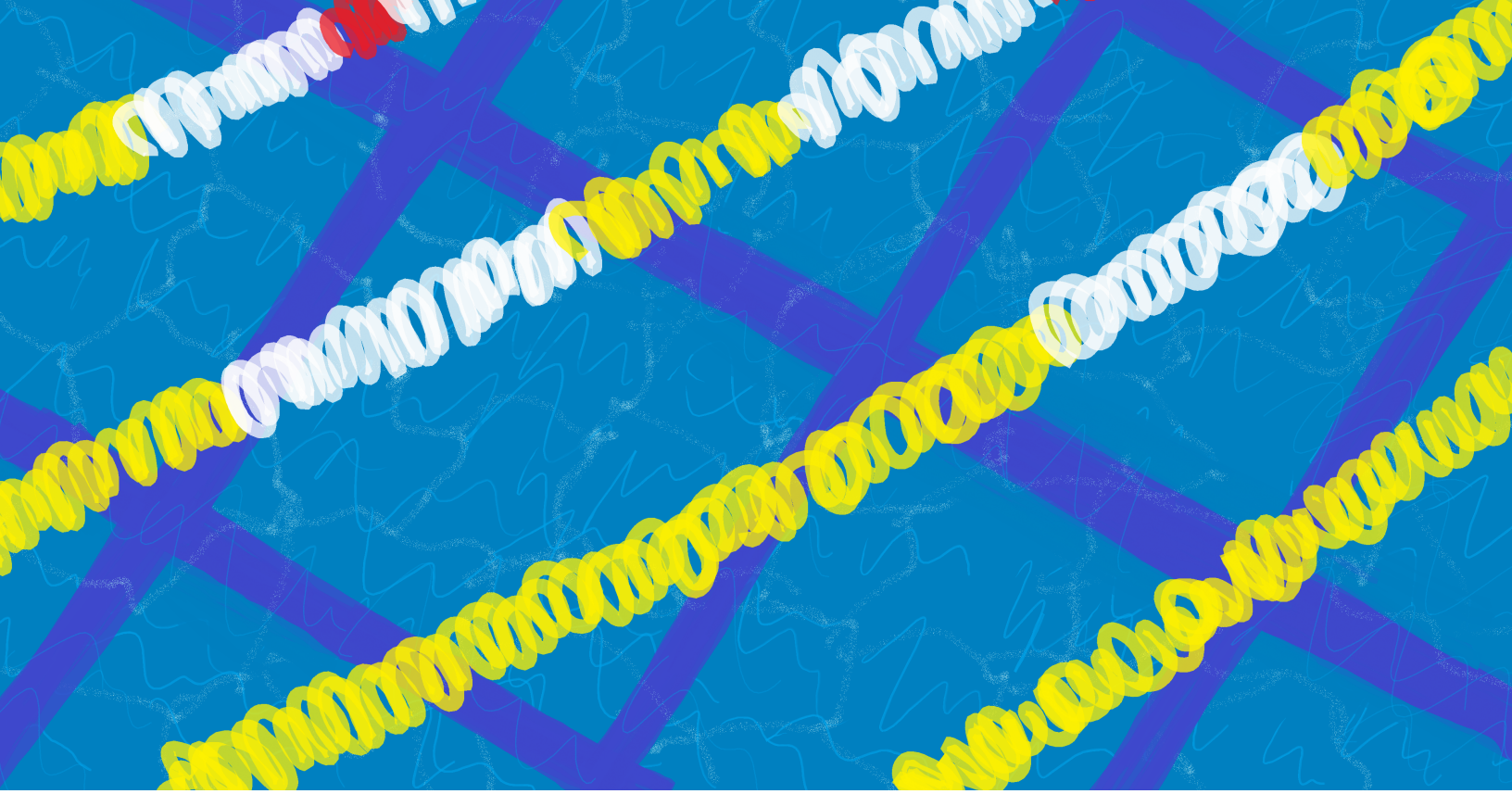
* Yep, I'm the "illustrator" (a generous title).

This book could also become part of your own book / course / blog / etc.—feel free to use the whole thing or pieces of it (non-commercially).

1.5 License

1.6 Acknowledgments

Thanks to Caius Brindescu, Raffaele de Amicis, Sèanar Letaw, and Tiffany Rockwell for their feedback, advice, and support. Additional 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, and Jack LaBarba. Thanks to the Oregon State University Open Educational Resources (OER) Unit for making the whole effort possible.



Chapter 2

Agile

This book is geared toward **Agile**, but there are other **software process models**. Each software process model has a different way of proceeding through the **software development lifecycle (SDLC)**. This chapter starts by describing the SDLC and Agile versus another software process model. That is followed by a discussion of **Scrum** (an Agile framework) and 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 Additional Resource section at the end of the chapter.

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

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

2.1 Software Development Lifecycle (SDLC)

Scrum: An Agile framework "for developing and sustaining complex products." (Schwaber and Sutherland 2020)

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

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

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

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

increment: In software, a measurable increase in functionality.

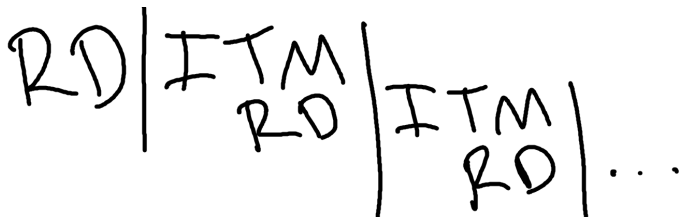
waterfall (software process model): Way of going about software development and management that is characterized by extensive planning, comprehensive documentation, and moving linearly through stages of the software development lifecycle (SDLC).

The **software development lifecycle (SDLC)** is the way a software project proceeds through the SDLC stages:

1. **Requirements**: Defining what the software must do, how well it must do what it will do, and under what limitations or constraints
2. **Design**: Defining how the code will be structured and how the user will experience the software
3. **Implementation**: Coding or otherwise converting the design into a product
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

There are different ways to travel through the SDLC stages. Patterns of travelling through the stages are called **software process models**.

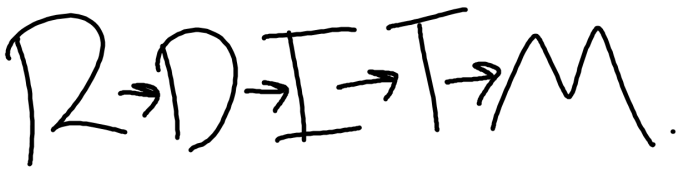
Commonly, people compare the Agile software process model with the Waterfall model. Agile, guided by the Agile Manifesto, moves through the SDLC approximately like this:



Vertical lines represent development cycle boundaries. Planning (R,D) for the next development cycle starts during the previous cycle.

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 frameworks for developing and managing software in an Agile way, such as Scrum, Extreme Programming (XP), and Kanban.

Waterfall moves through the SDLC approximately like this:



Movement in linear; Each stage must be completed before moving to the next, and turning back is not allowed (you can't swim up a waterfall)—unless the project is starting over. Lots of documentation is produced on the way. There are multiple variants of Waterfall, such as the V-Model, RAD, and the Royce model.

2.1.1 Why care about Agile, other software process models, and software engineering methods?

Some reasons:

- 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 the 2015 Standish Group CHAOS Report (International 2015), **17 to 22% of software projects fail**, 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 Agile-focused, the remainder of the chapter gives you a taste of the Agile software process model, one Agile framework (Scrum), and a few methods that are Agile but not specifically Scrum.

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

Some findings about software projects:

56% not on budget
60% not on time
44% not on target
43% of "grand" (largest) projects failed
7% of small projects failed

Full report: `https://tinyurl.com/chaos-report-2015`

2.2 Agile, Scrum, and Agile Methods

2.2.1 Agile

The Agile philosophy is summed up by the Agile Manifesto for Software Development:

> 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 methods for software or IT projects. For example, according to a 2017 survey by Hewlett Packard with 601 respondents, here is the distribution of what organizations use as their primary development method:

- 51%: Leaning toward Agile
- 46%: Hybrid
- 16%: Pure Agile
- 7%: Leaning toward Waterfall
- 2%: Pure Waterfall

Why do organizations choose Agile? According to HP:

Percent of respondents agreeing with statement about Agile development (respondents=403 organizations that have primarily adopted Agile):

- 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

2.2.2 Scrum

Scrum is a **set of methods** that align with the Agile philosophy. For example, the Scrum Guide (ever-evolving manual for Scrum) (Schwaber and 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. The Sprint Plan can be defined shortly before the Sprint; Teams (and their customers) might only know what they're doing for two weeks at a time.

Scrum gives teams **high-level** methods for carrying out a software development project. For example, it says nothing about how to code.

In the current version of the Scrum Guide, the methods are divided into three categories: the **team**, the **events**, and the **artifacts**. To give you a quick, convenient introduction to Scrum, the methods are listed below.

The Team

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

Method (Role)	**Definition** (Source: The Scrum Guide)
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

Method (Event)	**Definition**
Sprint	"fixed length events 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 ... **Scrum Team and stakeholders**"
Sprint Retrospective	"**to plan** ways to increase quality and effectiveness ... **Scrum Team**"

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

Method (Artifact)	**Definition**
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 Team is planning to do sometime, 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 and Sutherland 2020) describes the Scrum methods in more detail and defines some of the terms that were unexplained here (e.g., Sprint Goal).

2.2.3 Agile Methods

A few other Agile methods that aren't officially part of Scrum but are common and can be used with Scrum (or other frameworks, or other software process models):

Method	Description
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. Could be a physical bulletin board with sticky notes or index cards. 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.

2.3 Conclusion

"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 and Sutherland 2020), which changes frequently.

Additional Resources

Kent Beck (2000). *Extreme programming explained: embrace change*. addison-wesley professional

Hewlett Packard Enterprise (2017). "Agile is the new normal: Adopting Agile project management". In: *Hewlett Packard Enterprise Development LP*

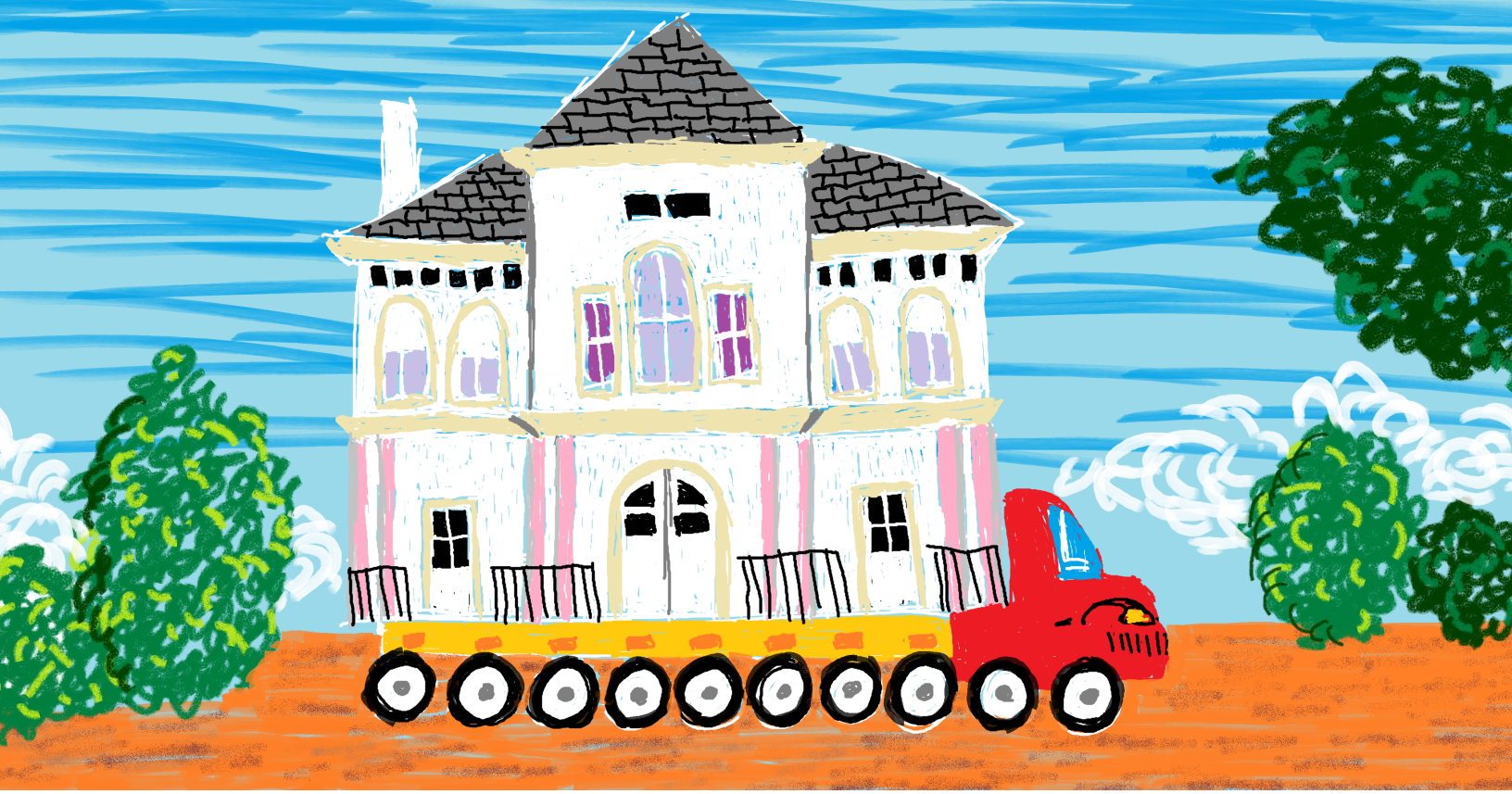
Extreme Programming: A Gentle Introduction (n.d.). `http://www.extremeprogramming.org/`. Accessed: 2021-01-01

Martin Fowler (2019a). "Agile Software Guide". In: URL: `https://web.archive.org/web/20210429215912/https://martinfowler.com/agile.html`

Winston W Royce (1987). "Managing the development of large software systems: concepts and techniques". In: *Proceedings of the 9th international conference on Software Engineering*, pp. 328–338

Ken Schwaber and Jeff Sutherland (Nov. 2020). "The Scrum Guide". In: *Scrum Alliance*

Standish Group International (2015). "The chaos report". In: *United States of America*. URL: https://web.archive.org/web/20210325103248/https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf



Chapter 3

Project Management & 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, or 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 select-

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

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.

ing suitable **tools**.

This chapter covers a variety of project management methods, including those related to **teamwork**. None of them are limited to just one type of software development environment but this chapter, like all of this book, is slanted toward Agile. There are many more methods that aren't discussed here; instead of hoping to be comprehensive, this chapter gives you a starter set of methods that are well known and highlight different areas of project management.

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

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.

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

3.1 Why learn about project management?

Since this book is aimed at people who want to become or are software engineers, why is there a chapter about project management? Reasons to learn project management:

- You might **become** a project manager (e.g., because your employer asks you to fill the role or you're interested).
- 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 (e.g., implications of the burn down chart analysis).
- You might need to **self-manage** (e.g., within an organization that has a flattened hierarchy or within an Agile team).

3.2 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:

- **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:

Other authors in other fields sometimes consider quality separate from constraint. In software engineering, requirements include quality.

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

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. Examples:

mitigation plan: What you will do if a contingency happens.

- 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**), might 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.

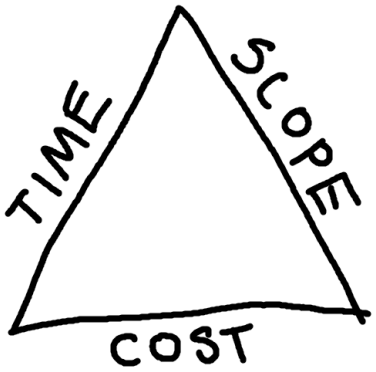
Making strategic project decisions involves adjusting project constraints. If you want to reduce time and cost spent on a project or increase project scope, you'll need a corresponding change in one or more other constraints. One way to visualize this:

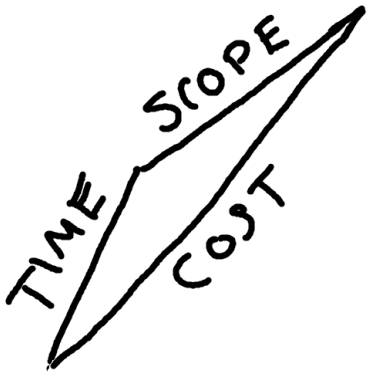
- Begin with an **equilateral triangle**. The three edges represent time, cost, and scope. Time and cost are already as small as possible. Scope is as large as possible, given the time and cost constraints.
- If you want the project to take less time (shorter time edge), you'll have to either increase the length of the cost edge, make the scope edge shorter, or do both. Likewise with adjusting the

other constraints.

- This model only goes so far. Don't, for example, get caught up with trying to keep the area or perimeter of the triangle constant.

The triple constraint triangle (a.k.a. project management triangle) is sometimes shown with each vertex labelled instead of each edge. However, that triangle isn't as useful for imagining the impact of your project decisions.





If you want your project to take less time, you might have to tolerate it costing more or having a reduced scope.

3.3 Managerial Skill Mix

managerial skill mix (MSM): Three categories of skills used by managers: (1) interpersonal, (2) technical, (3) administrative/conceptual.

What skills are required for managing a project? There are three broad categories comprising the **managerial skill mix (MSM)**:

- **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).

method: A pre-established way of achieving a specific outcome.

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. Since this chapter is about project management, we will focus more on interpersonal and technical skills.

3.4 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 while you read this section, consider **Tuckman's five stages of team development**:

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.

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, new team standards and roles, and team 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 will discuss specific methods a team can use to improve communication. Consider where each might fit in to these stages (there's not just one answer).

3.4.1 Establishing Ground Rules

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

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 here are questions teams can discuss to help:

- What is our **vision** for what this team is or what we're trying to accomplish together? (e.g., clients choose us because we're honest and transparent)
- What do we **prioritize** most? (e.g., delivering a high-quality product ahead of the deadline, input from all team members, honoring diverse end-users, making the big bucks, etc.)

. . . . . .

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 or the Agile Manifesto.

. . . . . .

If your team was to start with a single ground rule, what would be a good one? Maybe, "We agree to discuss adding more ground rules as needed."

- What methods will we use for **day-to-day communication**? (e.g., no interrupting, no 'splaining, listen to and acknowledge what other people are saying, ask people if they're busy before starting a long conversation, etc.)
- What methods will we use to **communicate** with each other **during conflict**? (e.g., we'll get trained on and use non-violent communication)
- What expectations do we have for **work habits**? (e.g., 1 to 3pm on Tuesday is silent time, be 5 minutes early to meetings, etc.)
- What expectations do we have for **responsiveness**? (e.g., respond within 2 hours during regular work hours and within 24 hours over the weekend, have the team Discord open during regular work hours, etc.)
- What will we do when team members **fail expectations**? (e.g., we'll discuss any team problems Friday at 3pm, etc.)
- How will we **get to know each other**? (e.g., we'll discuss each other's cognitive styles, we will not flirt with each other, we will have bring-your-pet-or-child to work days, etc.)

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, too authoritative, etc., that could invalidate your team's efforts toward creating the ground rules.

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.

minimum viable product (MVP): A low-effort or low-expense effort that results in you being able to better estimate whether people will want to use your product—before the product is fully developed.(Olsen 2015)

focus group (in usability engineering): A moderated discussion between researcher and a small number of potential users (usually 6-12) during which the researcher tries to gather information about the participants' attitudes, opinions, motivations, concerns, and problems related to a specific product or topic.(Odimegwu 2000)

3.4.2 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).

Basic example defining who should do what during the minimum viable product (MVP) development phase:

	Frontend Developers	Frontend Designers	Frontend Lead	Backend Developers	Backend Lead	Team Lead
Phase 1: MVP						
Focus groups	C	R	R / A	C	C	R / A
Requirements spec.	R	R	A / I	R	A / I	C
Throwaway code design			I	R	A	I
Implementation	R	C	A	R	A	C
User acceptance testing	R	R	R / A	R	C	C

Interpreting a RACI matrix:

- Top row: Roles. One person might have multiple roles.
- First column: Tasks or deliverables, organized into phases (if needed).
- Letters define what role is responsible for which task or deliverable.
- **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)**: Who 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).

3.4.3 Measuring and Building Consensus: Fist of Five Method

fist of five: A method for gauging and building group consensus that uses a 6-level voting system (zero to five fingers).

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 I've also seen examples of it being used with students of different ages (e.g.,

(Fletcher 2002), (Hulshult and 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 resolve later.
- **Four**: Accept. No issues.
- **Five**: Strong accept. Willing to lead or champion.

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

If anyone suggests rejecting the statement or idea by holding up two or fewer fingers, the team can stop, discuss, make changes, and re-vote 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.

3.5 Technical Skills: Project Definition

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

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

3.5.1 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.

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

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

Sprint Backlog: In 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.

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.

3.5.2 Balancing Constraints: Project Priority Matrix

Earlier, 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**:

	Time	Cost	Scope
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

For **example**, if 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, your project priority matrix might look like this:

	Time	Cost	Scope
Constrain			✓
Enhance		✓	
Accept	✓		

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 taxpayers. **Time**: While hopefully the project stays on track and delivers as promised, if needed your team can submit intermediate results to the NIH and (hopefully) 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.

project priority matrix: 3x3 grid for documenting how to respond when there are potential changes to a project's time, cost, or scope. Options: Only positive change allowed (constrain), negative change allowed (accept), or positive change sought (enhance).

"I have two kinds of problems, the urgent and the important. The urgent are not important, and the important are never urgent." – Dwight D. Eisenhower

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

Scrum: An Agile framework "for developing and sustaining complex products." (Schwaber and Sutherland 2020)

3.5.3 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 (i.e., someone representing the customer, like the **client**).

But how are task priorities decided? One high-level method is called the **Eisenhower matrix**:

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

Eisenhower matrix: 2x2 grid for helping decide whether to do, delegate, schedule, or eliminate a task based on its urgency and importance.

- **Do** (urgent, important): Needs to be done correctly and now. **Example**: Documenting your undocumented code so that a new hire can start contributing.
- **Decide** (not urgent, important): Needs to be done correctly but not immediately. **Example**: Refactoring your currently-working code. 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, etc.). **Example**: Someone needs 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. 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. **Example**: 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 them **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 non-urgent tasks getting eternally left at the end of the to-do list (perhaps resulting in project failure).

3.5.4 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? **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 past experience which tasks have more unknowns, more risk, dependencies, etc.

- If it's an implementation task and you're meant to be an expert, you can do a **focused research effort called a spike**, usability testing 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) Focus on answering the question, discovering additional questions and sub-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 write a test suite. You have experience writing a test suite 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 strongly 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 is waiting on you? How many other tasks depend on this task? Compare that to the important of the dependent tasks (or the importance of keeping the waiting person happy / productive) and how long it'll take to complete the 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 4-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 sub-optimal choice.

- If you're deciding which feature to implement, you can **ask the customer or users** directly (e.g., through a phone call, focus groups, etc.) or indirectly (e.g., by looking at support tickets, asking the marketing team, detecting an unmet need based on how people use other software, etc.).

- Other ways to select features: **Voting** (e.g., within your team) or **pairwise comparison** (e.g., Is Feature A more valuable than

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

focus group (in usability engineering): A moderated discussion between researcher and a small number of potential users (usually 6-12) during which the researcher tries to gather information about the participants' attitudes, opinions, motivations, concerns, and problems related to a specific product or topic.(Odimegwu 2000)

usability testing: Observing people while they try to use your software.(Barnum 2020)

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

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

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.

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

Intertwined with prioritization is **estimation**: Figuring out ahead of time how long a task is likely to take. But what does "how long" mean and how do we figure out "how long"?

Two methods, from the Agile community, of **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, etc. Implementing a major feature might, on the other hand, 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, no distractions, etc. For example, if it takes me 5 minutes to remove one 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 work days are eight or nine hours).

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", etc. Work completed (in story points or ideal days) is, in Agile teams, called the **velocity**. Teams can make initial estimates about velocity 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**. 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 / etc. 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

Common scales for story points: 1 to 10, Fibonacci, and powers of two. 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.

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

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

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.

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

are revealed. Variations in estimates are expected, and part of the process: differences open a discussion. Someone making a high estimate might, for example, have thought of good reasons why a task is likely to take a long time. Someone making a low estimate may have identified an efficient idea nobody else thought of. The team discusses and, once ready, can repeat the process until estimates become sufficiently consistent.

3.5.6 Scheduling: Project Network

Once a set of tasks has been defined, prioritized, and estimated, those tasks can be scheduled. **Scheduling** a task is placing it within the timeline 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**. A project network is a directed graph showing a project's tasks, the sequence in which they're to 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 moves left to right, where left is earlier in time.

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

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

Constructing a Project Network

Project networks can be created manually or automatically generated by software. If you want to include estimates in the project network, generating the network will likely be less cumbersome, especially since you might want to modify your tasks or estimates once you see how the network looks. If you don't care about entering estimates and just want to visualize the sequencing and dependency relationships between tasks, drawing the network by hand might be sufficient for your needs.

For automatically generating a project network using software (e.g., MS Project, Lucidchart), you'd use the software's user interface to enter the task details. For example, in a table:

This textbook does not cover strategies or methods for optimally assigning personnel or other resources to tasks.

While complex project networks may be less valued in a Agile development environment, they might also be just the method you need for understanding a complex project.

In Agile, predecessors are also called **blockers** or **impediments**, especially in cases when an activity could be started but is waiting on another activity (or external event) to occur.

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

Note that, 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, you might have access to more complex options like specific dates by which individual tasks must be completed.

3.5.7 Task Management Systems

task management system: Software for planning and organizing project activities.

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

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

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

- 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 timeline (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 universally have a way to generate project networks. For that, you might need a fully-featured **project management system** (e.g, MS Project). However, you may find that a **Gantt chart** or roadmap feature meets your needs and is available within your task management system.

3.6 Conclusion

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.

3.7 Additional Resources

Michael K Badawy (1995). *Developing managerial skills in engineers and scientists: Succeeding as a technical manager*. John Wiley & Sons

Kevin Brennan et al. (2009). *A Guide to the Business Analysis Body of Knowledger*. Iiba

Karen A Brown, Nancy Lea Hyer, and Richard Ettenson (2013). "The question every project team should answer". In: *MIT Sloan Management Review* 55.1, p. 49

Shawn Belling (2020). "Agile Values and Practices". In: *Succeeding with Agile Hybrids*. Springer, pp. 47–61

Mike Cohn (2005). *Agile estimating and planning*. Pearson Education

Gayle Cotton (2013). "Gestures to avoid in cross-cultural business: In other words,'Keep your fingers to yourself!'" In: *The Huffington Post. Avaiable at:< http://www. huffingtonpost. com/gayle-cotton/cross-cultural-gestures_b_3437653. html>(retrieved July 7, 2017)*

A Fletcher (2002). "FireStarter youth power curriculum: Participant guidebook". In: *Olympia, WA: Freechild Project*

Jarett Hailes (2014). *Business Analysis Based on BABOK® Guide Version 2–A Pocket Guide*. Van Haren

Brian Hambling and Pauline Van Goethem (2013). "User acceptance testing: a step-by-step guide". In: BCS

Andrea R Hulshult and Timothy C Krehbiel (2019). "Using Eight Agile Practices in an Online Course to Improve Student Learning and Team Project Quality." In: *Journal of Higher Education Theory & Practice* 19.3

J Mike Jacka and Paulette J Keller (2009). *Business process mapping: improving customer satisfaction*. John Wiley & Sons

Erik Larson and Clifford Gray (2018). *Project management: The managerial process*. Irwin/McGraw-Hill

Lucid (n.d.). *What is Fist to Five?* `https://www.lucidmeetings.com/glossary/fist-five`. Accessed: 2021-01-01

Viljan Mahnič and Tomaž Hovelja (2012). "On using planning poker for estimating user stories". In: *Journal of Systems and Software* 85.9, pp. 2086–2095

Debbie Thorne McAlister (2006). "The project management plan: Improving team process and performance". In: *Marketing Education Review* 16.1, pp. 97–103

Microsoft (n.d.). *The project triangle*. `https://support.microsoft.com/en-us/office/the-project-triangle-8c892e06-d761-4d40-8e1f-17b33fdcf810`. Accessed: 2021-01-01

Barry Overeem (2016). *Characteristics of a Great Scrum Team*

Andy Stuart (2014). "Ground rules for a high performing team". In: *Paper presented at PMI® Global Congress 2014—North America, Phoenix, AZ. Newtown Square, PA: Project Management Institute.* Pp. 328–338

Bruce W Tuckman (1965). "Developmental sequence in small groups." In: *Psychological bulletin* 63.6, p. 384

Bruce W Tuckman and Mary Ann C Jensen (1977). "Stages of small-group development revisited". In: *Group & Organization Studies* 2.4, pp. 419–427

Jasim MohJasim Mohamed Lahdan Fhadel Al Qubaisi et al. (2015). "Leadership, culture and team communication: analysis of project success causality-a UAE case". In: *International Journal of Applied Management Science* 7.3, pp. 223–243

Muhammad Usman et al. (2014). "Effort estimation in agile software development: a systematic literature review". In: *Proceedings of the 10th international conference on predictive models in software engineering*, pp. 82–91

C Jurie Van Wyngaard, Jan-Harm C Pretorius, and Leon Pretorius (2012). "Theory of the triple constraint—A conceptual review". In: *2012 IEEE International Conference on Industrial Engineering and Engineering Management*. IEEE, pp. 1991–1997

Li-Ren Yang, Chung-Fah Huang, and Kun-Shan Wu (2011). "The association among project manager's leadership style, teamwork and project success". In: *International journal of project management* 29.3, pp. 258–267

Chapter 4

Requirements

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

4.1 Types of Requirements

There are **two types of requirements**:

1. **Non-functional requirements** specify qualities the software should have (e.g., usable, portable, modular, etc.). They answer the questions, "How well should the software perform?" and "What limits or constraints is the software subject to?" This chapter includes a discussion of how **quality attributes** can be used in specifying non-functional requirements.

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

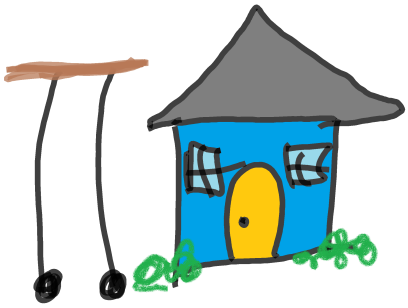
non-functional requirement: Description of how well software is expected to perform.

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

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

2. **Functional requirements** specify the desired functionality of software (e.g., if I click the Log In button, the Login page appears). They answer the question, "What should the software do?" In this chapter, we'll talk about specifying functional requirements with user stories and use cases.

User stories and use cases are two different methods for specifying functional requirements. As you will see later, one is more formal than the other.



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

4.2 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:

Requirements keep the development team on track and working together toward creating what the client (and hopefully the users) want.

- 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** with one another and **have the same goal**. Without requirements, time, effort, and money can be wasted implementing conflicting code.

Requirements can help protect projects from drift and failure.

- When requirements aren't specified, it's easier for project stakeholders (e.g., **clients**, partners, investors, consultants, management, etc.) 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).

client (a.k.a. customer): 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).

4.3 What Makes a Good Requirement

Good requirements have the following characteristics:

Correct	What they say is right.
Consistent	They aren't contradictory of each other.
Unambiguous	There is only one way to interpret them.
Complete	They cover all that's important.
Relevant	They meet a stakeholder need.
Testable	There's a way to figure out if they're satisfied.
Traceable	It's possible to figure out where they came from.

Sloppy requirements can be useless or worse.

Requirements that fail to have these characteristics can lead developers to making features of software nobody wants, wasting time and other resources and potentially jeopardizing the project.

stakeholder: Anyone who is or will be affected by the software or its development (e.g., clients, companies, users, developers, managers, politicians, etc.)

4.4 Requirements Elicitation

requirements elicitation: The process of gathering requirements from project stakeholders.

The process of gathering requirements is called **requirements elicitation**. Requirements can come from any stakeholder, including clients, managers, users, governments, developers of software your software will integrate with, your development team, and yourself.

Three of the most important, distinct, and universal (common across projects) categories of stakeholders:

- **Clients**: The people who request the software and have most of the authority over its requirements (e.g., because they are paying for it).
- **Users**: The people who will use the software.
- **Developers**: The people who will make the software, including those who manage the software engineers.

Aspects of these stakeholders that can affect the requirements elicitation processes and the software's development and ultimate success:

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.

focus group (in usability engineering): A moderated discussion between researcher and a small number of potential users (usually 6-12) during which the researcher tries to gather information about the participants' attitudes, opinions, motivations, concerns, and problems related to a specific product or topic.(Odimegwu 2000)

usability testing: Observing people while they try to use your software.(Barnum 2020)

minimum viable product (MVP): A low-effort or low-expense effort that results in you being able to better estimate whether people will want to use your product—before the product is fully developed.(Olsen 2015)

- **Clients might not have experience or expertise**. Developers can help fill the gap between what the client wants and what is technically feasible and reasonable (e.g., given time, cost, and scope, a.k.a. the **triple constraint**).

- **Clients might not have good ideas**. They may be incorrect about what users will want or will use. Developers sometimes try to guide clients 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.

- **Clients might not know what they want**. They might have a rough idea, or they might have an idea that's at odds with their goals. Developers, through requirements elicitation, can help clients define their goals clearly and reasonable ways for accomplishing those goals.

- **Users might not know what they want or will use**. They may be unaware of their own needs or wants until there's a product in front of them that addresses those needs or wants. Even if 10,000 users tell you, "I would definitely use an app that does X", they might be wrong, they might only use the app once, or they might not be willing to pay for the app. MVP can be a good method for figuring out early whether users will be interested enough in the software to use or pay for it.

- **Users might want what's bad for them**. You can probably think of multiple examples.

- **Developers have their own tendencies**. They may have technologies and ideas they prefer or feel most comfortable with. For better or worse, they bring their own influences to a project.

- **Clients, users, and developers are all humans**. They communicate imperfectly.

Deciding what software to make, and doing so successfully, is a complex process influenced by human factors affecting all involved.

So how does one elicit requirements? By having conversations or otherwise collecting information from stakeholders. The amount of stakeholder communication can vary by project, project type, the software process model being used, and other factors.

4.5 Non-Functional Requirements

non-functional requirement: Description of how well software is expected to perform.

Non-functional requirements describe how well the software needs to perform.

Examples of non-functional requirements:

- Response time should be a few seconds or less in all operating environments.
- The keylogger must be indetectable to 99.999% of test users.
- The software must be available 24 hours a day, 7 days a week, and must have an uptime of 99.99%.

Notice that each requirement has a **quantity associated** with it: That makes it testable (a criterion for a good requirement).

4.5.1 Quality Attributes

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

Quality attributes are characteristics of software used to describe how good it is. They can be used in specifying non-functional requirements.

Examples of quality attributes:

- **Reliability**: How often does function X succeed?
- **Efficiency**: How many resources does the software need?
- **Integrity**: How frequently does the software have errors that require a restart?
- **Memorability**: How many times must users learn a function before they no longer need documentation?
- **Flexibility**: How many ways can the software be used?

- **Interoperability**: How well can the software integrate with other software?
- **Reusability**: To what extent can the code be used to solve other problems without being modified?

Each quality attribute can be converted to a scale. For example, the lowest value on a reliability scale 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 non-functional requirement by defining a performance threshold:

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

A quality attribute is **not the same as a non-functional requirement**. Rather, quality attributes are good for labelling what a non-functional requirement is about.

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 non-functional requirements) in mind for the duration of the project; If the software is not meeting the non-functional requirements, either the software or the threshold of acceptability needs to change.

4.6 Functional Requirements

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

Example of a functional requirement:

> When a user clicks the "register" button, their information is added to the database and the user is shown a "thank you for registering" screen.

4.6.1 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 non-technical people (e.g., users, clients, other stakeholders) can understand them.

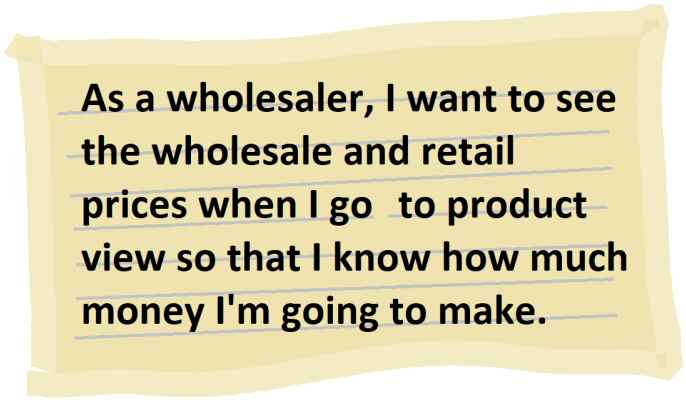
user story: "Short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system." (Cohn n.d.)

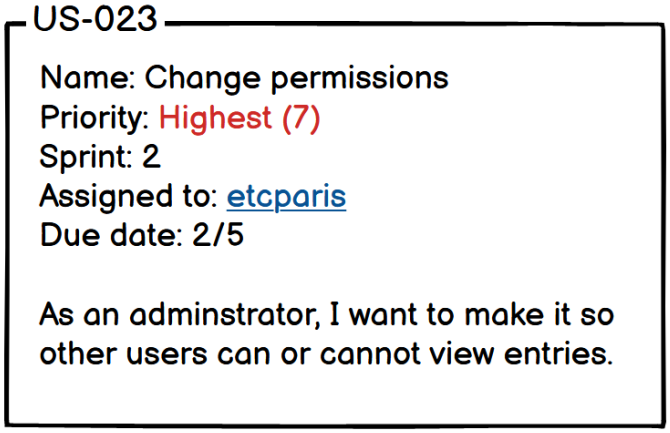
User stories have a title and are commonly written using this format:

> As a ⟨ROLE⟩, I want ⟨SOME FUNCTIONALITY⟩ so that I get ⟨SOME BENEFIT⟩.

These short sentences can be written on 3x5" 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, etc.).

Examples of what user story cards can look like:





More examples of user stories: `https://tinyurl.com/user-story-examples` `https://twitter.com/shituserstory`

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.

Characteristics of good user stories (INVEST):

INVEST: Characteristics of good user stories (independent, negotiable, valuable, estimable, small, testable) (Wake 2003).

I	**Independent**: Doesn't depend on other user stories.
N	**Negotiable**: Can be changed during development.
V	**Valuable**: Fulfills a user need.
E	**Estimable**: Can be given a time estimate.
S	**Small**: Can fit into a single development period (e.g., a 2-week Sprint)
T	**Testable**: Possible to determine it's done.

acceptance criterion: A statement about functionality that, when satisfied, mean the functionality has been satisfactorily implemented.

Definition of Done (DoD): The set of acceptance criteria which, once satisfied, mean a user story has been satisfactorily implemented.

There is some overlap between INVEST and the general characteristics of good requirements mentioned earlier in this chapter.

How do you know when you are done with a user story? 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 in order for the user story to be considered done (i.e., establishing the **Definition of Done** for the user story).

Example acceptance criterion:

> **Given** the user is playing a video file **and** their operating system is Windows, **when** they do the Ctrl-T keyboard shortcut **then** they will see the "Go to Time" screen **and** the video will pause.

There are bolded words in that example because its using a common format (Alliance n.d.) for acceptance criteria:

> Given ... when ... then ...

The "and"'s are optional parts of the format. Ideally, acceptance criteria testing can be automated.

Example pseudocode for testing acceptance criteria:

```
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)
```

4.6.2 Use Cases

use case: "A contract for the behavior of the system under discussion" (Cockburn 2001)

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 with it.

Use cases are not specific to a particular software process model (e.g., Agile, Waterfall, Spiral) or environment. Instead, like much of what you will encounter in this book, they are a well-known method software teams can choose to use (and many do), or not.

More examples of use cases: `https://tinyurl.com/use-case-examples`

Example of a use case:

- **Name**: Generate list of recovered patients
- **Actor**: Clinician
- **Flow**:
  1. Clinican authenticates using smart card
  2. Software confirms credentials and access 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

Required Parts of a Use Case

What every valid use case has:

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

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).

The correct amount of detail to give a use case is the minimum amount to adequately describe what you're trying to communicate.

Additional Parts of a Use Case

Sometimes included in use cases:

- **Identifier**: A unique way of referring to the use case (e.g., UC-002)
- **Pre-conditions**: What must be true before the flow (e.g., The shopper has added at least one product to their shopping cart.)
- **Post-conditions**: 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. This unique identifier is handy for this part.
- **Extensions**: Contingencies, alternate routes, and branches to other use cases
- **Priorities**: The importance of the use case
- **Non-functional requirements**: How well the software must perform during the flow

4.7 Requirements Specification

requirements specification: Converting stakeholder requests into written requirements.

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 is also called an **SRS** (Software Requirements Specification). The best way to understand what an SRS looks like is to look at some.

Software Requirements Specification (SRS): A document that contains software requirements.

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

- **SRS for apps and a data repository for distributing manufacturing data**: Thomas Hedberg Jr., Moneer Helu, and Marcus Newrock (Dec. 2017). *Software Requirements Specification to Distribute Manufacturing Data*. `https://web.archive.org/web/20201208070659/https://nvlpubs.nist.gov/nistpubs/ams/NIST.AMS.300-2.pdf`

Another type of software document, which sometimes gets confused with an SRS, is a Software Design Document (SDD). If the SRS is what the software *should* do, the SDD is what the software *is*. However, there is often overlap between the two.

- **SRS for data system that assesses conservation practices**: Data System Team (n.d.). *System Requirements Specification for STEWARDS*. https://web.archive.org/web/20200923200038/https://www.nrcs.usda.gov/Internet/FSE_DOCUMENTS/nrcs143_013173.pdf

- **SRS for an app that splits and merges PDFs**: Ploutarchos Spyridonos (Feb. 2010). *Software Requirements Specification for PDF Split and Merge, Version 2.1.0*. https://web.archive.org/web/20170225043950/http://selab.netlab.uky.edu/%7Eashlee/cs617/project2/PDFSam.pdf

- **SRS for software that processes EEG data**: Inria Innovation Lab (n.d.). *Software Requirement Specification for CertiViBE, v1.0*. https://web.archive.org/web/20190710221933/http://openvibe.inria.fr/openvibe/wp-content/uploads/2018/04/CERT-Software-Requirement-Specification.pdf

- **SRS for library software**: Fred Eaker (Nov. 2006). *Software Requirements Specification for Vyasa*. https://web.archive.org/web/20161127184329/http://vyasa.sourceforge.net/vyasa_software_requirements_specification.pdf

4.8 Conclusion

Gathering and writing down requirements for a project can help with keeping the project on track and communicating about the project to others. Doing requirements well can save a project from failing.

4.9 Additional Resources

Agile Alliance (n.d.). *What is "Given - When - Then?"* https://web.archive.org/web/20201124202211/https://www.agilealliance.org/glossary/gwt

Roger Atkinson (1999). "Project management: cost, time and quality, two best guesses and a phenomenon, its time to accept other success criteria". eng. In: *International journal of project management* 17.6, pp. 337–342. ISSN: 0263-7863

Carol M. Barnum (2020). *Usability Testing Essentials: Ready, Set...Test!* 2nd ed. Morgan Kaufmann

Alistair Cockburn (2001). *Writing effective use cases*. Boston

Mike Cohn (n.d.). *User Stories and User Story Examples*. https://web.archive.org/web/20201124004807/https://www.mountaingoatsoftware.com/agile/user-stories

Martin Fowler (2004). *UML distilled : a brief guide to the standard object modeling language*. Boston

Clifford Odimegwu (July 2000). "Methodological Issues in the Use of Focus Group Discussion as a Data Collection Tool". In: *Journal of Social Sciences* 4, pp. 207–212. DOI: 10.1080/09718923.2000.11892269

Dan Olsen (2015). *The lean product playbook : how to innovate with minimum viable products and rapid customer feedback*. Hoboken: Wiley. ISBN: 9781118961025

Rebecca Parsons (June 2003). "Components and the world of chaos". In: *Software, IEEE* 20, pp. 83–85. DOI: 10.1109/MS.2003.1196326

Bill Wake (Aug. 2003). *INVEST in Good Stories, and SMART Tasks*. https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/. Accessed: 2020-12-31

Chapter 5

Unified Modeling Language (UML) Class and Sequence Diagrams

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

"Nobody, not even the creators of the UML, understand or use all of it."

Martin Fowler
UML Distilled (3rd Ed.)

5.1 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, etc.).

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.

class diagram: 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.

sequence diagram: Interaction diagram showing how different participants (e.g., users, software components, classes, etc.) collaborate during a single use case.

Audiences often have short attention spans.

IDE: Integrated development environment. Software specifically for creating software.

Example audiences for your diagrams: 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 **IDE**/tools you're using, diagrams can be automatically generated from your code, which helps make documentation maintenance easier and more likely to happen.

5.2 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.

5.3 What is UML?

UML: Unified modeling language: A set of notation and methods for describing and designing software.

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 have 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.

5.4 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 more tricky 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 common vocabulary for communicating about software. That being said, expect to encounter some variation in how UML notation is used—it can be difficult to remember all the details of UML notations; many developers will make mistakes or adapt the notation to their own way of thinking. That is ok to do so long as you 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.

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. However, the generated diagrams can sometimes have more detail than you want, making them less good for communicating.

5.5 Why NOT use UML?

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

- People tend to **vary their UML notation**, which can cause **confusion**. Tips for avoiding that problem: (1) Keep your notation basic and (2) explain more complex notation usage to the people you're trying to communicate with.

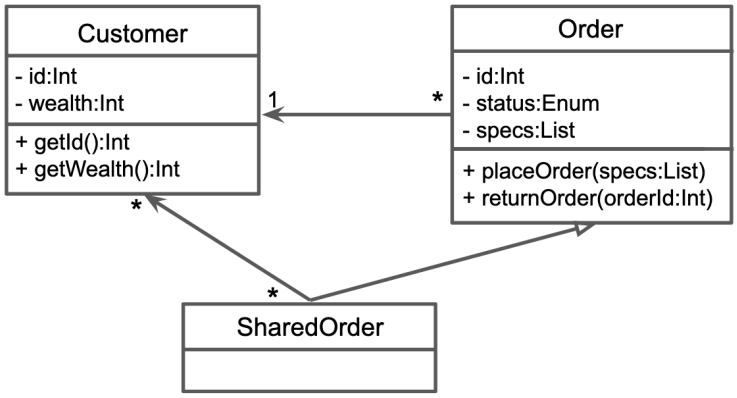
- Trying to get the UML notation details right **can take a lot of time**. Remember that diagrams are for communicating; If creating the diagram takes longer than explaining the code a different way, the diagram isn't helping.

- 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.

5.6 Class Diagrams

A **class diagram** describes types of objects in a system and the static relationships that exist among them. Class diagrams also show properties and operations of a class and constraints on how objects are connected. UML uses the term "feature" as a general term that covers properties and operations of a class.

Example class diagram:

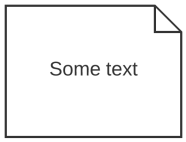
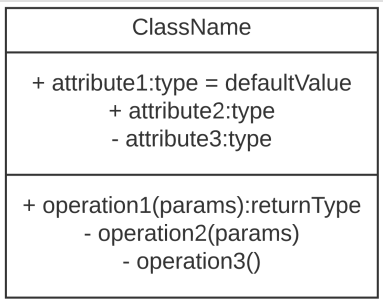
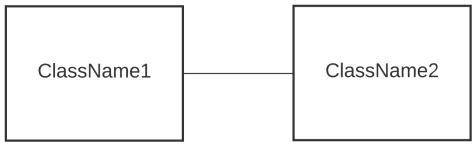
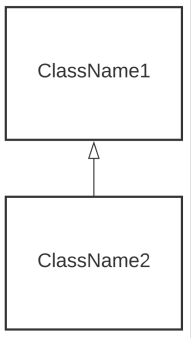
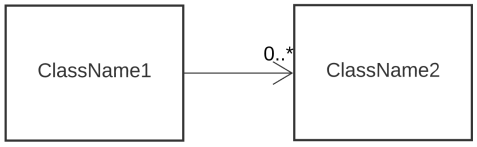


This class diagram shows the relationships between three classes: Customer, Order, and SharedOrder. An Order has one Customer—but the same Customer can be on multiple Orders. A SharedOrder is a type of Order that can have multiple Customers. The classes have "attributes" (e.g., id) and "operations" (e.g.,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

The next page explains each of the notational elements shown in the example. Class diagram notations gets more complicated than is described here; see publications in Additional Resources.

5.6.1 UML Class Diagram Notation

Below is a subset of UML class diagram notation. Some of the other notation tends to be confusing and so more people get it wrong (leading to miscommunication). However, if you'd like to learn about it anyway, see the references section at the end of this chapter.

Graphical Representation	Name	Description
	note	A note. Notes are for putting comments on diagrams.
	class	A class, potentially with attributes and operations (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.
	association	Association means that a class contains a reference to an object(s) of the other class in the form of an attribute. If Class1 points to Class2, Class1 *has a* Class2.
	inheritance	Inheritance means that one class is a subclass of another. If Class2 points to Class1, Class2 *is a* Class1.
	multiplicity	Multiplicity constrains the number of objects. If, for example, Class1 has three objects of type Class2, that's indicated with a 3 near the arrow pointing to Class2. 0..* (or just *) means zero or more. An integer N (e.g., 1) means exactly N. N..M means N to M (inclusive).

5.7 Sequence Diagrams

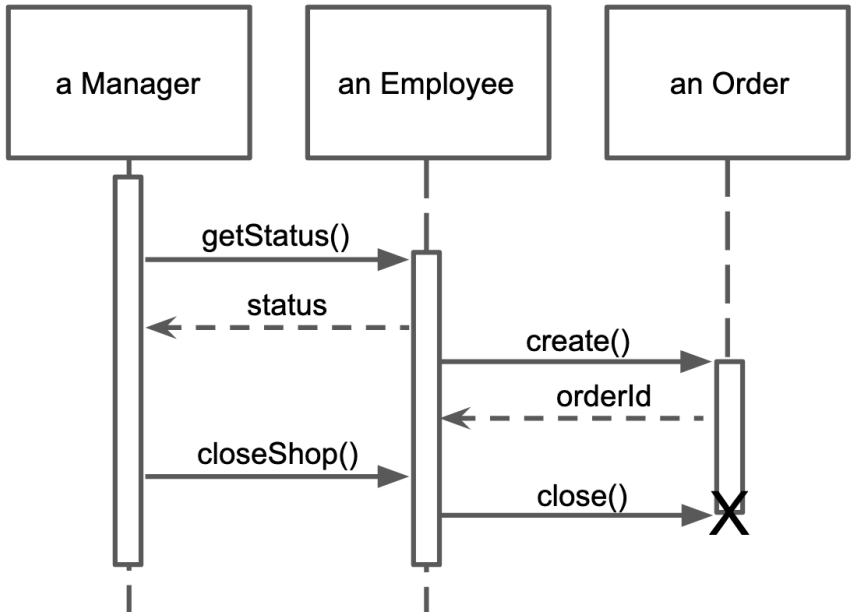
sequence diagram: Interaction diagram showing how different participants (e.g., users, software components, classes, etc.) collaborate during a single use case.

interaction diagram: Visualization of collaboration between different parts of software.

A **sequence diagram** describes interactions between objects. Usually, the diagram is showing 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 shows an example of a sequence diagram and commonly-used sequence diagram notation. To see more obscure notation, check the publications in the Additional Resources section.

Example sequence diagram:

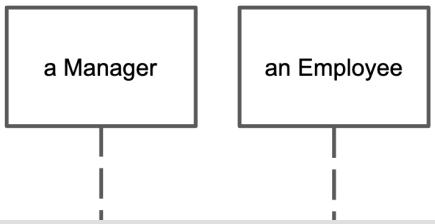
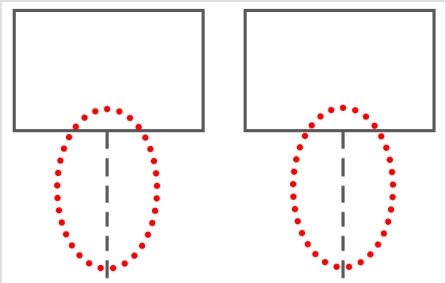
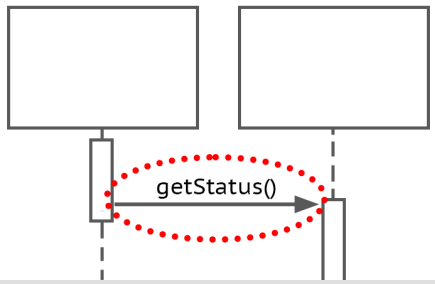
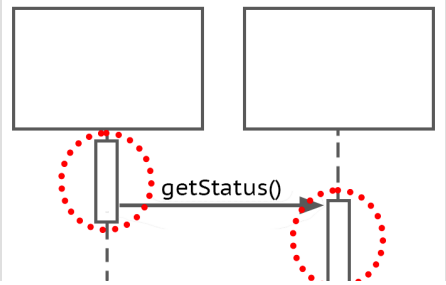
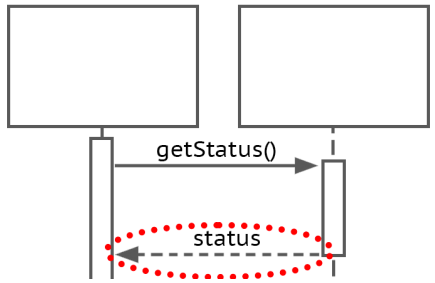


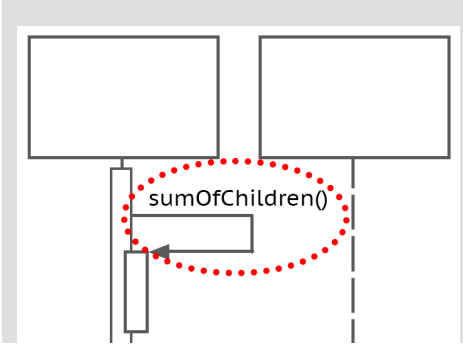
When making any diagram, know your audience and what you're trying to communicate. If your audience is a human, they have limited capacity for absorbing tiny details (and probably limited time). Focus on showing them what's most important in a way they will understand.

This sequence diagram shows 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.

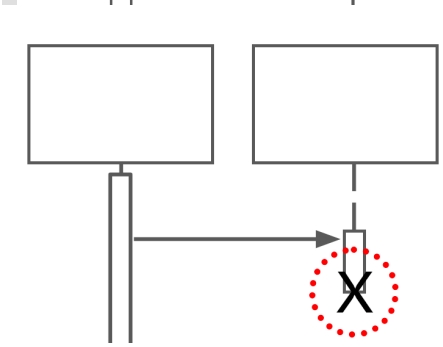
In the example, each of the columns (called "participants") are objects, but this is not always the case. For example, a participant can be a user. Users, if they are human, are sometimes represented as stick figures (without the box). Another possible non-object participant could be a database (although, 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.

5.7.1 UML Sequence Diagram Notation

Graphical Representation	Name	Description
	participant	The "columns" of a sequence diagram. Often objects. Name of the participant goes in the box.
	lifeline	Vertical dashed line representing the lifespan of the participant. Top is beginning of life, bottom is end. Life ends when the participant is deleted.
	message	Interaction from one participant to another. Solid line with arrow. Often a method call.
	activation	Box on lifeline indicating when the participant is active. Indicates method is on call stack.
	return	Dashed line with arrow indicating method return. Use only when it helps communicate something important about the interaction.



self-call — Method calling self. Solid line with arrow pointing back to participant's own lifeline.



deletion — End of participant's life. Indicated by "X" on lifeline.

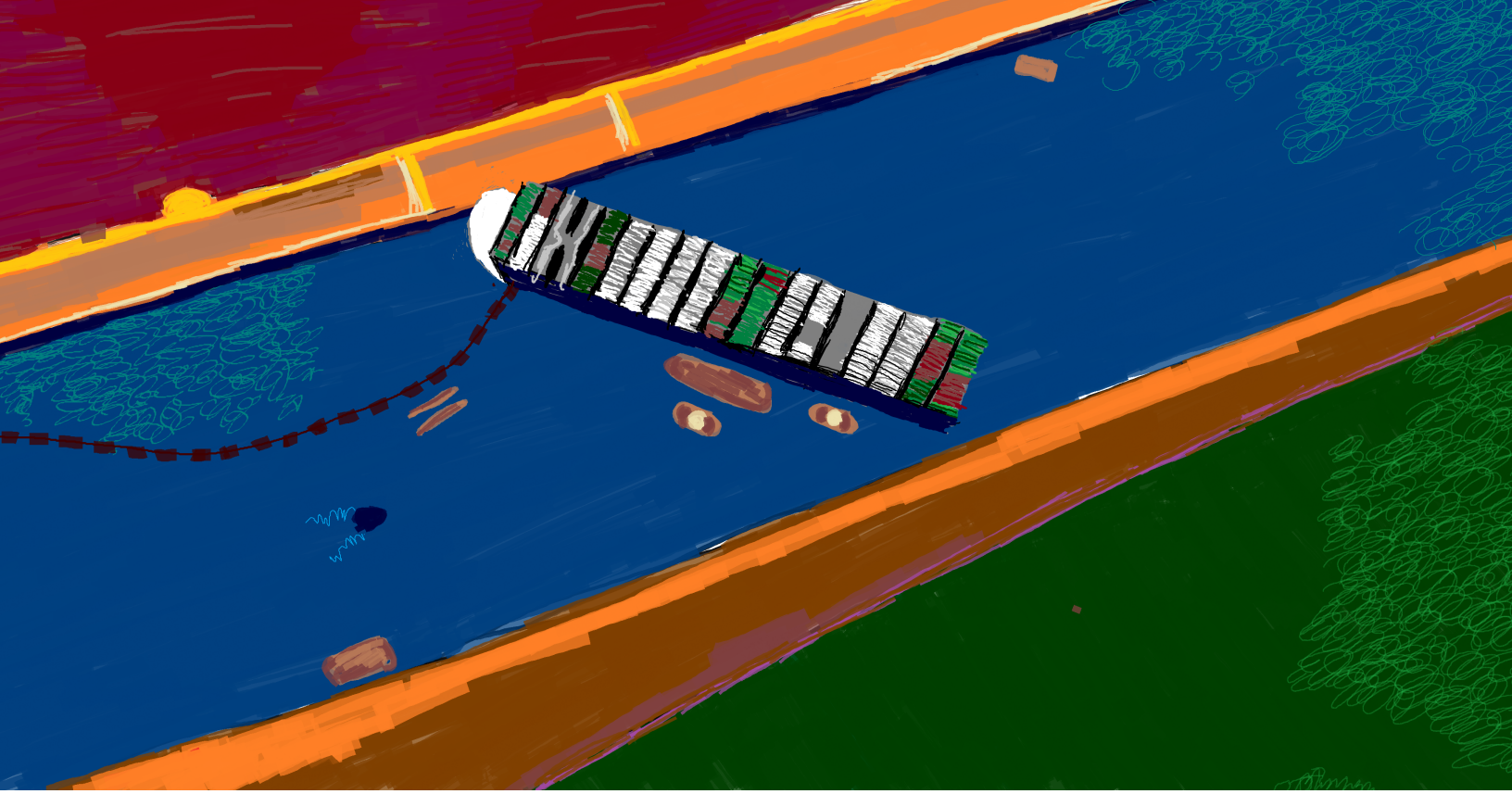
5.8 Conclusion

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

5.9 Additional Resources

Russ Miles and Kim Hamilton (2006). *Learning UML 2.0: a pragmatic introduction to UML*. O'Reilly Media, Inc.

Martin Fowler, Kendall Scott, et al. (2003). *UML distilled: a brief guide to the standard object*



Chapter 6

Monolith vs. Microservices Architectures

High-level architecture is the software's all-encompassing code design. When described with the 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, architecture can refer to code design at lower levels).

If you're developing new software, you might get to choose the high-level architecture, or it may already be baked into a framework you've chosen. For example, many web application frameworks use

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

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

the Model-View-Controller (MVC) architecture or variants. In the latter case, you have to learn MVC and how to work within it; The design decision is made for you.

Since my aim is to help you make choices about software, 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 architectures in general.

monolith architecture: Overall code design characterized by being in one or few pieces; cannot be easily divided into components that run separately and are independently useful.

microservices architecture: Overall code design characterized by multiple independent components that each run in their own process and communicate between one another without direct access.

abstraction: Representation that is purposely missing details to focus attention on purpose of the object / idea / etc. being represented.

6.1 Monolith Architecture

Monolith software is one or few pieces and **cannot easily be divided** into multiple independent components that run separately and are individually useful. What about when the client side, the server side, and the database are all separate, can that be a monolith? Yes. If the client-side part of the software will not start or is not useful without the database or server-side part of the software, that's a monolith.

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; 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 make a different design decision.

6.2 Microservice Architecture

Microservices are **multiple pieces** of software, each of which runs in a **separate process** and can be **individually useful**. This section describes core characteristics of software that uses the microservice architecture. Additional commentary from Lewis and Fowler at (Fowler and J. Lewis 2019).

6.2.1 "Smart endpoints and dumb pipes"

"Dumb pipes" does not imply simple message contents.

The **communication pipe** within microservice architectures is **simple** and the services themselves take care of translating and otherwise processing messages. For example, microservices commonly communicate through a REST API, which allow 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.

6.2.2 "Componentization via services"

In a microservice architecture, **components are services**. The Lewis and Fowler 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. In a monolith, it's more common to have more **tightly coupled** code and components that run in the same process.

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

service: A unit of software that received and fulfills requests.

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

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

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

Advantages of splitting components into services:

- **Independence**: Each individual service can be updated, tested, launched, and stopped without requiring the same from other parts of the software. In contrast, with some monolithic software, for example, 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**: Whereas in a monolith communication between components can be direct calls (fast, lightweight), with microservices **requests** often happen **over a network**, need to include metadata to explain the request, and, because the pipes are "dumb", responses can contain extra data the requester didn't ask for (slower, heavier).

- **Potentially less secure communication**: Communication over a network can be more prone to interception and alteration.

6.2.3 "Organized around business capabilities"

You may have heard of the **client-server high-level architecture**, which usually consists of multiple instances of client-side software that communicate with server-side software, which communicates

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

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 integrated definition 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" (Michell 2011).

business capability: "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" (Michell 2011)

. . . . . .

In each of these examples, what are the business resources producing customer value? What is the value to the customer? How are the business resources acting on their environment? What other resources are the business resources using?

. . . . . .

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.

. . . . . .

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

Examples of business capabilities:

- The manufacturer can slice a 20ft by 40ft rectangle of wheat dough into 0.5cm 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 convert a video so it works better on mobile devices.

One implication of being focused on business capabilties is that each microservice has its own **tech stack** (including its own database).

6.2.4 "Decentralized data management"

In a **microservice** architecture, each service **typically has its own database** instead of sharing a centralized database. This helps keep the microservices independent, which has many benefits including **failure containment**. A disadvantage is that interoperating microservices can end up with copies of the same data that are inconsisent (e.g., because one database has not yet received the update). The term for this is **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).

6.2.5 "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, etc. If, later, the team needs to change to different technologies, only the one service is affected. On the other hand, 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.

6.2.6 "Design for failure"

When services running in different processes on different machines and potentially being written with different technologies by different teams with different standards, that **can change how developers think**. Instead of keeping the whole ship afloat, thinking can shift toward what to do if a service fails. With that comes **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.

6.3 Comparison Between Monolith and Microservices

This section recaps and expands upon differences between monolith and microservice architectures.

6.3.1 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 endpoints need to be smarter. Also, **communication over a network can be less reliable**.

6.3.2 How is a monolith deployed vs. 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.

6.3.3 How is a monolith scaled vs. 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 s**pace, memory, processing** speed, etc. 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.

6.3.4 How is a monolith tested vs. 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**).

6.3.5 How is a monolith upgraded vs. 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, etc.). **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).

6.3.6 How is the database used in a monolith vs. 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 backing up and restoring, among other drawbacks. However, if you only have one database, that's **just one place for managing** database access accounts and one database to maintain / back up / restore / etc. In contrast, **each microservice** typically has its **own data storage**.

6.4 Conclusion

The **microservices** architecture has advantage of being **modular**, where each service can be independently-managed. Communication mechanisms between modules can be **standardized**. However, creating

a **monolith** can require **less planning** ahead of time and modules within a monolith can **communicate directly**, which can be **more reliable, less expensive, and provide better consistency** than communicating to many pieces of software through a network.

6.5 Additional Resources

Martin Fowler (May 2011). *TolerantReader*. `https://martinfowler.com/bliki/TolerantReader.html`

Martin Fowler (July 2015). *Microservice Trade-Offs*. `https://martinfowler.com/articles/microservice-trade-offs.html`

Martin Fowler and J Lewis (Aug. 2019). *Microservices Guide*. `https://martinfowler.com/microservices/`

IBM (n.d.). *HTTP Responses*. `https://www.ibm.com/docs/en/cics-ts/5.3?topic=protocol-http-responses`. Accessed: 2021-01-01

IBM Cloud Education (Apr. 2021a). *ESB (Enterprise Service Bus)*. `https://www.ibm.com/cloud/learn/esb`

IBM Cloud Education (Apr. 2021b). *REST APIs*. `https://www.ibm.com/cloud/learn/rest-apis`

Vaughan Michell (2011). "A focussed approach to business capability". In: *First International Symposium on Business Modelling and Software Design–BMSD*, pp. 105–113

Mozilla Developer Network (n.d.). *HTTP Messages*. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages`. Accessed: 2021-01-01

Sam Newman (2015). *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc.

Chapter 7

Paper Prototyping

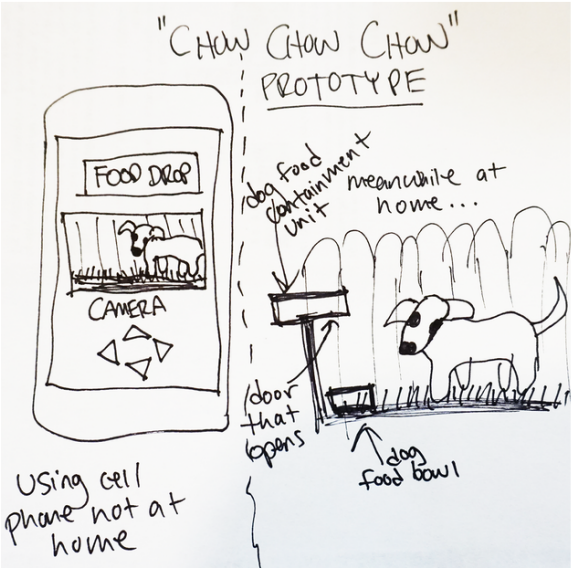
User interface (UI) design often involves prototyping: Iteratively creating depictions of what you think the UI should look like, and how users should interact with it, based on the software's requirements. **Prototyping** gives you a way to try out a UI design and find problems early. Changing a drawing (digital or physical) is 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. **Definitions I use:**

- **Low-fidelity**: A rough sketch that is often drawn by hand, drawn using an app and stylus, or made using software specif-

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, etc.).

ically 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.

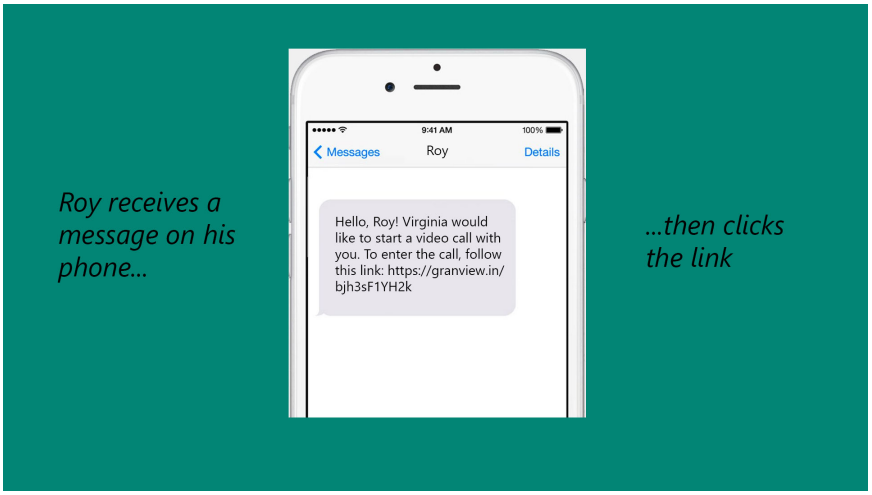


graphical user interface (GUI): A user interface with interactive graphics, in contrast to a text-based user interface.

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

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.

- **Medium-fidelity**: A **detailed** illustration often created using a professional drawing or presentation tool (e.g., Visio, PowerPoint, etc.), 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.



- **High-fidelity**: A **polished**, detailed illustration that looks like a finished UI. These designs might be created in a full-featured

graphics editor (e.g., 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 very focused and incremental improvements.

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

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**.

paper prototype: A hand-drawn sketch used to communicate a potential user interface design to be implemented, especially a graphical user interface design (Snyder 2003).

A paper prototype is a hand-drawn sketch of a UI design that's based on the software's requirements. It **does not 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).

7.1 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

interaction design: An approach to technology design that involves helping users understand what's happening with the technology, what just happened, and what they can do (Norman 2013).

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.

7.2 Showing Your Concept to Others

think-aloud protocol: A method for gathering feedback about the usability of a design that involves a test user speaking their thoughts as they interact with the design (C. Lewis, Rieman, and Blustein 1993). More information: `https://tinyurl.com/think-aloud-protocol`

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 an objective (e.g., submit data report) or let them explore on their own. Watch as they tap buttons or otherwise interact. Be ready to place other drawings on top of the one they have to indicate what would happen in the real software (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, etc.

7.3 Conclusion

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.

7.4 Additional Resources

Clayton Lewis, John Rieman, and Amended J. Blustein (1993). *Task-Centered User Interface Design: A practical introduction. A shareware book published by the authors*. URL: https://web.archive.org/web/20201126014548/http://www.hcibib.org/tcuid/tcuid.pdf

Don Norman (2013). *The Design of Everyday Things: Revised and Expanded Edition*. eng. Rev. and expanded ed. Boulder: Basic Books. ISBN: 9780465050659

Carolyn Snyder (2003). *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces*. eng. The Morgan Kaufmann series in interactive technologies. Kidlington: Elsevier Science & Technology. ISBN: 9781558608702. URL: https://web.archive.org/web/20140628171628/http://www.paperprototyping.com/

Chapter 8

Cognitive Style Heuristics

The **Cognitive Style Heuristics (CSH)** are eight principles of **interaction design** used to improve software usability. They are framed around how different people use software in different ways. The CSH were created by a research team headed by Margaret Burnett, one of the world's leading experts in usability research and **inclusive software design**.

The CSH were created with new users in mind: People who have never seen, interacted with, or received previous direction on the software. They can also improve usability for more seasoned users who have figured out how to use the software to complete their tasks but may still be unhappy with the software.

Cognitive Style Heuristics (CSH): Eight principles of interaction design for finding and fixing usability bugs in software. They are based around different cognitive styles different people use when they problem-solve in software.

interaction design: An approach to technology design that involves helping users understand what's happening with the technology, what just happened, and what they can do (Norman 2013).

8.1 Cognitive Style Facets

inclusive software design: A type of software user interface design with the goal of increasing usability for traditionally under-served user populations while also increasing usability for mainstream users.

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

motivation CSF: Why someone is using the software (task completion vs. interest)

information processing style CSF: How a person looks through or absorbs information in software (comprehensively vs. selectively)

computer self-efficacy CSF: A person's confidence in their ability to use computers or software (low vs. high)

At the core of the Cognitive Style Heuristics are the **cognitive style facets**, five aspects of human cognition that affect how users problem-solve in software:

1. **Motivations** for using software (task completion vs. tech interest). A person who is feeling task-motivated might choose to use software because they have something specific they need to accomplish, and might focus on that task immediately when they open the software and the whole time they're using it. A person who is feeling interested in the software itself might seek out the new and exciting features and spend a lot of time exploring them.

2. **Information processing style** (comprehensive vs. selective). A person processing comprehensively might want to understand details, implications, or to get a sense of overall structure before taking action in software. A person processing selectively might take action as soon as they detect what seems like the beginning of a promising path.

3. **Computer self-efficacy** (low vs. high). A person who is feeling low computer self-efficacy might think it's their fault when they make a mistake or encounter a problem in the software. A person feeling high computer self-efficacy might feel the software is poorly made if they make a mistake or encounter a problem in the software.

4. **Attitude toward risk** (risk-tolerant vs. risk-averse). A person who is feeling risk-averse might avoid taking actions that have unknown consequences or seem dangerous or irreversible. A person who is feeling risk-tolerant might take actions even if they know those actions could lead to bad consequences.

5. **Learning style** (tinkering vs. by process). When someone is tinkering, they might click a button just because it's clickable then learn from what happens. When someone is learning by process, they might seek a logical first step, and want to proceed smoothly from start to finish.

As you probably noticed, each of the cognitive style facets has two polar **cognitive style facet values** (or **cognitive styles**). Each

pair of facet values creates a spectrum. When each of us uses software, the way we feel and behave corresponds to somewhere on each of those spectra. Our individual facet values may be similar each time we use software, but they can also vary by context and change over time. For example, many people feel cognitively impatient when reading paragraphs of text ("text walls") on websites and might process them more selectively, whereas they might want to catch every word of a new novel by their favorite author (comprehensive processing).

8.2 Cognitive Style Personas

The CSH are stated from the perspective of improving usability for the three **cognitive style personas**: Abi, Pat, and Tim. A **persona** is fictional person that is created to represent a group of users within a target audience. Personas are used to help marketing teams keep important subsets of their target audience in mind, and in software UI design for the same reason. Personas are typically documents that include a photo, name, age, gender, other background information, and information about how the made-up person interacts with product or software.

The cognitive style personas have a similar purpose but are distinct from traditional personas in multiple ways:

- There are three and only three (Abi, Pat, and Tim).
- Abi, Pat, and Tim each have a different set of cognitive styles. The cognitive styles are fixed.
- Abi, Pat, and Tim are each multi-personas: They each have multiple photos of different people who appear to be of different ages, races, genders, etc.
- Abi, Pat, and Tim were specifically created for evaluating *software* (not marketing).
- Abi, Pat, and Tim represent different positions on the cognitive style facet spectrum: Abi and Tim are on the ends and Pat is in the middle.

The idea of the cognitive style personas is that creating software that works well for Abi and Tim (the two ends of the cognitive style facet value spectrum) will result in software that's better for them and everyone in between.

attitude toward risk CSF: How willing a person is to take chances in software (risk-tolerant vs. risk-averse)

learning style CSF: How a person prefers to move through software (tinkering vs. by process)

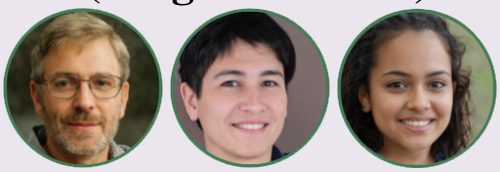
cognitive style facet value (A.K.A., cognitive style): A position on the spectrum of a cognitive style facet

persona: A fictional character that represents a subset of users in a target audience. Personas are used in marketing and UI design to help with focusing on particular groups of users and customers (Pruitt 2010; B. Martin 2012).

cognitive style personas: Three specialized personas (Abi, Pat, and Tim) used for making software UI designs more usable to people with different cognitive styles.

8.2.1 Abi, Pat, and Tim

Abi (Abigail/Abishek)



Motivation: Uses technology to accomplish their tasks.
Computer self-efficacy: Lower self-confidence than their peers about doing unfamiliar computing tasks. Blames themselves for problems.
Attitude toward risk: Risk-averse about using unfamiliar technologies that might require a lot of time
Information processing style: Comprehensive
Learning style: Process-orientated learning

Pat (Patricia/Patrick)

Motivation: Learns new technologies when they need to
Computer self-efficacy: Medium confidence doing unfamiliar computing tasks. If a problem can't be fixed, they will keep trying
Attitude toward risk: Risk-averse and doesn't want to expend time when they might not receive benefits
Information processing style: Comprehensive
Learning style: Likes to explore and purposefully tinker

Tim (Timara/Timothy)

Motivation: Likes learning all the available functionality on all their devices
Computer self-efficacy: High confidence in technical abilities. If a problem can't be fixed, blame goes to software vendor.
Attitude toward risk: Doesn't mind taking risk using features of technology
Information processing style: Selective
Learning style: Likes tinkering and exploring

8.3 The Heuristics

Adapted from (Burnett, Sarma, Hilderbrand, Steine-Hanson, Mendez, Perdriau, et al. 2021).

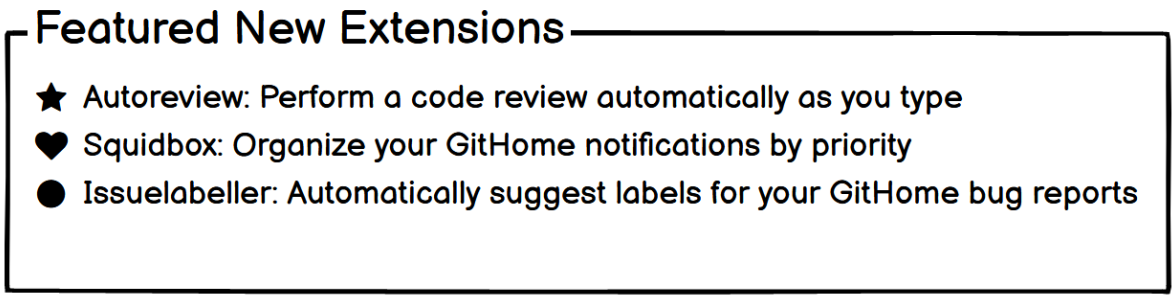
Note: The designs shown below were modelled after examples found in published software.

8.3.1 Heuristic #1 (of 8): Explain the *benefits* of using new and existing features

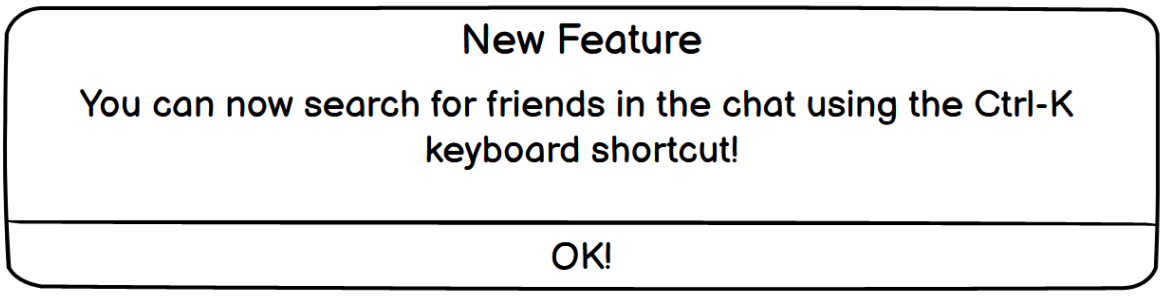
- **Abi and Pat are task-motivated** so might lose interest if they don't see how a feature relates to their task.
- **Abi is risk-averse** so might avoid features with too many unknowns.
- **Tim is risk-tolerant** and **motivated by tech interest** so might take a chance on features then be disappointed at how mundane they are.

To support users' motivations and attitudes toward risk, **provide Abi and Pat ways to decide whether a feature relates to their task and provide Tim ways to decide whether a feature is new and unique**.

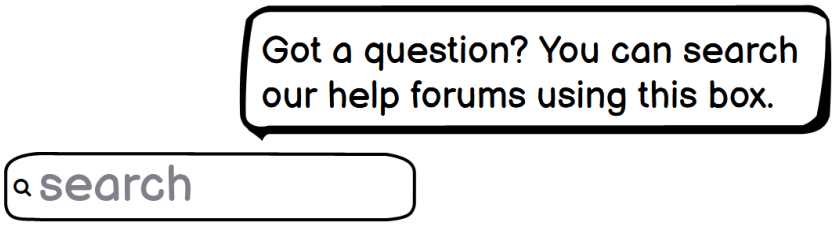
Example 1: Each featured extension has a brief description that says what the extension does and why somebody would use it.



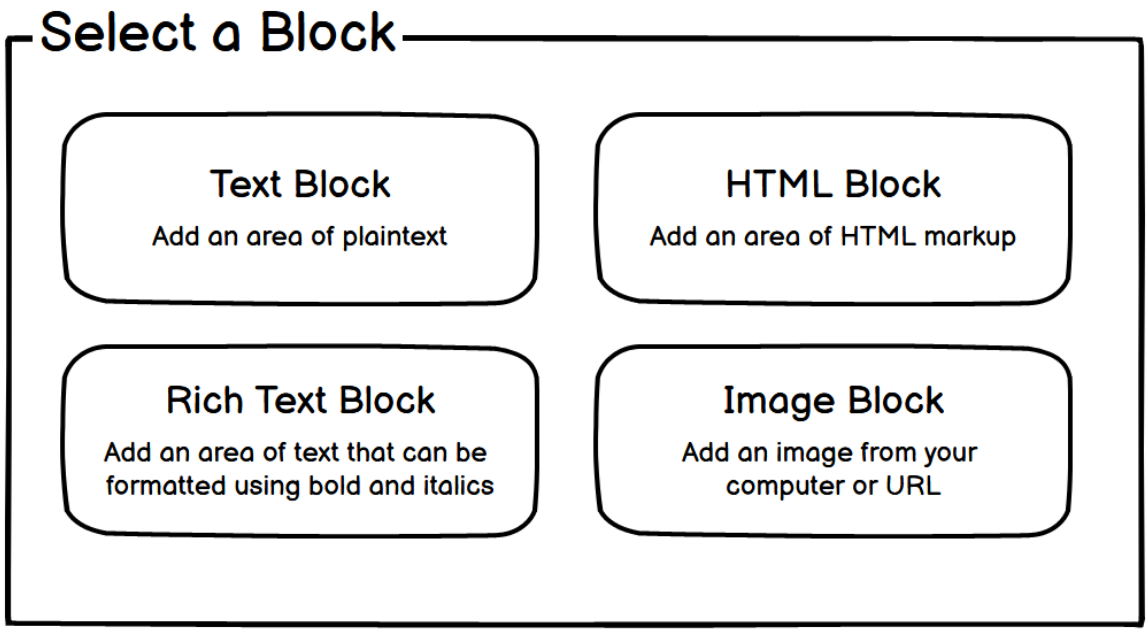
Example 2: Announcement briefly describes a new feature and how to use it.



Example 3: Tooltip says why someone might use the search.



Example 4: Each tile explains a feature and the benefit of using the feature.

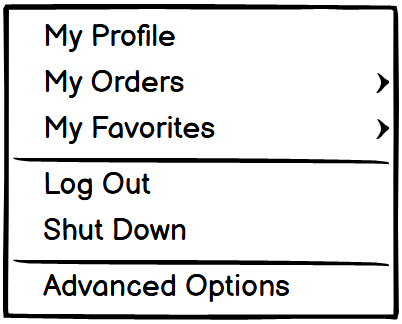


8.3.2 Heuristic #2 (of 8): Explain the *costs* of using new and existing features

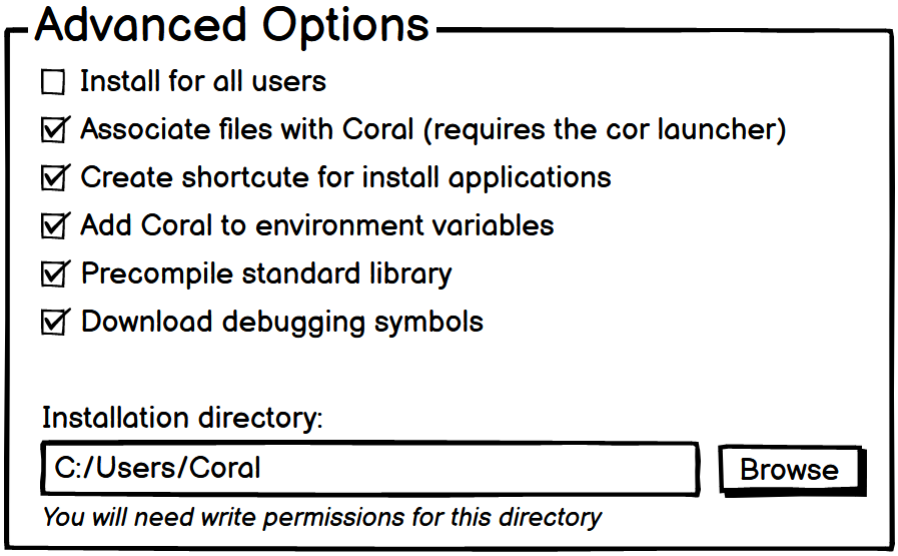
- Abi and Pat are risk-averse, so they may want to avoid features with high effort costs if the benefits of using these features are unclear.
- Tim is risk-tolerant, so may begin using features that require extra effort and time, and that are unrelated to the task at hand.

To support their **attitudes toward risk**, allow Abi and Pat to decide whether or not a feature will require too much effort to use. To help Tim stay on track with their task, allow them to understand that a feature may take extra effort, and thus more time.

Example 1: Placing "Advanced Options" at the bottom of the menu indicates to the user that "advanced" features may take more effort.



Example 2: The dialog indicates that "cor launcher" will be needed to "associate files with Coral" and that the user will need write permissions for the installation folder.

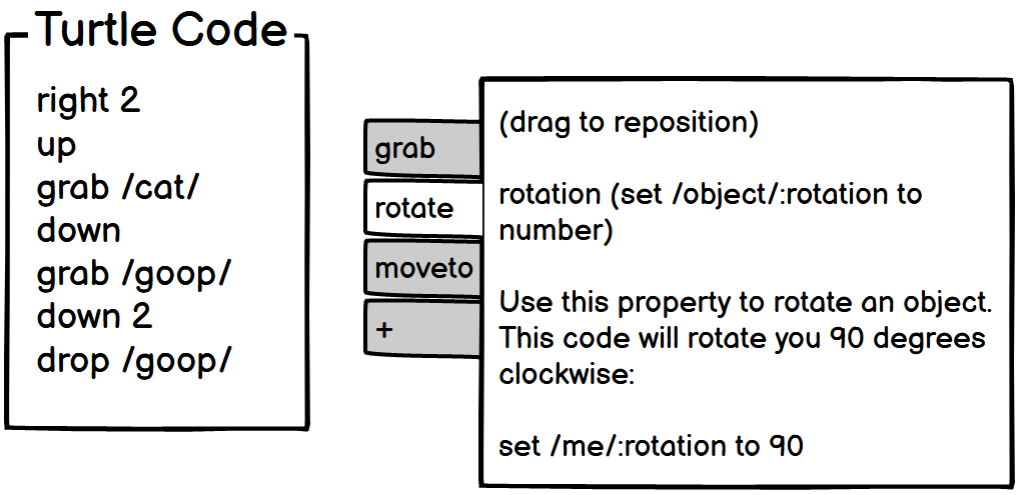


8.3.3 Heuristic #3 (of 8): Let people gather as much information as they want, and no more than they want

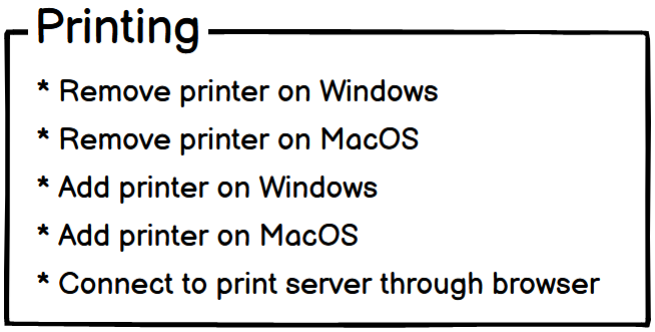
- Abi and Pat gather and read relevant information comprehensively before acting.
- Tim likes to delve into the first option and pursue it, backtracking if need be.

To support their **information processing styles**, allow Abi and Pat to easily obtain as much information they want, but don't require them to spend excessive time or effort gathering that information. Allow Tim to get to directly useful information immediately so that they can act upon it without wading through a lot of information they don't want.

Example 1: Users can choose to view code documentation while still viewing their code.



Example 2: Users can quickly see the contents of the webpage and jump to the section they're interested in.

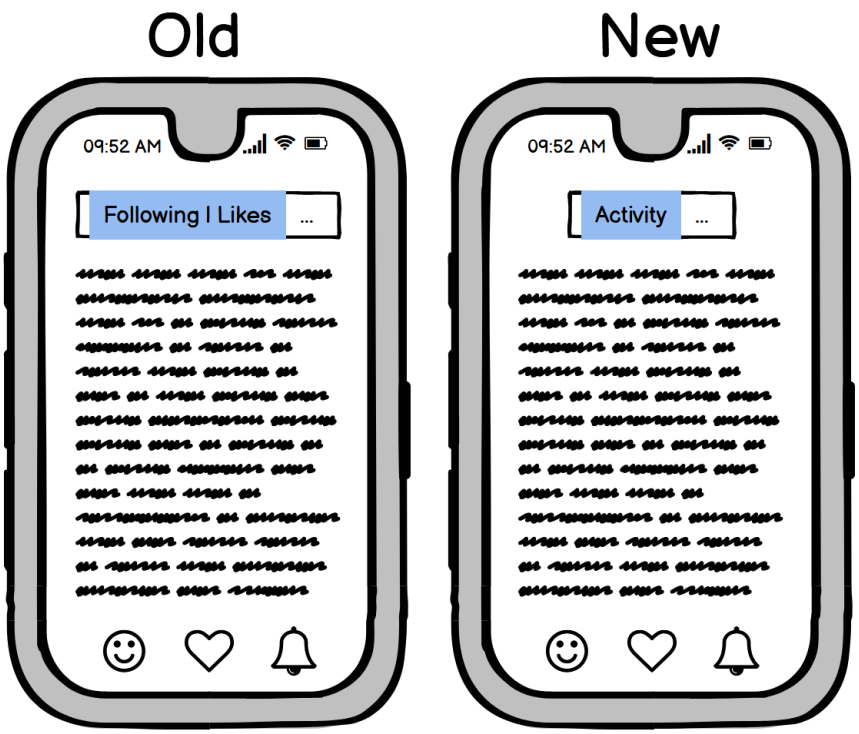


8.3.4 Heuristic #4 (of 8): Keep familiar features available

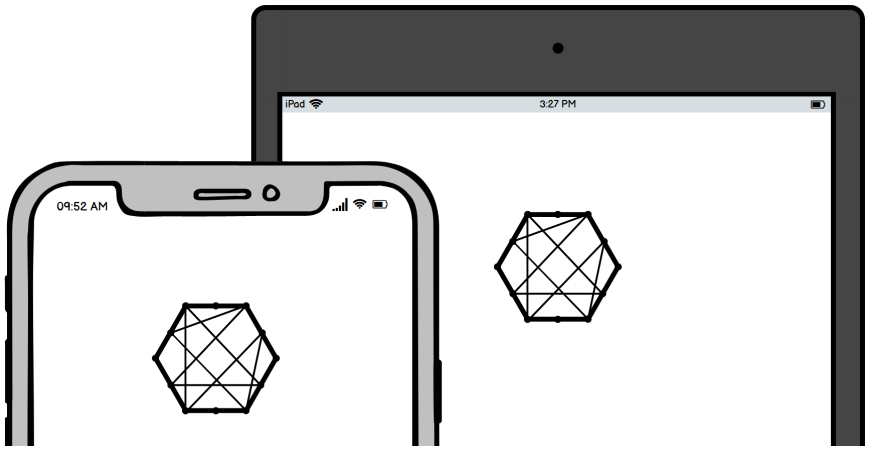
- Abi has lower computer self-efficacy and is more risk-averse than Tim, so if a problem arises when they are trying to use an unfamiliar feature, Abi blames themself and stops using the tech rather than potentially wasting their time trying to get the unfamiliar feature working.
- Pat has medium self-efficacy with technology, so if a problem arises when they are trying to use an unfamiliar feature, Pat will try alternative ways of succeeding for a while. However, Pat is also risk-averse so prefers to perform tasks using familiar features, because they're more predictable about what Pat will get from them and how much time they'll take.
- Tim has higher computer self-efficacy and is more risk-tolerant than Abi, so if a problem arises when they are trying to use an unfamiliar feature, they'll blame the tech, and may spend a lot of extra time trying to work around a problem in numerous ways.

To support their **computer self-efficacies** and **attitudes toward risk**, and to encourage Abi, Pat, and Tim to keep using the tech without wasting their time, enable them to interact with it using the same features they've used in the past.

Example 1: Although the "following" page is gone, the new update looks similar to the previous version so that users are still familiar with the app.



Example 2: The smartphone and tablet versions of this app offer the same features which makes switching between the two easy.

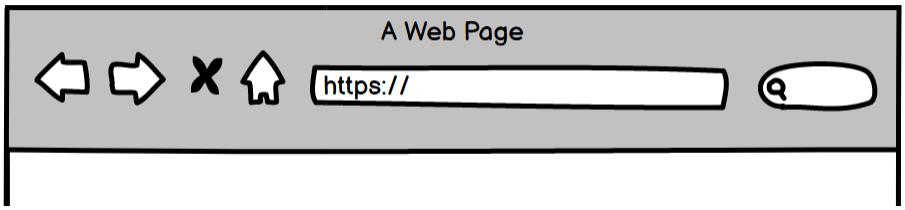


8.3.5 Heuristic #5 (of 8): Make undo/redo and backtracking available

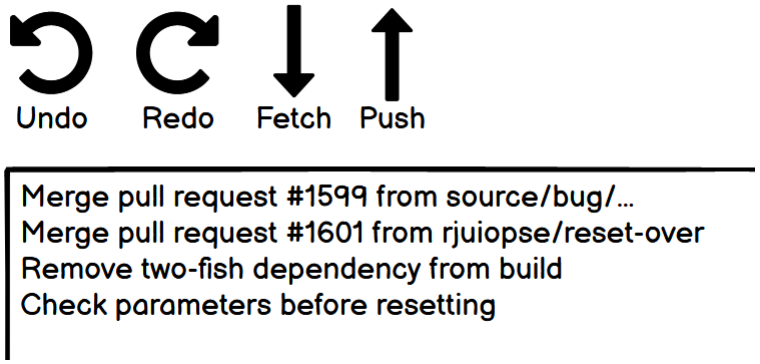
- Abi and Pat are risk-averse, so they prefer not to take actions in technology that might not be easy to reverse.
- Tim is risk-tolerant, so is willing to take actions in technology that might be incorrect and need to be reversed.

To support their **attitudes toward risk**, provide undo/redo and backtracking to allow Abi and Pat to feel comfortable proceeding with actions whose consequences may not be clear, so that that they know they can easily reverse these actions, and so that Tim can recover from mistakes.

Example 1: Browser back/forward buttons allow users to backtrack through their browsing history.



Example 2: An undo button allow users to make and recover from mistakes. Also, version control systems allow users to revert to any previously-committed code state.

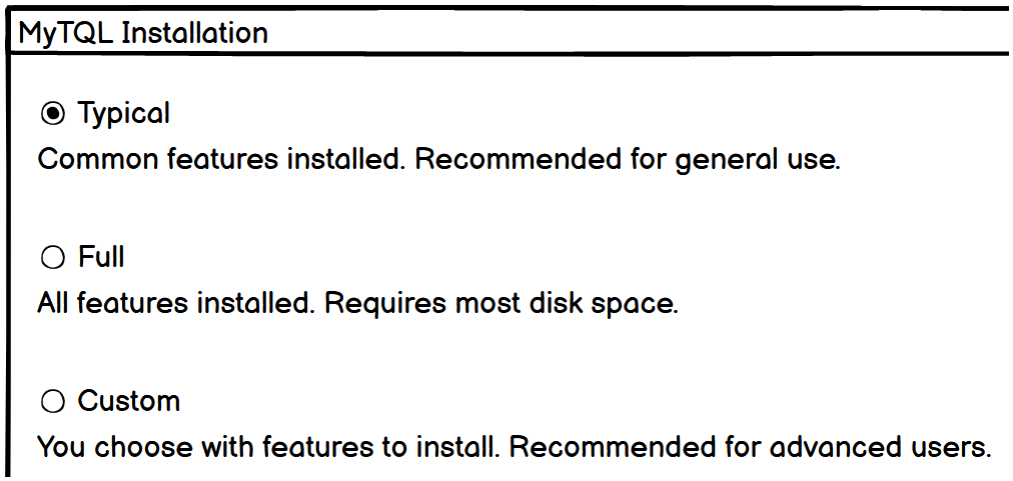


8.3.6 Heuristic #6 (of 8): Provide an explicit path through the task

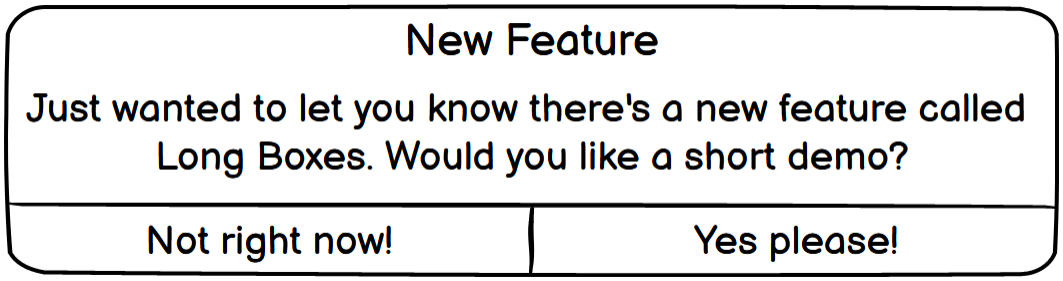
- Abi is a process-oriented learner, so prefers to proceed through tasks step-by-step.
- Tim and Pat learn by tinkering, and therefore prefer not to be constrained by rigid, pre-determined processes.

To support their **learning styles**, explicitly provide Abi a clear process to go through the task, and provide Tim and Pat a way to bypass step-by-step processes and tutorials if those are not required for learning the technology.

Example 1: Users can choose their entry point, and each path is explained.



Example 2: Users get to choose either the path of learning more about the new feature or going back to what they were doing.



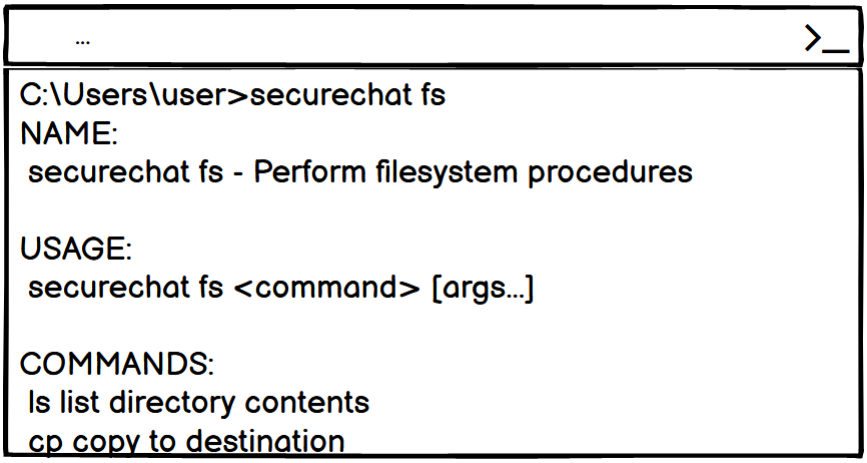
8.3.7 Heuristic #7 (of 8): Provide ways to try out different approaches

- Abi has lower computer self-efficacy than Tim, so if a problem arises when they are trying to use technology, Abi blames themself and stops using the tech.
- Pat has medium self-efficacy with technology, so if a problem arises when they are trying to use technology, Pat will try alternative ways of succeeding for a while.
- Tim has higher computer self-efficacy than Abi, so if a problem arises when they are trying to use technology, they'll blame the tech, and then will try numerous workarounds to get around the problem.

To support their **computer self-efficacies**, point Abi toward a different approach when they feel unable to proceed with the current one. This will also point Tim and Pat to multiple ways they can try to solve the problem.

Example 1: If users don't find what they need on the "Choose a Question" drop-down menu, they can try the chat.

Example 2: If users encounter a problem using the SecureChat UI, they can attempt the same operations using the command line interface.

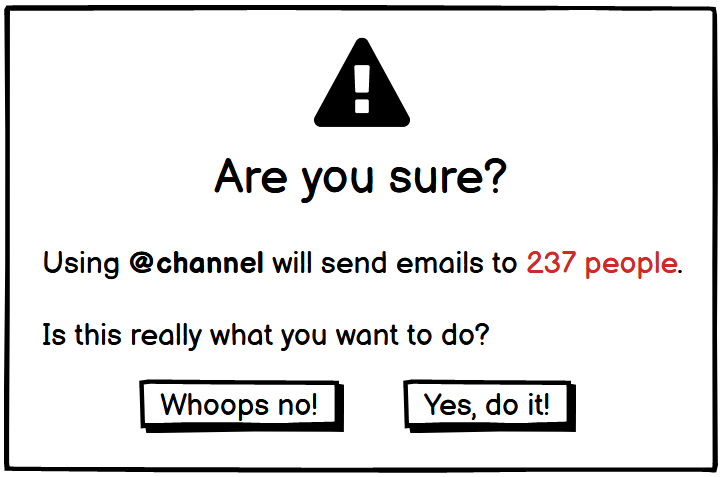


8.3.8 Heuristic #8 (of 8): Encourage tinkerers to tinker mindfully

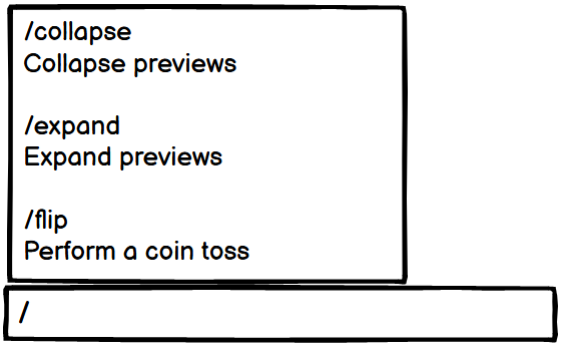
- Tim learns by tinkering, but sometimes tinkers addictively and gets distracted from their task.
- Pat learns by trying out new features but does so mindfully, reflecting on each step.

To support their **learning styles**, encourage Tim not to over-tinker (e.g., by adding an extra click), so that they make fewer mistakes, have time to absorb important information, and stay on-task.

Example 1: This design encourages users to tinker mindfully by showing they will notify them before impactful actions are executed, like emailing 237 people.



Example 2: This design encourages users to try out new "slash" commands by showing all the commands when a user types "/", and explaining what each does and how to use it.



8.4 Background

heuristic evaluation: A usability inspection method where evaluators independently check that a design reflects a set of heuristics, then compare results (Nielsen and Molich 1990).

The Cognitive Style Heuristics are meant to be used in a **heuristic evaluation**, a process where software designers or evaluators go through heuristics one-by-one like a checklist, deciding whether the design does or does not reflect the heuristic. The evaluation is done independently by two or more people, who then compare findings.

The Cognitive Style Heuristics are derived from the GenderMag Heuristics (Burnett, Sarma, Hilderbrand, Steine-Hanson, Mendez, and Perdriau 2018) and the **GenderMag Method** (Burnett, Stumpf, et al. 2016). The GenderMag Method is a process for finding and fixing gender-inclusivity bugs in software. Instead of heuristic evaluation, it uses a **cognitive walkthrough**. It uses the same personas: Abi, Pat, and Tim. However, in addition to their cognitive styles, each GenderMag persona has additional sections, such as one with customizable background information.

The cognitive style personas are simplified versions of the GenderMag personas. You can find the the GenderMag personas, and a full description of their research origins, at GenderMag.org

GenderMag Method: A method for finding and fixing gender-inclusivity bugs in software that uses a specialized cognitive walkthrough and the customizable Abi, Pat, and Tim personas (Burnett, Stumpf, et al. 2016)

cognitive walkthrough: A usability inspection method that involves stepping through a user interface as a user, stopping to ask specific questions about the user's experience (Nielsen and Mack 1994).

What do cognitive styles have to do with gender? Software tends to be biased against the cognitive styles often favored by women. Designing with cognitive styles in mind can make software less gender-biased (Vorvoreanu et al. 2019).

In addition, "designing software so that it works for diverse populations matters to software companies' profitability, to equity in the workplace and at home, and to anyone in a situation that changes the way they think, such as when under deadline pressure."(Mendez et al. 2019)

8.5 Conclusion

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

8.6 Additional Resources

Margaret Burnett, Simone Stumpf, et al. (Oct. 2016). "GenderMag: A Method for Evaluating Software's Gender Inclusiveness". In: *Interacting with Computers* 28.6, pp. 760–787. ISSN: 0953-5438. DOI: 10.1093/iwc/iwv046. eprint: https://academic.oup.com/iwc/article-pdf/28/6/760/7919992/iwv046.pdf. URL: https://doi.org/10.1093/iwc/iwv046

Charles G Hill et al. (2017). "Gender-Inclusiveness Personas vs. Stereotyping: Can we have it both ways?" In: *Proceedings of the 2017 chi conference on human factors in computing systems*, pp. 6658–6671

GenderMag.org (n.d.). http://gendermag.org. Accessed: 2020-12-27

Margaret Burnett, Anita Sarma, Claudia Hilderbrand, Zoe Steine-Hanson, Christopher Mendez, and Christopher Perdriau (July 2018). *The GenderMag Heuristics (Beta Version)*. https://gendermag.org/flyers_handouts.php

Bella Martin (2012). *Universal methods of design : 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Digital ed. Beverly, MA: Rockport Publishers. ISBN: 9781610581998

Christopher Mendez et al. (2019). "From GenderMag to InclusiveMag: An Inclusive Design Meta-Method". eng. In:

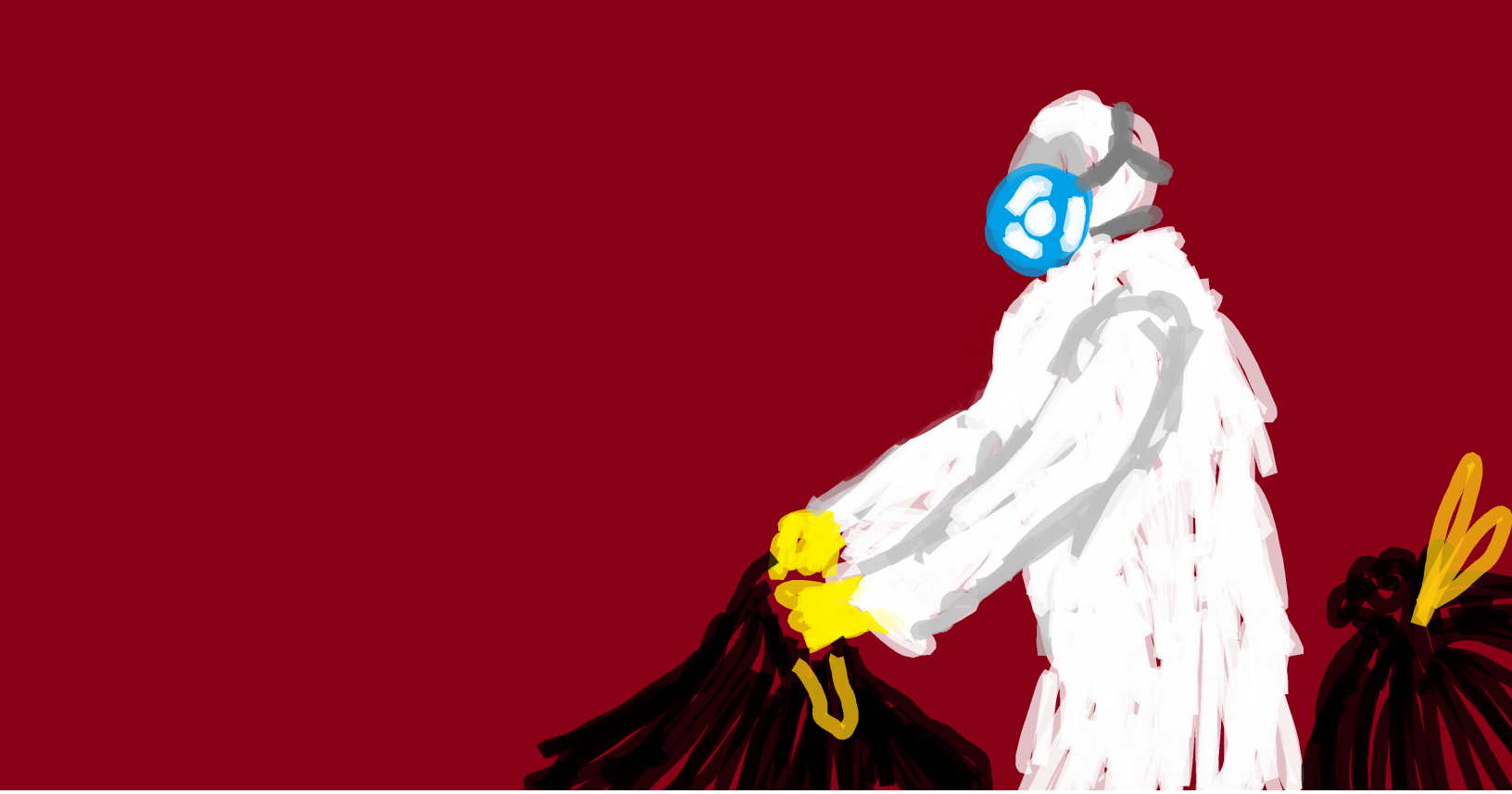
Jakob Nielsen and Rolf Molich (1990). "Heuristic Evaluation of User Interfaces". In: *IN: PROCEEDINGS OF THE CHI'90 CONFERENCE, SEATTLE*. S, pp. 249–256

Jakob Nielsen and Robert L. Mack (1994). *Usability inspection methods*. New York

Don Norman (2013). *The Design of Everyday Things: Revised and Expanded Edition*. eng. Rev. and expanded ed. Boulder: Basic Books. ISBN: 9780465050659

John Pruitt (2010). *The essential persona lifecycle : your guide to building and using personas*. San Francisco, Calif. : Oxford: Morgan Kaufmann ; Elsevier Science [distributor]. ISBN: 9780123814180

Mihaela Vorvoreanu et al. (2019). "From Gender Biases to Gender-Inclusive Design: An Empirical Investigation". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, pp. 1–14. ISBN: 9781450359702. DOI: 10.1145/3290605.3300283. URL: https://doi.org/10.1145/3290605.3300283



Chapter 9

Code Smells and Refactoring

Code smells are aspects of code that indicate the code needs to be reorganized—signs your software is **decaying**. 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 re-write this code from scratch."
- "Every time I look at this code, I have to **re-figure-out** what it does."
- "I **don't** think these code comments **match** what the code is doing..."
- "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."

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

code decay (AKA software rot): Reduction of code quality over time. Can result in decreased maintainability, more bugs, and irretrievable failure.

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)

If you want to learn more about any of the code smells and refactorings described in this chapter, or want to know MORE ways your code can smell, (R. C. Martin 2013) and (Shvets 2021) are two good resources.

9.1 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.

If you think it's more fun to write code than organize code, you may need to be strict with yourself about using good programming practices.

- 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. Ultimately, the software may have to be redeveloped entirely (which doesn't always solve the problem). Or, the project could fail.

9.2 Your code stinks, now what?

If you're in a position to (e.g., your manager allows it), strongly consider **refactoring**. Refactoring is when you improve your code without changing what the code does. Refactoring is how you fight against technical debt.

refactoring: Improving code design without changing what the code does.

The remainder of this chapter is about code smells and how to clean them up. This is not an exhaustive list. You can find a lot more by looking through the resources listed at the end of this chapter.

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.

9.3 Comments

When we first learned to code, many of us didn't write comments: solving problems is fun and coding can be addictive, no time for boring comments! Then, we got more experience, started coding with others, were formally trained on coding, or attempted to pick up 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**.

9.3.1 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 a 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 means greater likelihood some will be neglected, 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", that code is more important to call attention to with comments.

- Writing lots of comments could **indicate the code needs to be simplified**. Ideally, most of the the code you write will be self-explanatory so that comments are infrequently needed.

9.3.2 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.

As a challenge to myself, I kept the code example boxes narrow and tried to make the "good" code fit.

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, etc.)

```
# 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 into ASCII, then
    that creates a visualization of a
    typewriter typing the input. Problem is, as
     you might imagine, sometimes there's no
    good conversion to ASCII and so some
    meaning is lost.
"""
```

9.4 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 need only draw those two buttons. Then, when you add a menu and a settings button, you could update the function to draw those elements, too. Then you 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**.

9.4.1 Code Smells about Functions

Software made of 3 to 4-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()

```

```
# AFTER
def updateState():
    updateTime()
    updateScores()

def updateGUI():
    updateTimeDisplay()
    updateScoreDisplay()
    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!

```
# BEFORE
initOutdoorPlace(floraList, faunaList,
   temperature, windSpeed, cloudiness,
   rockiness, birdNoises, grassLength)

# AFTER
initOutdoorPlace(world1data)
```

9.5 Code

Code smells should not be refactored blindly. Always consider how your changes might affect the rest of your software; living with smells is sometimes the wiser choice.

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.

9.5.1 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, etc.

```
# BEFORE
if (rectangle.coordinate[1][0] - rectangle.
   coordinate[2][0] > 500 && rectangle.
   coordinate[2][1] - rectangle.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.

```
# BEFORE
if (whale.isSinging) {
```

Thresholds like "100 characters" or "5 lines" are somewhat 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!

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

```
    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);
}
```

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

- **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.

```
# BEFORE
a = 100
b = 2

# AFTER
retail_price = 100
wholesale_multiplier = 2
```

9.6 Conclusion

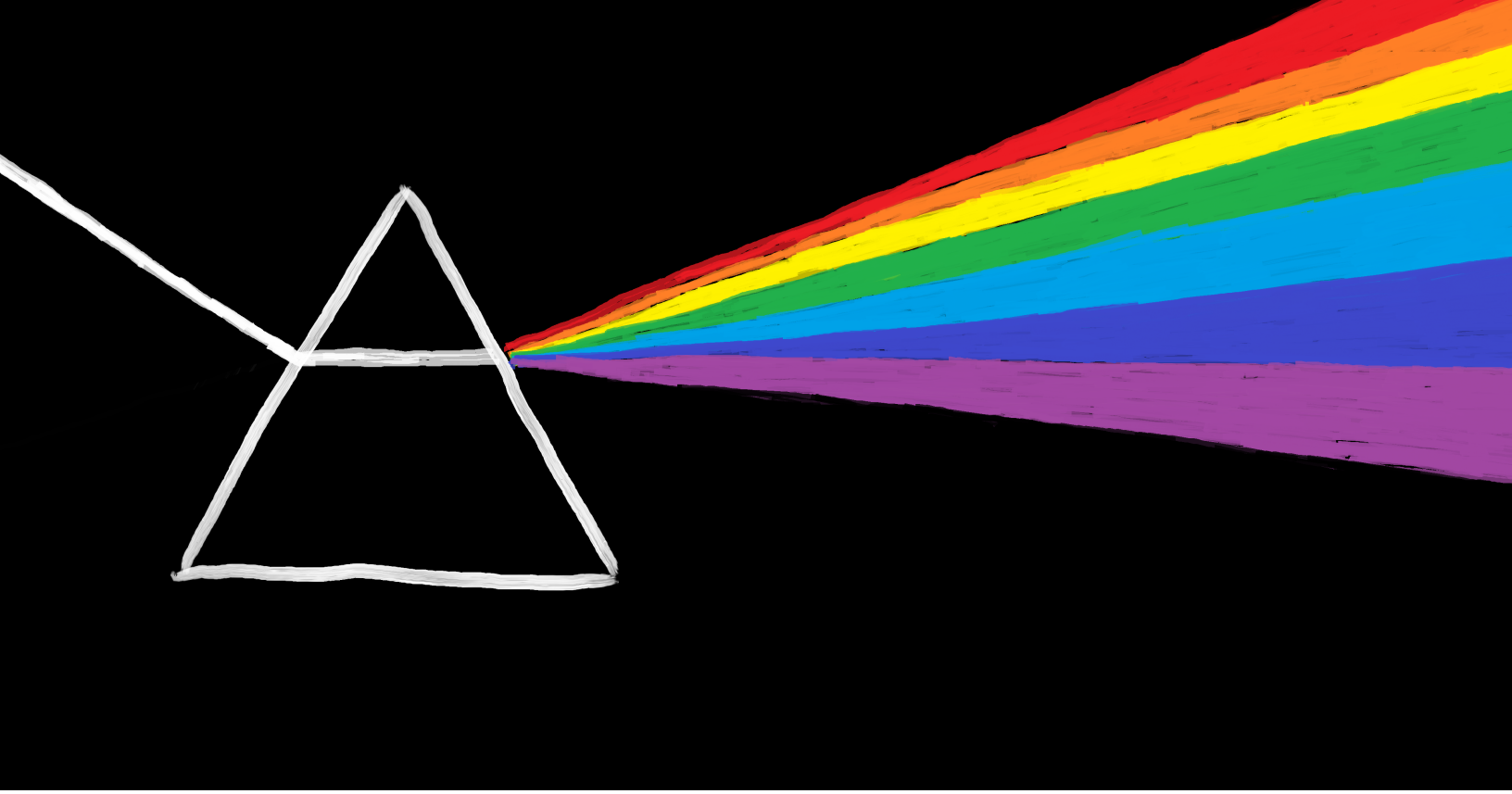
Cleaning up your code can help make your software sustainable and extensible and can make your teammates happier too.

9.7 Additional Resources

Martin Fowler (2019b). *Refactoring : improving the design of existing code*. Boston

Robert C. Martin (2013). *Clean Code*

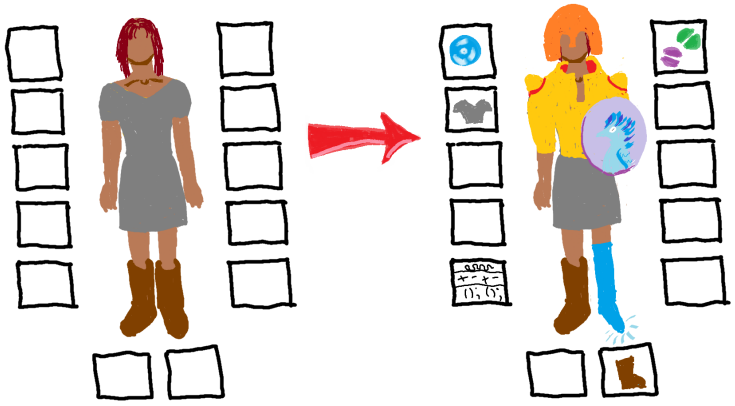
Alexander Shvets (2021). *Refactoring Guru*. Accessed: 2021-01-05. URL: https://refactoring.guru/



Chapter 10

Conclusion

I hope you're now better equipped for your next software project. Updated versions of this book will be available at `https://github.com/setextbook`



Glossary

A

acceptance criterion: A statement 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: Individuals and interactions, working software, customer collaboration, and responding to change.

attitude toward risk CSF: How willing a person is to take chances in software (risk-tolerant vs. risk-averse)

B

business capability: "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" (Michell 2011)

C

class diagram: 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 (a.k.a. customer): 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).

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

code decay (AKA software rot): Reduction of code quality over time. Can result in decreased maintainability, more bugs, and irretrievable failure.

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

cognitive style facets (CSFs): 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 facet value (A.K.A., cognitive style): A position on the spectrum of a cognitive style facet

Cognitive Style Heuristics (CSH): Eight principles of interaction design for finding and fixing usability bugs in software. They are based around different cognitive styles different people use when they problem-solve in software.

cognitive style personas: Three specialized personas (Abi, Pat, and Tim) used for making software UI designs more usable to people with different cognitive styles.

cognitive walkthrough: A usability inspection method that involves stepping through a user interface as a user, stopping to ask specific questions about the user's experience (Nielsen and Mack 1994).

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

computer self-efficacy CSF: A person's confidence in their ability to use computers or software (low vs. high)

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.

D

Definition of Done (DoD): The set of acceptance criteria which, once satisfied, mean a user story has been satisfactorily implemented.

E

Eisenhower matrix: 2x2 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 that 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.

extensible: Built in such a way to support adding more functionality later.

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

F

fist of five: A method for gauging and building group consensus that uses a 6-level voting system (zero to five fingers).

focus group (in usability engineering): A moderated discussion between researcher and a small number of potential users (usually 6-12) during which the researcher tries to gather information about the participants' attitudes, opinions, motivations, concerns, and problems related to a specific product or topic.(Odimegwu 2000)

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

G

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

GenderMag Method: A method for finding and fixing gender-inclusivity bugs in software that uses a specialized cognitive walkthrough and the customizable Abi, Pat, and Tim personas (Burnett, Stumpf, et al. 2016)

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.

H

heuristic evaluation: A usability inspection method where evaluators independently check that a design reflects a set of heuristics, then compare results (Nielsen and Molich 1990).

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.

I

IDE: Integrated development environment. Software specifically for creating software.

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

inclusive software design: A type of software user interface design with the goal of increasing usability for traditionally under-served user populations while also increasing usability for mainstream users.

increment: In software, a measurable increase in functionality.

interaction design: An approach to technology design that involves helping users understand what's happening with the technology, what just happened, and what they can do (Norman 2013).

interaction diagram: Visualization of collaboration between different parts of software.

INVEST: Characteristics of good user stories (independent, negotiable, valuable, estimable, small, testable) (Wake 2003).

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.

L

learning style CSF: How a person prefers to move through software (tinkering vs. by process)

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

M

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

managerial skill mix (MSM): Three categories of skills used by managers: (1) interpersonal, (2) technical, (3) administrative/conceptual.

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 pre-established way of achieving a specific outcome.

microservices architecture: Overall code design characterized by multiple independent components that each run in their own process and communicate between one another without direct access.

mitigation plan: What you will do if a contingency happens.

monolith architecture: Overall code design characterized by being in one or few pieces; cannot be easily divided into components that run separately and are independently useful.

motivation CSF: Why someone is using the software (task completion vs. interest)

minimum viable product (MVP): A low-effort or low-expense effort that results in you being able to better estimate whether people will want to use your product—before the product is fully developed.(Olsen 2015)

N

non-functional requirement: Description of how well software is expected to perform.

P

paper prototype: A hand-drawn sketch used to communicate a potential user interface design to be implemented, especially a graphical user interface design (Snyder 2003).

persona: A fictional character that represents a subset of users in a target audience. Personas are used in marketing and UI design to help with focusing on particular groups of users and customers (Pruitt 2010; B. Martin 2012).

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.

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

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

project priority matrix: 3x3 grid for documenting how to respond when there are potential changes to a project's time, cost, or scope. Options: Only positive change allowed (constrain), negative change allowed (accept), or positive change sought (enhance).

Q

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

R

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: What the software must to, 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.

S

sequence diagram: 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 received and fulfills requests.

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

Scrum: An Agile framework "for developing and sustaining complex products." (Schwaber and Sutherland 2020)

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

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

software engineering: The art and science of using different methods to efficiently create extensible, sustainable programs that solve problems people care about.

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

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

Sprint Backlog: In 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.

Software Requirements Specification (SRS): A document that contains software requirements.

stakeholder: Anyone who is or will be affected by the software or its development (e.g., clients, companies, users, developers, managers, politicians, etc.)

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).

T

task management system: Software for planning and organizing project activities.

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.

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

think-aloud protocol: A method for gathering feedback about the usability of a design that involves a test user speaking their thoughts as they interact with the design (C. Lewis, Rieman, and Blustein 1993). More information: `https://tinyurl.com/think-aloud-protocol`

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.

U

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.

UML: Unified modeling language: A set of notation and methods for describing and designing software.

usability testing: Observing people while they try to use your software.(Barnum 2020)

use case: “A contract for the behavior of the system under discussion” (Cockburn 2001)

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, etc.).

user story: “Short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.” (Cohn n.d.)

V

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?”).

W

waterfall (software process model): 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 lifecycle (SDLC).

Bibliography

Alliance, Agile (n.d.). *What is "Given - When - Then?"* https://web.archive.org/web/20201124202211/https://www.agilealliance.org/glossary/gwt.

Atkinson, Roger (1999). "Project management: cost, time and quality, two best guesses and a phenomenon, its time to accept other success criteria". eng. In: *International journal of project management* 17.6, pp. 337–342. ISSN: 0263-7863.

Badawy, Michael K (1995). *Developing managerial skills in engineers and scientists: Succeeding as a technical manager*. John Wiley & Sons.

Barnum, Carol M. (2020). *Usability Testing Essentials: Ready, Set...Test!* 2nd ed. Morgan Kaufmann.

Beck, Kent (2000). *Extreme programming explained: embrace change*. addison-wesley professional.

Belling, Shawn (2020). "Agile Values and Practices". In: *Succeeding with Agile Hybrids*. Springer, pp. 47–61.

Brennan, Kevin et al. (2009). *A Guide to the Business Analysis Body of Knowledger*. Iiba.

Brown, Karen A, Nancy Lea Hyer, and Richard Ettenson (2013). "The question every project team should answer". In: *MIT Sloan Management Review* 55.1, p. 49.

Burnett, Margaret, Anita Sarma, Claudia Hilderbrand, Zoe Steine-Hanson, Christopher Mendez, and Christopher Perdriau (July 2018). *The GenderMag Heuristics (Beta Version)*. https://gendermag.org/flyers_handouts.php.

Burnett, Margaret, Anita Sarma, Claudia Hilderbrand, Zoe Steine-Hanson, Christopher Mendez, Christopher Perdriau, et al. (Mar. 2021). *Cognitive Style Heuristics (from the GenderMag Project*. URL: %5Curl%7Bhttps://web.archive.org/web/20210804014933/http://gendermag.org/Docs/Cognitive-Style-Heuristics-from-the-GenderMag-Project-2021-03-07-1537.pdf%7D.

Burnett, Margaret, Simone Stumpf, et al. (Oct. 2016). "GenderMag: A Method for Evaluating Software's Gender Inclusiveness". In: *Interacting with Computers* 28.6, pp. 760–787. ISSN: 0953-5438. DOI: 10.1093/iwc/iwv046. eprint: https://academic.oup.com/iwc/article-pdf/28/6/760/7919992/iwv046.pdf. URL: https://doi.org/10.1093/iwc/iwv046.

Cockburn, Alistair (2001). *Writing effective use cases*. Boston.

Cohn, Mike (2005). *Agile estimating and planning*. Pearson Education.

— (n.d.). *User Stories and User Story Examples*. https://web.archive.org/web/20201124004807/https://www.mountaingoatsoftware.com/agile/user-stories.

Cotton, Gayle (2013). "Gestures to avoid in cross-cultural business: In other words,'Keep your fingers to yourself!'" In: *The Huffington Post. Avaiable at:< http://www. huffingtonpost. com/gayle-cotton/cross-cultural-gestures_b_3437653. html>(retrieved July 7, 2017)*.

Eaker, Fred (Nov. 2006). *Software Requirements Specification for Vyasa*. https://web.archive.org/web/20161127184329/http://vyasa.sourceforge.net/vyasa_software_requirements_specification.pdf.

Education, IBM Cloud (Apr. 2021a). *ESB (Enterprise Service Bus)*. https://www.ibm.com/cloud/learn/esb.

— (Apr. 2021b). *REST APIs*. https://www.ibm.com/cloud/learn/rest-apis.

Enterprise, Hewlett Packard (2017). "Agile is the new normal: Adopting Agile project management". In: *Hewlett Packard Enterprise Development LP*.

Extreme Programming: A Gentle Introduction (n.d.). http://www.extremeprogramming.org/. Accessed: 2021-01-01.

Fletcher, A (2002). "FireStarter youth power curriculum: Participant guidebook". In: *Olympia, WA: Freechild Project*.

Fowler, Martin (2004). *UML distilled : a brief guide to the standard object modeling language*. Boston.

— (May 2011). *TolerantReader*. https://martinfowler.com/bliki/TolerantReader.html.

— (July 2015). *Microservice Trade-Offs*. https://martinfowler.com/articles/microservice-trade-offs.html.

— (2019a). "Agile Software Guide". In: URL: https://web.archive.org/web/20210429215912/https://martinfowler.com/agile.html.

— (2019b). *Refactoring : improving the design of existing code*. Boston.

Fowler, Martin and J Lewis (Aug. 2019). *Microservices Guide*. https://martinfowler.com/microservices/.

Fowler, Martin, Kendall Scott, et al. (2003). *UML distilled: a brief guide to the standard object*.

GenderMag.org (n.d.). http://gendermag.org. Accessed: 2020-12-27.

Hailes, Jarett (2014). *Business Analysis Based on BABOK® Guide Version 2–A Pocket Guide*. Van Haren.

Hambling, Brian and Pauline Van Goethem (2013). "User acceptance testing: a step-by-step guide". In: BCS.

Hill, Charles G et al. (2017). "Gender-Inclusiveness Personas vs. Stereotyping: Can we have it both ways?" In: *Proceedings of the 2017 chi conference on human factors in computing systems*, pp. 6658–6671.

Hulshult, Andrea R and Timothy C Krehbiel (2019). "Using Eight Agile Practices in an Online Course to Improve Student Learning and Team Project Quality." In: *Journal of Higher Education Theory & Practice* 19.3.

IBM (n.d.). *HTTP Responses*. https://www.ibm.com/docs/en/cics-ts/5.3?topic=protocol-http-responses. Accessed: 2021-01-01.

International, Standish Group (2015). "The chaos report". In: *United States of America*. URL: https://web.archive.org/web/20210325103248/https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf.

Jacka, J Mike and Paulette J Keller (2009). *Business process mapping: improving customer satisfaction*. John Wiley & Sons.

Jr., Thomas Hedberg, Moneer Helu, and Marcus Newrock (Dec. 2017). *Software Requirements Specification to Distribute Manufacturing Data*. https://web.archive.org/web/20201208070659/https://nvlpubs.nist.gov/nistpubs/ams/NIST.AMS.300-2.pdf.

Lab, Inria Innovation (n.d.). *Software Requirement Specification for CertiViBE, v1.0*. https://web.archive.org/web/20190710221933/http://openvibe.inria.fr/openvibe/wp-content/uploads/2018/04/CERT-Software-Requirement-Specification.pdf.

Larson, Erik and Clifford Gray (2018). *Project management: The managerial process*. Irwin/McGraw-Hill.

Lewis, Clayton, John Rieman, and Amended J. Blustein (1993). *Task-Centered User Interface Design: A practical introduction. A shareware book published by the authors*. URL: https://web.archive.org/web/20201126014548/http://www.hcibib.org/tcuid/tcuid.pdf.

Lucid (n.d.). *What is Fist to Five?* https://www.lucidmeetings.com/glossary/fist-five. Accessed: 2021-01-01.

Mahnič, Viljan and Tomaž Hovelja (2012). "On using planning poker for estimating user stories". In: *Journal of Systems and Software* 85.9, pp. 2086–2095.

Martin, Bella (2012). *Universal methods of design : 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Digital ed. Beverly, MA: Rockport Publishers. ISBN: 9781610581998.

Martin, Robert C. (2013). *Clean Code*.

McAlister, Debbie Thorne (2006). "The project management plan: Improving team process and performance". In: *Marketing Education Review* 16.1, pp. 97–103.

Mendez, Christopher et al. (2019). "From GenderMag to InclusiveMag: An Inclusive Design Meta-Method". eng. In:

Michell, Vaughan (2011). "A focussed approach to business capability". In: *First International Symposium on Business Modelling and Software Design–BMSD*, pp. 105–113.

Microsoft (n.d.). *The project triangle*. https://support.microsoft.com/en-us/office/the-project-triangle-8c892e06-d761-4d40-8e1f-17b33fdcf810. Accessed: 2021-01-01.

Miles, Russ and Kim Hamilton (2006). *Learning UML 2.0: a pragmatic introduction to UML*. O'Reilly Media, Inc.

Network, Mozilla Developer (n.d.). *HTTP Messages*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages. Accessed: 2021-01-01.

Newman, Sam (2015). *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc.

Nielsen, Jakob and Robert L. Mack (1994). *Usability inspection methods*. New York.

Nielsen, Jakob and Rolf Molich (1990). "Heuristic Evaluation of User Interfaces". In: *IN: PROCEEDINGS OF THE CHI'90 CONFERENCE, SEATTLE*. S, pp. 249–256.

Norman, Don (2013). *The Design of Everyday Things: Revised and Expanded Edition*. eng. Rev. and expanded ed. Boulder: Basic Books. ISBN: 9780465050659.

Odimegwu, Clifford (July 2000). "Methodological Issues in the Use of Focus Group Discussion as a Data Collection Tool". In: *Journal of Social Sciences* 4, pp. 207–212. DOI: 10.1080/09718923.2000.11892269.

Olsen, Dan (2015). *The lean product playbook : how to innovate with minimum viable products and rapid customer feedback*. Hoboken: Wiley. ISBN: 9781118961025.

Overeem, Barry (2016). *Characteristics of a Great Scrum Team*.

Parsons, Rebecca (June 2003). "Components and the world of chaos". In: *Software, IEEE* 20, pp. 83–85. DOI: 10.1109/MS.2003.1196326.

Pruitt, John (2010). *The essential persona lifecycle : your guide to building and using personas*. San Francisco, Calif. : Oxford: Morgan Kaufmann ; Elsevier Science [distributor]. ISBN: 9780123814180.

Qubaisi, Jasim MohJasim Mohamed Lahdan Fhadel Al et al. (2015). "Leadership, culture and team communication: analysis of project success causality-a UAE case". In: *International Journal of Applied Management Science* 7.3, pp. 223–243.

Royce, Winston W (1987). "Managing the development of large software systems: concepts and techniques". In: *Proceedings of the 9th international conference on Software Engineering*, pp. 328–338.

Schwaber, Ken and Jeff Sutherland (Nov. 2020). "The Scrum Guide". In: *Scrum Alliance*.

Shvets, Alexander (2021). *Refactoring Guru*. Accessed: 2021-01-05. URL: https://refactoring.guru/.

Snyder, Carolyn (2003). *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces*. eng. The Morgan Kaufmann series in interactive technologies. Kidlington: Elsevier Science & Technology. ISBN: 9781558608702. URL: https://web.archive.org/web/20140628171628/http://www.paperprototyping.com/.

Spyridonos, Ploutarchos (Feb. 2010). *Software Requirements Specification for PDF Split and Merge, Version 2.1.0*. https://web.archive.org/web/20170225043950/http://selab.netlab.uky.edu/%7Eashlee/cs617/project2/PDFSam.pdf.

Stuart, Andy (2014). "Ground rules for a high performing team". In: *Paper presented at PMI® Global Congress 2014—North America, Phoenix, AZ. Newtown Square, PA: Project Management Institute.* Pp. 328–338.

Team, Data System (n.d.). *System Requirements Specification for STEWARDS*. https://web.archive.org/web/20200923200038/https://www.nrcs.usda.gov/Internet/FSE_DOCUMENTS/nrcs143_013173.pdf.

Tuckman, Bruce W (1965). "Developmental sequence in small groups." In: *Psychological bulletin* 63.6, p. 384.

Tuckman, Bruce W and Mary Ann C Jensen (1977). "Stages of small-group development revisited". In: *Group & Organization Studies* 2.4, pp. 419–427.

Usman, Muhammad et al. (2014). "Effort estimation in agile software development: a systematic literature review". In: *Proceedings of the 10th international conference on predictive models in software engineering*, pp. 82–91.

Van Wyngaard, C Jurie, Jan-Harm C Pretorius, and Leon Pretorius (2012). "Theory of the triple constraint—A conceptual review". In: *2012 IEEE International Conference on Industrial Engineering and Engineering Management*. IEEE, pp. 1991–1997.

Vorvoreanu, Mihaela et al. (2019). "From Gender Biases to Gender-Inclusive Design: An Empirical Investigation". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, pp. 1–14. ISBN: 9781450359702. DOI: 10.1145/3290605.3300283. URL: https://doi.org/10.1145/3290605.3300283.

Wake, Bill (Aug. 2003). *INVEST in Good Stories, and SMART Tasks*. https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/. Accessed: 2020-12-31.

Yang, Li-Ren, Chung-Fah Huang, and Kun-Shan Wu (2011). "The association among project manager's leadership style, teamwork and project success". In: *International journal of project management* 29.3, pp. 258–267.

Index