Software Engineering

According to Kinser

Table of Contents

Software Engineering

According to Kinser

Software Engineering

According to Kinser

Software Engineering

According to Kinser

Introduction

This book has been created based on the two Software Engineering courses taught at East Tennessee State University (ETSU) and is meant to be used as an additional source for the students. The courses were originally created as a combination of lecture and lab, using a flipped classroom modality where students are expected to read certain topic materials in advance of the lecture. The materials normally consist of a slide deck where the presenter's notes on each slide detailed what would be covered in a more traditional lecture style delivery. The slides themselves provide a summary or study guide outline of that lecture material. The in-class lecture itself was then abbreviated to hit the most important aspect and to tie the topics together for students. The lab is done as an in-class exercise that puts the topics into practice.

While the original course construction was designed to address as many learning styles as possible, feedback from some students pointed to a need for a more comprehensive source (instead of pointing them to multiple sources online where duplication and conflicting messaging made it more challenging to get the "big picture"). This book is meant to be that source and serves as an alternative to the presenter's notes in the slide deck. This book does not replace the slides as the slides include instructions and content specific to the classroom activities that this book does not address.

Software Engineering

According to Kinser

Chapter 1 – How the classes are set up

East Tennessee State University (ETSU) has two sequential courses for Software Engineering, CSCI 4250 and CSCI 4350. The first course is an introduction to classic concepts and processes. The second course delves deeper into these same concepts and processes while introducing modern extensions and variations and tools.

Both courses are required for all computing concentrations because they tie together the bulk of the topics from each of the concentrations, culminating in a semester-long project that puts into practice as many of the concepts and practices and processes as possible.

Because these two courses are required by all concentrations, they are not coding centric. They are about how to engineer software-based solutions. They address the systematic process that all projects (software based or otherwise) must follow. Recognizing that no single person can build a complex solution in a single semester, and everyone has a unique set of skills as well as a unique perspective, the two courses lead up to a semester long team project that enables students to apply the course topics and learn from each other as they build a real-world solution and deploy it into a production environment using modern tools and techniques.

The goal of the two courses is not to make every student into a project manager, or an architect, or a Scrum Master, etc. The goal is to instill in them an understanding of the big picture of how solutions are created, start to finish; showing them how many of the courses they have taken in pursuit of their degree fit together to make this picture a reality.

Software Engineering

According to Kinser

Software Engineering I (CSCI 4250)

The first course, CSCI 4250, introduces students to the Software Development Lifecycle (SDLC) and the most commonly used development models (Waterfall, Iterative/Incremental, Scrum). Project Management is overlaid on these topics as we address planning, estimation, dependencies, risk management, change management, and related topics.

Additional supporting topics are included to help round out some of the sharp edges. This includes Ethics and Legal aspects of SE and how they manifest themselves in the early stages of their careers. Softer skills like job searching, goal setting (e.g., SMART), team formation/dynamics (e.g., Tuckman's Model), process evolution (e.g., CMMI), and problem solving/brainstorming techniques are interspersed with the core lessons.

The semester culminates with a Scrum based project that runs for 4 sprints. Teams are self-formed and self-organized. Each team works on the same problem and is encouraged to collaborate within and across teams (and across sections if available).

To help make learning more persistent, there is a lot of repetition of core concepts and language throughout the semester. Supporting materials from external sources are provided to students. The students are also encouraged to find their own supporting materials and share that (some of which is eventually incorporated into future courses). Students are also encouraged to produce their own materials (often a paraphrase or summation of course material) so that future students see how others have connected the dots.

An example of repetition can be found in how attendance is taken. Each section has a GitHub repository setup at the start of the semester. Students must fork the repository such that they have a local copy connected back to the original. At the start of each class session, they are given an "attendance word/phrase" which they must add to the attendance file next to their student ID and then submit a pull request to update their change to the original. This is a simple introduction to source control so that when we do the in-class exercise on source control, they have some familiarity with GitHub and basic push/pull requests.

To address the most dominant learning styles, the course material is delivered using a variety of formats. For more visual learners, presentations include graphics, illustrations, and other visual aids. There is often a video before each class that either explains the topic from a different perspective or connects it to a seemingly unrelated but familiar topic (thus supporting memory mapping). The provided slide decks summarize salient points and can be used as a study guide.

Software Engineering

According to Kinser

For auditory learners, the lecture at the start of each class session ties together the main points and explains the more complex topics/concepts. The lecture often includes real-world examples of the topics. In addition, the presenter's notes and slides were written so that a text reader could process them.

For kinesthetic learners, almost every topic has an in-class exercise where the students perform some activity to put the concepts/process into practice in a fun way. Some exercises are done individually, others using small teams, others with large teams, and others with the entire class.

For students that learn through the action of reading or writing, every topic has required and optional reading materials. Each student is also required to submit weekly status reports where they summarize what they have learned, what they plan to learn in the next week, and any obstacles they face.

The SDLC phases and methodologies for SE include the Agile philosophy and how it is manifested into the Scrum process. To that end, the course was built to be an example of the Agile philosophy by allowing students to self-form and self-organize in each team exercise and the team project. All team exercises are done in class because face-to-face is the best way to communicate. There is a predictable cadence to the course (each class has required and optional reading, starts with a video, has a short lecture, and an in-class exercise followed by a quiz). There is a published course schedule/outline, but it can be adjusted if necessary. There are multiple opportunities for the exchange of ideas, where the students learn together and learn from each other. Students have the freedom (and the responsibility) to chart their own path to success in the class. By submitting status reports each week and receiving individualized feedback on each, students receive frequent feedback and can adjust their approach to the course. The status report format mimics the Scrum retrospective. The team exercises are done using a variety of team sizes so they experience the range Scrum teams can form in. Each exercise focuses on following and applying the topic, and less focused on the final outcome of the work.

Each student must attain 100% on 8 topic quizzes, which map directly to the learning outcomes for the course. They have 5 attempts available to them. All of the course materials are available to them at the outset of the course and they can attempt the topic quizzes anytime throughout the semester but must get 100% by the end. This allows them to work at their own pace and encourages them to use the Project Management topic and Agile principles as they define their own strategy for taking them. They are encouraged to make the first attempt without notes. If a second attempt is needed, they should study the questions missed, and then wait several days before trying again. If a third attempt is needed, they can ask for a hint and/or use their notes. If a fourth attempt is needed, better hints will be provided upon request. If the fifth and final attempt is

needed to obtain a 100%, it is recommended that the student work directly with the professor during office hours so that it is guaranteed they will get 100%. By requiring everyone to get 100%, the quizzes become another instrument of learning in addition to being an instrument of assessment.

Although not enforced, most ‘A’ students will be able to achieve 100% on most topic quizzes on their first attempt without use of notes or help. Most ‘A’ students will also attempt the quiz before or just as the topic is covered in class. Most ‘B’ students will be able to achieve 100% on most topic quizzes on their first or second attempt using notes. Most ‘B’ students will attempt the quiz soon after the topic is covered in class. Most ‘C’ students will achieve 100% by the 3rd or 4th attempt. Most ‘C’ students will attempt the quiz closer to the due date no matter when the topic is covered in class. Importantly, the quizzes are scored but not graded. It is how the students approach the quiz combined with their performance that is evaluated.

To keep the course current and interesting, the in-class exercises, the videos, and quiz questions are refreshed/replaced each semester based on past semester’s performance and/or student feedback.

The final scrum project spans semesters. Each new set of students in 4250 will build on and extend the previous semester’s project. That is, every semester, all the scrum teams will work on an application called ScavengerRUs for which exists an extremely robust set of requirements, already broken down into a series of major (epic) releases. After the semester is over, one of the teams’ final releases will be selected as the baseline for the next semester including all of the supporting materials: documentation, scripts, diagrams, etc. The goal of this recurring project is to expose students to the most likely real-world situation where they must build solutions based on an existing solution.

As a result, students have a much better appreciation of the importance of “good” documentation and “good” comments in the code when they have to build on top of an existing baseline that they didn’t build.

Grading in this course is done using a contract labor approach. Students are given a job description that defines the expectations of their performance during the semester (e.g., attendance, required reading, quiz performance, etc.). Their grade is based on their ability to perform to these expectations:

- A = consistently exceeding expectations defined in the job description
- B = occasionally exceeding expectations defined in the job description
- C = meets all expectations defined in the job description
- D = occasionally fails to meet expectations defined in the job description
- F = consistently fails to meet expectations defined in the job description

Software Engineering

According to Kinser

When students are able to exceed expectations that are associated with learning objectives and/or SE topics, demonstrating a deeper level of understanding, these actions are of more value, have greater weight, than activities that require less effort. For example, it is expected that a student will be on time for each class. Arriving early is a form of exceeding expectations. It is expected that a student achieves 100% on the topic quizzes. Achieving 100% on the first attempt without using notes or resources is a form of exceeding expectations. The quiz example is of much greater value than being early to class.

Each student must write a self-evaluation essay outlining their performance to the job description twice during the course. The first essay is around the mid-term timeframe. The last essay is done at the end of the semester. The first essay provides them an opportunity to get feedback so that their final essay is well-formed and targeted. This approach models how most companies do performance evaluations.

Software Engineering

According to Kinser

Software Engineering II (CSCI 4350)

The second course, CSCI 4350, is a senior level course that goes into more depth on the same topics from CSCI 4250 while also introducing more modern topics such as Microservices, Emergent Architectures, Internet of Things (IoT), DevOps, and CI/CD.

Keeping in mind that 4350 is required by all Computing concentrations, coding is not emphasized as the outcome that is measured. The students are expected to actively participate in a team project that runs the duration of the semester. In the project they will be given a Vision statement (one or two sentences in length) and must generate a product backlog of requirements in the form of user stories and then run 5 two-week long sprints using the Scrum methodology and Agile Principles.

In addition to the project, each student must achieve 100% on 7 learning outcome quizzes. Each quiz maps directly to one of the learning objectives for the course. The quizzes in SE II are more difficult than the ones in SE I.

Every week students must submit a progress report where they outline their work on the project, how the team adhered to the ideal of the process and how they deviated from it (and why), what they have learned, and summarize the topics covered in class.

Finally, all students must produce two oral presentations on SE related topics. They can choose from a list of topics provided or propose their own. The presentations must have an introduction, body, and conclusion format, and cite sources. The better presentations will be done with a voice of confidence and authority, including some personal insights or examples about the topics. The content of the better presentations will include graphics, animations, and/or embedded videos to support their topic. The use of appropriate humor or other content to make the presentations entertaining as well as educational, is encouraged.

The CSCI 4350 course structure is a continuation of what was used in CSCI 4250 with a few exceptions. There are no in-class quizzes. Because the team project runs the entire semester, all of the in-class exercises and lectures are defined and ordered to support the natural evolution of the project. The semester project is new each semester and, as stated earlier, students start from scratch creating their own requirements and ultimately deploying the solution in a production environment.

Upon completion of CSCI 4250, students have an awareness and surface level experience with SDLC, Agile, and Scrum. By the end of CSCI 4350, their experience with these same topics has moved in the direction of mastery while also getting awareness and surface level experience with the modern topics introduced in the second course.

Software Engineering

According to Kinser

Software Engineering

According to Kinser

CSCI 4250

Software Engineering

According to Kinser

Chapter 2 - Legal and Ethics

From Bing/ChatGPT:

> As a software engineering professional, it is important to adhere to ethical and legal standards in order to ensure the safety and well-being of the public. The **IEEE Computer Society** and the **Association for Computing Machinery (ACM)** have established a **Code of Ethics and Professional Practice** that outlines the principles that software engineers should follow [1]. These principles include acting in the public interest, ensuring that products meet the highest professional standards, maintaining integrity and independence in professional judgment, and advancing the integrity and reputation of the profession [1]. In addition to ethical considerations, software engineers must also be aware of legal issues related to software development, such as intellectual property rights and data privacy laws [2]. Finally, software engineers should exhibit professional behaviors such as effective communication, collaboration, and continuous learning to ensure that they are able to work effectively with others and stay up-to-date with the latest developments in the field [3].
>
> Source: Conversation with Bing, 12/21/2023
> (1) Code of Ethics for Software Engineers - IEEE Computer Society. https://www.computer.org/education/code-of-ethics.
> (2) Ethics in Software Engineering: A Key Component of Professional Practice. https://pdh-pro.com/pe-resources/ethics-in-software-engineering/.
> (3) Ethics in Software Engineering | ICS 314, Spring 2021. http://courses.ics.hawaii.edu/ics314s21/modules/ethics/.
> (4) Software engineering code of ethics is approved - ACM Digital Library. https://dl.acm.org/doi/pdf/10.1145/317665.317682.

The above quote starts a conversation about whether it's ethical to leverage content generated by AI in any work you produce.

> Is it ethical to use code generated by AI?
> Is it ethical to ask AI to review your code?
> Is it ethical to ask AI to help you with work or with this class?

Like with any other source, if you don't claim it as your own, if you cite your sources, then you are probably okay. AI will be a revolution of productivity much like the industrial revolution and the digital revolution. Our job is to use it in an ethical way.

You may think that as a junior engineer that you won't have any ethical dilemmas to resolve because you won't be in a position of authority. I can tell you from personal experience, that you will be asked to do things that seem reasonable but chip away at your ethics. Some examples:

> Pressure to revise an estimate you provided to something not possible
> Pressure to cut corners (usually to hit a date that wasn't realistic)

Software Engineering

According to Kinser

Pressure to commit to complete a set of tasks within a timeline you don't agree with
Pressure to declare something complete when it isn't
    This may include falsifying test results or hiding test results
Turning a blind eye to misuse of personal data, bias in algorithms, etc.
Installing a backdoor (for all the right reasons, I'm sure)
Witnessing harassment or illegal activities at work

With the examples above, you may feel compelled to do something you know isn't right because there is an implication that you could lose your job or somehow be held back in your career if you don't. It probably won't be stated in obvious terms. It may only be your imagination. And this is where the dilemma comes into play. If you resist, will it hurt your career? If you refuse, will they fire you? Is this really important enough of an ethical violation to risk it? Or is it all in your head and if you were to push back, everyone will agree with you and thank you for pointing out the issue?

Then there are dilemmas you place on yourself like, how perfect do you make your code before declaring it done. If the solution you are responsible for passes all the tests and would be considered "done", is it ethical for you to keep working on it to make it better, more fault tolerant, more extendable, more scalable? Wouldn't that be a good thing? What if you finished something ahead of schedule, is it ethical to relax for an hour or two and play a game on your phone? After all, you had to work a couple of extra hours (over your 40) a few weeks ago with no compensation.

What about legal ramifications of what you do at work or how your work product is used? Some examples of legal dilemmas you might face include:

Using pirated software (ostensibly to save money)
Using third party products or libraries or scripts without approval
Falsifying reports to your superiors or your clients (to avoid conflict)
Initiating DOS attacks from your work computer (really happens)
Discriminate or harass you coworkers ("I was just trying to be funny")
Threatening others with violence ("I didn't really mean it. It's a figure of speech")

It sounds a little scary, but when you accept a job at a company, you are responsible for your actions (behavior and speech) and inactions. Some of these can be the right thing to do but hurt your career at that company. Some of them will be the right thing and help your career. Some of them will be the wrong thing to do and could get you into legal trouble personally, or just get you fired. Welcome to the world of being a professional.

Software Engineering

According to Kinser

Chapter 3 - Jobs

While the unknowns of the “real world” can be intimidating and anxiety inducing, CSCI majors are better positioned to take them on than many others because we deal with the unknown all the time. When writing code or setting up a new network or tracking down security vulnerabilities, we often don’t know what the solution is or even if there is a solution, but we use tools and processes that help us break problems down into smaller more manageable challenges and we gain confidence in ourselves with each success.

Job Hunting

In preparation for your job search, critique your on-line persona. What will a hiring manager think about your Facebook account, twitter feeds, etc. Some companies do ask for this as part of the application process though it is more common with very senior positions. Either way, be aware of your digital self. You can change it to be more “socially acceptable” or leave it as-is. Just realize that anything that is publicly available is fair game in the interview process.

The easiest time to find a job is when you already have one, because you don’t have to explain why you don’t currently have a job, there is no gap in your resume timeline, and you don’t have to stress about having no money coming in. So, if you have a job offer(s) but it's not the perfect job you want, consider taking it for the above reasons. You might like the current one more than you expect, or you might find an opportunity within the company that wasn't advertised, that is perfect for you. If neither of those happen, look for another job while you keep the job. You are under no obligation to stay a certain amount of time in your first job.

If you don't have a job, work at finding one 40 hours a week. Establish a schedule, e.g. 9-5 M-F, during which time you search for job opportunities. Be organized about finding a job: track the ones you applied to, schedule follow ups, send thankyous, track and summarize interviews, etc.
Work full-time to find a job and you'll find one faster. By working full-time to find a job, it will probably help you relax during your "off-time" knowing that you are doing everything you can to find a job.

One of the best ways to find a job is to know what you want to do and what you don’t want to do and what you won’t do. What part of your degree program did you enjoy the most? What part were you really good at? These can be a good starting point to define what you want to do. Are you okay with relocation, working nights, travel (if so, how much), working in a

big team or a small team, etc. Knowing what you don't like is as important as knowing what you do like. Are there things you won't do? Are you okay with building weapons, working for politically driven organizations, etc.

Elevator Pitch

During your job search, you will have to introduce yourself repeatedly. You may be familiar with the old saying, "you only have one chance to make a first impression." Having a well thought out and practiced introductory statement will help make a good first impression on prospective employers. An elevator pitch is a short description of yourself and what you are looking for. The elevator pitch should only take the time of a typical elevator ride; 30 seconds. For example:

> I'm a computer programmer with 3 years of industry experience looking for back-end server development using Java, C# or C++. I am versed in SQL and Web Services. I am willing to relocate and/or travel.

As part of your job search, you can use your personal contacts; including teachers, counselors, friends who have a job, friends of your parents, parents of your friends, etc. These people make up your personal network. Leveraging them to help you find a job is called "networking". Let everyone know you are looking for a job by practicing your elevator pitch on them. They will be more forgiving of your delivery. By the time you give your elevator pitch in an interview, you will have more confidence in your delivery.

You can use the keywords from your elevator pitch when you search for job ads that might fit your goals.

Interviewing

Normally, except for boutique shops or small companies, you will have to endure a series of phone and/or video interviews before you get the "real" one which will result in a Yes decision. The first interview is almost a courtesy interview done by a human resources (HR) representative to see if you are indeed a human being, are still interested, still available, are able to articulate your resume, etc. The HR person can answer company questions like purpose, benefits, organization hierarchy, and general expectations of the position (like showing up on time). The HR interview is often the precursor to scheduling an interview with someone that can evaluate your fitness for the position being sought.

If your interview is on-site, you can ask whoever schedules it what is appropriate attire for their office and then do one nicer. If they say jeans are fine; wear nice, clean jeans

with a dress shirt and sports coat, or equivalent. If they say business casual, wear dress slacks, dress shirt and tie and jacket; or the equivalent.

The next interview (which could be immediately after the HR one but most often occurs a week or two later) is with a technical person. They can usually tell you technical stuff like what tools they use, what methodologies they use, but they often can't tell you specifics about your prospective position. Most of the interviewers are what are often referred to as Gatekeepers. A Gatekeeper is someone that can recommend "yes" or "no" but they can't actually make the decision to hire you or not. The person that can make the decision is often called the Hiring Manager.

It's possible there would be additional technical interviews or "culture fit" interviews but often the next one is the one with the person or persons deciding whether to hire you. This could include more technical people as well. The person that can decide to hire you is often referred to as the hiring manager. This may or may not be the person that will be your manager if you are hired.

If you get an interview, do your research about the company. You've probably heard that one. What does it mean? Just like you would study before taking a test, you should spend some time preparing before you go to the interview. What is the company's main business/market? How does it relate to the job you are applying for? Where are their offices? How long have they been around? Are they a leader in their industry? What other jobs are they looking to fill? Have they had any major press releases? All of these questions give you fodder for questions to ask during the interview. And, most of the answers to these questions can be found on the company's website.

The best interviewers ask open-ended questions: "tell me about yourself". An open-ended question is one that can't be satisfactorily answered with a simple yes or no response. Give descriptive but not too long an answer. They probably have several questions to ask. Be respectful of their time. You can always end your response with something like, "I'm happy to provide more detail if you would like more" or "Would you like me to elaborate on any part of that".

You should ask open-ended questions too: "how would you describe your ideal candidate for this job", "why did you decide to work for this company", "can you describe your experience working here". It's good have a binder with some notes in it about the job and company to help remind you (since you are probably researching more than one company) and a list of questions to ask so you don't forget. It is recommended to have at least 5 questions ready in priority order. You may not get a chance to ask all of them, but you will be prepared if you do.

Things not to ask: "How much does this pay?", "What happened to the person that had this job before?", "How's the morale at your company?", "When would I get my first raise or promotion?". They will tell you what the job pays if they make you an offer. The last person is no longer there. They aren't going to say bad things about the person or the company. The morale is always good (companies are not going to let disgruntled employees interview candidates). When you get your raise/promotion depends on your performance and you don't have the job yet so there is no way to answer this question.

Don't let interviewers intimidate you. The people that conduct interviews typically have a full-time job that does not include doing interviews so be respectful of their time. While a certain amount of deference is appropriate, remember that the interviewer is just another person; no better than you, no worse than you; they have flaws and weaknesses, abilities and strengths. Oftentimes, they have not read your resume prior to sitting down with you (because they have a full-time job already). So, be patient and give them time to formulate their thoughts. Even if they have read it in advance, they aren't likely to work hard at understanding you or your skills from the resume (they probably won't look at your GitHub projects or sample code). In both scenarios, the interviewer is likely going to ask you questions that are answered in the resume. Politely answer the question and point back to the part of the resume that is relevant.

Always be polite, even to the janitor. I was waiting for an interview once when someone was coming to the door with their hands full, so I jumped up and opened the door for them. Turned out to be a future co-worker who saw me going to the interview later and told them what I did. Score!

Pretty much the best way to nail the interview is to do all the things they taught us in kindergarten (be nice, be honest, be polite) plus a few grownup things like be concise and on time. You don't have to have the right answer for every question in order to win the job. It's best to be honest. I've found that the people that like to conduct interviews also like to talk. The more you let them talk, the more positive they will feel about you.

Employers are looking for hard-working, self-starters that can learn quickly and are interested in success. If you can describe yourself in those terms (with some examples), you'll do fine in interviews.

The Offer

If the company decides to hire you, an HR person will contact you and give you the offer. Most companies will do this over the phone followed up with a written copy of the offer. They may even do it at the end of the interview process while you are still in-person, though this is rare. The reason they verbally give you the offer first is to verify you will accept before they go through the formal written process. If you verbally accept, you can still change your mind. It's not like an oral contract that binds you to them. It just means there weren't any objections at the time.

An offer is more than just the salary they intend to pay you. The total compensation package can include benefits (health insurance, dental insurance, vision insurance, etc.), time off (vacation time, sick leave, holidays, etc.), work flexibility (remote work, flexible start/end times each day, etc.), stock options (rare for new graduates but can happen), 401K matching, etc. etc.

To compare two offers is difficult because you can't really just look at salary as the only component. It is a major component. It is not the only component.

Speaking of salaries, it is good to know when the raise cycle occurs because if you are hired half way through the cycle, you will only have been there around six months when raises are offered which means you probably won't get any raise. A lot of companies operate with the unwritten rule that employees don't get raises in their first year because it takes that long before they really start to add value beyond their original salary. Raises are given (ideally) to people that add more value to the business than they did the year before and so their compensation should increase. So, if you are coming in mid-cycle it will be 18 months before you get a raise.

If you feel you are in a strong position, you can counter the offer. Most new graduates are not in this type of position. Does it hurt to counter? It depends on how you do it and what you counter with. If you ask for an extra thousand or two, they won't be offended. They might or might not agree depending on the need, their evaluation of you, where your salary currently falls in the range for that position, etc. If you laugh at their offer and say you want a 20% or more increase, they probably will be offended and might even rescind the original offer (because you just said you won't be happy there so why would they want you to come on board).

If they do include stock options, they can't change the vesting schedule but they might be open to increasing the number of shares. Again, not by a lot so don't ask for twice the number of shares.

Onboarding

Most big companies have an onboarding process that includes paperwork, tours, introductions, meetings, desk/computer setup, etc. This can take up the first week or so (I was in training for 3 weeks at one company, boring). Take good notes and be respectful. You probably won't remember the people you meet, or what was discussed in meetings. Some of these activities are as much for the rest of the organization as it is for you. For example, the people you meet now know who you are (they only have to learn one new name and face so it is more likely they will remember you than you remembering them). Small companies may do all this in a day or two.

You probably won't start coding on day 1 or even week 2. You won't be expected to know much about the company, the project, the code, etc. They hired you because they believe you are a hardworking, self-starter that learns quickly and is interested in success. They won't set you up to fail, it serves no one's interests.

Software Engineering

According to Kinser

It can cost a company ~$30K to hire you. There is lots of paperwork and hours of effort on the part of multiple people. They must file forms with the government (to pay your taxes, pay unemployment insurance, etc.). They must set up your 401K and enroll you in all the benefits.

It can cost a company ~$50K to fire or downsize someone. Any company with a dedicated HR department will want to do the necessary paperwork and steps to protect the company against possible retaliation or lawsuits. They must undo all the things they set up when you were hired. They must process all your equipment (e.g. cleansing your company computer after archiving all your files and emails). They often offer a severance (e.g. 2 weeks of pay is pretty common, but it could be more if you've been there a long time). Typically, your benefits are good to the end of the month, which is an expense for the company.

It can cost a company ~$20K to lose someone that leaves on their own. Not as expensive as letting them go but many of the same costs come into play.

To bring you on in a way that makes you unsuccessful does not help the company. Mistreating new hires hurts retention and morale. Morale is something that is hard to quantify but imagine if a group of 20 employees are no longer motivated to do their best work. The cost to the business can be very high.

Sometimes they don't even know which team you're going to be on when you start. If they aren't well organized, take it as a sign that you can make a positive impact. Clearly they have more work to be done than people to do it and they need you.

Day in the life

Some companies build software products they sell and support. These jobs usually have you doing bug fixes and enhancements to the product when you first start.

Some companies sell a service based on their software (e.g. Google) which will look to you a lot like the ones with a software product.

Some companies build software on contract for other companies. In this case, every project will be like a new adventure full of mystery and intrigue. As a new employee, they will most likely add you to an existing project that is relatively stable and have you pair up with a mentor on that project.

Some companies do contracting which is similar to the above company except the employees are contracted out to their clients. Contractors are staff augmentation, like a temporary worker. Some clients will treat contractors as if they were employees, others keep you at a distance (e.g. won't invite you to company events or even company meetings). A good contracting company would send you to training the first few weeks and have you shadow an existing employee on one of their client engagements (sometimes at no cost to that client) until you are ready to contract on your own.

Some companies do consulting work which is similar to contracting except you are expected to be an "expert" in some topic or field. In these jobs, you are expected to tell the client what needs to be done and then do it better than they could. This type of job is usually reserved for senior people with lots of experience and who are able to adapt quickly.

From the above, you probably get the idea that entry level people spend most of their time coding/testing/maintaining. You usually get a list of tasks (or can pick from a list) in priority order. Tasks may or may not have a time estimate associated with them. There are always more tasks than can be done. You'll probably get lots of interruptions (meetings, visitors, emails). After you settle in, you are expected to be self-managed on a day-to-day basis, maybe even week-to-week.

How to succeed

With great power comes great responsibility.

As an entry level person, you have zero power and probably no responsibilities other than to show up on time (which means …. early) and do what the leads ask you to do. The team won't likely know you or what you are capable of. They probably were getting things done before you showed up so they may view your arrival as an obstacle. It's up to you to overcome all this and demonstrate your value and skills.

One way to succeed is to submit status reports weekly to your immediate supervisor (whether you are asked to do this or not). A status report typically outlines:

- What you accomplished since last report
- What obstacles or issues you are having
- What you plan to do going forward

By submitting this to your immediate supervisor weekly, you create the opportunity for them to give you frequent feedback, course corrections, etc. Listen to what they are interested in knowing and provide that information proactively. Adjust your content in quality and quantity based on their feedback. By doing these status reports, you keep your name in front of them in a manner you have control over. Remember, if you don't practice some level of self-promotion, who is going to do that for you? Also, giving this information to your boss makes their job easier and that is always a good thing.

The status reports serve other purposes too. It can help you stay organized because you are listing what your obstacles are and what you plan to do next. It is a kind of to-do list you can check off throughout the week. Annually, the company will do a performance review and most companies ask employees to do a self-review as part of this activity. You can concatenate all your status reports and use the accomplishment sections to find the most relevant accomplishments throughout the year, so your review is accurate and comprehensive. Most people can remember the last 3 months of accomplishments with reasonable clarity. Very few people can remember a year's work of accomplishments without some kind of tracking system.

If you know Scrum, you'll recognize the status report content as the format for the daily standups. That's because this approach works.

You will be doing weekly status reports in this class. If you are a full-time student you will report on your activities across all of your classes, not just this one. If you also have a job, you will report on your classes and your job activities. This will help you stay organized, avoid losing track of assignments, and track your progress for the semester. I provide feedback on the status report each week for each student so that you get regular updates on your performance and any corrective actions you should consider.

Raises and Promotions

A salary band defines the lower and upper bound of pay for a position. They are usually divided into 3 sub-bands; entry, average, expert. Someone in the entry sub-band is usually new to the position and may not be fulfilling all of the job description on a consistent basis. Employees in the average sub-band are consistently meeting and occasionally exceeding expectations of the job description. The expert sub-band is for people that consistently exceed expectations.

Sequential positions in a company (e.g. Software Engineer and Senior Software Engineer) have salary bands that overlap. Most of the time, the upper sub-band (expert) of one position will overlap with the lower sub-band (entry) of the next position. This is not a hard rule. Overlapping is very common. How much the positions overlap can vary by company.

Most companies hand out raises once a year based on their fiscal year which are commonly aligned with the calendar but not always. Large companies may start the review process in October/November but not have the raise take effect until January or February. A few companies do a review based on your hire date. These are probably small companies only. Imagine trying to manage that approach for 2,000 plus employees.

Most use a bell curve (but won't admit it). Each company will determine what the medium raise will be times their labor costs which gives them their "raise pool". This is the amount of money the company will distribute for raises. The bell curve describes

how they distribute the raise pool among the staff (a few people get big raises, lots get medium raises, and a few get no raises). Most companies don't give raises to people with less than a year with the company because a raise is only deserved if the person adds more value to the business than was expected based on their salary. Most new hires don't add as much value as they are being paid until they have been there several months.

A medium raise is often ~= rate of inflation plus 1% (maybe 2). So, if inflation is running at 2%, the medium raise is 3%. Who gets raises is somewhat of a popularity contest. It is supposed to be based on value added to the business but this is hard to quantify so it is normally determined based on the boss' perception of value added using a ranking system; If Bob is perceived to add more value than Nancy who adds more value than Jerry the new hire, then Bob will get an above medium raise, Nancy gets a raise close to the medium and Jerry gets nothing.

The salary bands can come into play with raises. If you are in the entry sub-band but performing like an expert, you can get a large raise. If you are already in the expert sub-band and performing like an expert, you may not get much of a raise at all because they won't increase your pay above the upper bound of the salary band. People that can't get promotions at this point often become demoralized and "retire in place", RIP.

Some people take the position that they won't work harder unless they get a raise. They expect the company to go first (which is what happens when you first get hired). Other people work harder in hopes they will be recognized for the effort and rewarded with a raise or promotion. As a boss, I was never a fan of the people that expected the company to go first.

Key to getting good raises:

- make your boss' job easier
  - don't make them come to you to find out what's going on;
  - don't make them ask more than once for something;
  - try to anticipate their expectations by what they ask for the most often
- do what no one else wants to do (the not fun jobs)
- know your job description and the job description for the next level up (or two) then work like you have the next level job.
  - doing your assignments (your job) means you get to keep your job
  - you have to do more in order to get more (faster or better or more of it)
- you are either ok with what you get paid or you aren't.
  - if you aren't then start looking for another job (don't complain about it)

Promotions work very much like raises with a few differences. The money for promotions is usually calculated separately from the raise pool. Raises may occur at the same time as promotion but it doesn't have to be done that way. Some companies give promotions based on time served (this is a holdover from the era where you worked at the same company your whole life but is still common in government service or the military).

The salary band comes into play with promotions as well. If you are in the entry sub-band, it will be nearly impossible to get a promotion to the next position because your

current salary is so far below the lower sub-band of the next position. Getting a promotion when you are in the upper sub-band might not result in much of a pay increase because you are already in the lower sub-band of the new position.

The best time to get a promotion is when you are in the middle sub-band. Your salary is close to but not in the next salary band. Also, you are moving to a new salary band before you start to top out in your current band (remember the RIP scenario). The jump to the next band will be noticeable if not significant but doable from HR's perspective.

The best way to get promoted is to do the work for the job you want before you get the title. This is why I recommend you get a copy of your job description and that of the position above yours. When you get a promotion, the best reaction from your co-workers is "I thought you already were X". That means they believe you deserve it and won't begrudge you for it.

Sometimes the promotion can be to a job that didn't exist until you created it by being you. Somebody invented the role of scrum master by being a scrum master before the term existed. So don't get too fixated on existing titles in the company especially if it is a growing company.

Blue Boat

Personal storytime. Shortly after starting my first job after college, I was selected at random along with a few other new hires, to have lunch with some vice president. During lunch he tried to convince us that money is not a motivator and that we should work hard regardless of compensation.

I grew up very poor (free lunch and free breakfast programs, all my clothes were used, vegetarian not by choice). Being fresh out of school with an apartment without furnishing, loans to pay, a car on the brink (actually I was riding my bike to work because of the car situation), etc.; money was very much a motivator to me. So, I asked the VP, "what color is your boat?" He replied, "I don't understand what that has to do with anything." I said, "just humor me please. What color is your boat?" He answered, "It's blue." I asked the person next to me what color his boat was. He didn't have a boat. I asked another person. Same answer. I asked the last person. They didn't have a boat either.

I finally said to the VP, "I think that when I get a blue boat, money will not be a motivator to me, but right now I need enough money to buy a blue boat." The lunch ended somewhat abruptly. I was never invited to lunch by any other executives.

Savings

If you ever want to retire, you need to start saving as soon as possible. Don't save what you can, save as much as you are allowed to. A general rule of thumb is that you can

retire when you can live off of 4% of your capital assets. For example, if you can live off of $80K, you need $2M in savings that's producing at least 4% annually.

How long will it take to save $2M? If you are 20 years old and can save $639/mo for 47 years (lots of assumptions involved), you can save $2M and retire at age 67. If you wait until you are 30 years old to start, you will have to save almost TWICE as much each month to retire at 67.

What happens in 47 years? Prices go up. In 2022 the poverty line was $18K, median income was $54K, and upper middle class was >$100K. Looking back 47 years ago, median income was $15K (basically today's poverty level). So, $54K will likely be the poverty line in 47 years. If you were planning on living on $80K in 47 years, you will be just above the poverty line.

Enroll in your 401K immediately if available. Contribute as much as you can so that at a minimum you maximize the companies matching contributions upfront. A 401K match accelerates your savings (by 50-100%) and then you get the compounding gains on both your contribution and the matching each year.

Several financial advisers out there will tell you, and this is backed up by my own experience as well as several people I know, you won't miss the money you never saw as much as the money you have in your hand. That is to say, if you have money taken directly out of your check and put into savings (like 401Ks or stock purchase plans) you won't miss it as much as if you get the money in your account and have to decide how much you can "afford" to save.

Goals (Be SMART)

Why should you set goals? Life is full of choices. One tool for helping you make those choices is to set goals. I prefer the 5 1 90 goal process. You set a somewhat vague 5-year goal. Like with problem solving, you break this goal down into smaller goals. You set a smaller but more specific goal of what you need to do in 1 year to be on track to achieve the 5-year goal. Then you break the 1-year goal down into smaller but very specific goals for the next 90 days that you need to achieve to be on track for the 1-year goal. You then revisit these goals every 90 days, adjusting your 90 day and 1-year goals to stay on track with the 5-year goal.

Example of a professional goal for an entry level programmer at a new company:

- 5 year goal : Become a team leader
- 1 year goal : Be fluent in the business domain of the company, needs/goals of the department and my team
- 90 day goal : Become self-sufficient with my day-to-day tasks and become fluent in the needs/goals of my team

Notice that the goals are not specific to the job duties. They build on each other to reach the team leader goal. Thus, when new programmer assignments are available, you can sign up for the ones that give you more exposure to the needs of the team and/or

department.

An example a senior student might have:

5 year goal: Have at least $20K in savings
1 year goal: Graduate college and get a job
90 day milestone: Earn at least a 3.5 GPA this semester

How do you write the goal? It should be a SMART goal. SMART goals were developed by George Doran, Arthur Miller and James Cunningham in their 1981 article “There's a S.M.A.R.T. way to write management goals and objectives” . In the examples below the second goal is more aligned with SMART than the first:

Specific - Do better in School vs. Increase semester GPA over last current average
Measurable - Study better vs. Dedicate at least 1 hour each day (Mon-Sat) to studying
Achievable - Give all my money to charity vs. Give 10% of my take home pay to charity
Relevant - Achieve highest ranking in video game vs. graduate college with a 3.0+
Timely - Be rich someday vs. secure a full time job within 30 days of graduating

Goals should be positive in their direction. For example, if you are trying to reduce being tardy, instead of having a goal “to never be late to class”, you should have the goal “to be at least 5 minutes early to class”.
https://www.accaglobal.com/ca/en/member/discover/cpd-articles/leadership-management/yeung-nov17cpd.html

So, what would be your lifetime goal? What do you want to achieve in life? Everyone dies. Not everyone dies content with their life choices.

Have a family and be a good spouse/parent?
Make a significant impact on the community around you?
Own your own company?

In high school, I defined my success as "never be poor again in my life, only marry for love". The first one is a negative goal but I was in high school and hadn’t learned the best ways to set goals.
As time went by I added "Do good, Be happy". That is one reason why I teach.

Chapter 4 - Problem Solving

How do you solve problems?

https://asq.org/quality-resources/problem-solving

Software Engineering

According to Kinser

Define the problem
Identify possible solutions
Evaluate to find the best one
Test to verify it solves the problem

Is this how you solve problems?

According to Kinser: If a problem has an obvious list of possible solutions it is a task to be completed, not a problem to be solved. A problem is something without a clear solution.

Rather than defining the problem, I want to understand why it's a problem. Sometimes a problem is stated poorly. Understanding why it's a problem can help ensure the solution that is developed is actually addressing the real issues.

What is the goal that the problem is preventing us from reaching? Somewhat related to understanding why it's a problem, knowing the goal helps us as we encounter choices or alternative paths so that we make the best choices.

Who is involved and what resources are available? If you had to build a house all by yourself, you probably wouldn't use the same process, tools, etc. as you would if you had 20 people available full-time. Knowing the resources available gives you an understanding of the constraints you are facing in solving the problem.

What has been tried already? Knowing what was tried before and why those attempts didn't work saves time and potentially narrows down the options to choose from. If someone says your proposed idea won't work because they tried that before, make sure you ask clarifying questions to understand if what they tried really is the same and if key factors haven't changed since that attempt. For example, we need to get a couch out of the living room so we suggest taking out the front door to which someone says that's already been tried and didn't work. They neglect to say it was a different couch, or the door has since been replaced with a wider one, or it was a different house altogether.

Is it similar to a problem previously solved? If a similar problem has been solved, knowing what the solution was and how well it worked could give clues as to what solution might work for the current problem. We want to build a website that sells dog food and dog related products. We know that someone built a retail site before. How did they do it? How do the items sold in that situation compare to the ones we are selling? Is there a difference in customer profiles that might impact how we design the site?

Understanding a problem can be a challenge in software engineering. Some of the problems we have to solve are complex and possibly ever changing. If I ask you to write

code to loop 100 times and print out the counter each time. This isn't a problem because the solutions are readily available. If I ask you to write code that can drive a car from downtown LA to downtown NYC, the solutions are not obvious. To develop the solution, you have to ask enough questions until you feel you understand the problem.

Break the problem down into a set of smaller more solvable parts. The union of the set must be equal to the original problem. Notice I say, "more solvable". When you break down a complex problem into smaller parts, those parts by themselves may not be solvable either but they are more solvable than the original. Any of the parts that have an obvious solution become tasks to execute, while the others need to be broken down further. You keep repeating this cycle until all the parts are tasks.

In the self-driving car problem, we can break it down to navigation and vehicle control. That is to say, what is the route a regular car would use to get from LA to NYC (Navigation)? There are several solutions that will define the path already in existence (Google Map, MapQuest, Garmen, etc.). Controlling the car involves steering, accelerating/braking, understanding road signs and regulations, ability to "See" the road and other objects in the road, etc.
Cruise control is a way a car can automatically accelerate and brake in order to maintain a specified speed. Cruise control is a part of the controlling problem that has a known solution, so it is a task. "Seeing" the road and other objects does not have a known solution.

A quick peek ahead at the software development lifecycle (SDLC): In Software Engineering, problems are documented as Requirements (representing What needs to be done). Requirements analysis breaks these down until they are solvable, meaning we understand them well enough to identify a solution. The possible solutions are documented as a design (representing How it needs to be done). Thus, designs are the solution to the requirements. Design can be high level designs which are broken down into detailed designs. Detailed designs outline the tasks that need to be done as part of the Implementation. Implementation is the solution to the Design. The full implementation is usually the solution. It still needs to be tested to verify it really solves the problem. Testing is the unwinding of the recursive path that got us here.

Project Management is the process organizations follow to solve software engineering problems. The specifics of the process taken will vary based on the business domain, problem type, technology available, timeline available, budget available, quality desired, etc. Project Management is problem solving that focuses on the three pillars of any endeavor: schedule, cost, and scope. We'll explore Project Management in greater detail in a later chapter.

Basic problem solving skills are instrumental in almost every aspect of Software Engineering. The better you understand how you personally approach and solve problems, the better Software Engineer you will be.

Software Engineering

According to Kinser

Chapter 5 - Project Management

Before we get into Project Management, I want to introduce a team formation concept commonly referred to as Tuckman's Model.

https://en.wikipedia.org/wiki/Tuckman%27s_stages_of_group_development
Primarily Sections 1.1-1.4 on the Tuckman link are REQUIRED reading.

The idea is that whenever teams are formed, they go through a predictable sequence of states/phases. This is true whether it is a sports team or software Dev Team. When the team is first formed there is usually a goal or purpose behind the creation of the team, and this gives everyone something in common and something to strive for. It creates a sense of excitement and positive energy. This state is called the Forming Phase. Everyone is unusually upbeat and very polite to one another. They are often a bit guarded about sharing personal information. Sometimes more dominant members will step forward to assume leadership positions whether formal or informal. The rest of the group tends to follow along.

As time goes by and as obstacles to progress are encountered, the team enters the Storming Phase. Personalities start to clash. The team starts to question those in leadership positions. Quieter members start to voice their opinions and concerns because things are not going in the direction they like. While this state may feel negative because of the conflicts, it is necessary in order to establish trust within the team. The mistrust that led to Storming was there all along and the only way to replace it with trust is to communicate concerns. The more constructive and professional the team can do this, the faster they can emerge from Storming. If they bicker and take passive aggressive positions, they may never get out of Storming. Sometimes the conflict may get too extreme, and the teams disband. Sometimes the conflict makes enough team members uncomfortable that they "put on a happy face" and the team feels like it is making progress, but they really just went back to Forming.

With guidance and a little bit of luck (actually, purposeful perseverance) the team can advance to the Norming Phase. This is where most well-functioning teams land and, more often than not, will stay. In this phase, there is more trust and acceptance of others as individuals. The team has learned what each other's strengths and weaknesses are, and leverage these to make the whole team successful.

But there is another state, the Performing Phase. In this phase, each team member has learned to value each other and invest in each other's success because they genuinely care about each other in a way that both serves the team's goal/purpose but also goes beyond it. Very few teams in my experience reach this state. Those that do almost always had a strong outside mentor or leader helping them through the transitions.

One of the really interesting things about Tuckman's model, in my opinion, is that it is really a discovery, not an invention. These states occur whether you know about the

model or not. They just are. Also, you can't force a team to the Performing Phase. It has to occur organically. They have to have conflict at some point or they can't establish the level of trust that is the foundation for Norming and Performing. So, the next time you see your team go from all smiles to arguing, just smile because you know that means they are starting to become a real team. Just don't let it get out of hand.

Another interesting thing about the model is that the state transitions going from left to right (from forming up to performing) are sequential. However, teams can go backwards quickly and without hitting each phase along the way. A team in the norming phase will revert to forming if the team membership changes or if their goal/project/purpose changes, and they have to go through storming again to get back to Norming (and they may not ever get there). This model will come up again here and there as we talk about Agile. I put it here in the book because I think it makes most sense when talking about project management because project management involves leveraging one or more teams towards a goal. It's important to understand how teams evolve and to build your plan accordingly or your plan will be fighting against forces it can't beat.

So, what is Project Management?

- A way of organizing people, tools, and resources to accomplish a goal
- It is non-static, requiring constant data gathering and adjustments
- It is part methodology, part psychology, part negotiation, part strategy

The goal can be software applications, buildings, starting a new business, buying a house, etc. Even a team of experienced people need to organize or be organized before they can achieve a goal. They will need tools to help them do their work. It will take time and money to accomplish. Project Management brings that organizational element into the effort. It includes a project plan with milestones that lead up to the goal much like we talked about setting 5-year goals, and a one year goal that gets you on track to it, and a 90 day goal to get you on track to the 1 year goal. The 90 day and 1 year goal are like milestones that must be achieved to ensure you are on track to the final goal. The project plan also defines what resources are needed and when. It defines who is involved and when and to do what.

Projects are never static because there are always unknowns (details, details, details) or unplanned interruptions (someone quits, inclement weather, power outages, pandemics, etc.). If the plan is too rigid, then these changes can cause major disruptions and even overall failure. Managing a project is not just about making sure everyone does what they are assigned to do; it's about adjusting the tasks definitions and estimates to circumstances; it's about motivating and inspiring the people involved to see the end goal as worthwhile and achievable, keeping them focused on win-win solutions.

Project Management is a formalized way to consistently solve problems in Software Engineering. It is actually agnostic of Software Engineering.

As you read the materials on project management, you can see how the process of solving problems (the same ones you probably use to develop code currently) can be

applied to running a project. The project's problem is "How do we deliver a solution for this set of SE requirements within the time and cost and scope constraints defined". If you agree with this assertion, then it shouldn't be a huge leap to view project management as a form of programming; breaking the project's problem down into smaller tasks that the team can execute and verifying this meets all your project goals/requirements.

The Project Plan

A problem is something that must be done but there is no known solution so we break it down into smaller more solvable pieces. When any of these pieces has a clear solution then it becomes a task to do. A project plan should not be composed of problems to be solved. It should be composed of tasks to be done.

A project is a collection of tasks

- Tasks often have dependencies (Functional and/or Resource)
- Tasks vary in complexity and duration and priority
- Effort estimates are more accurate for simple, short tasks
- The tasks are laid out in a schedule based on dependencies and staff availability

Oftentimes, the order tasks are done matter. This creates a dependency between tasks. Because a project doesn't have limitless resources, the same people have to work on different tasks. This creates a dependency between those tasks because you can't really work on more than one task at a time (you can time slice between them but that isn't the same thing).

Think in terms of building a house. You must build the foundation for a house before you build the roof. So, the foundation has to be poured and has to be set before you can add the walls. ALL the weight bearing walls have to be in place before you can build the roof. So, each wall is dependent on the foundation being poured and set. The roof is dependent on all the weight bearing walls being done.

Sometimes you can optimize for time. For example, you can build the rafters for the roof in advance (if you know the exact dimensions of the house and placement of the walls). This requires an architectural design to be fully detailed in advance. Thus, you can build the rafters any time before they are placed but they can only be placed once the walls are up. Of course, placing fully constructed A-frame rafters onto a structure is different from building them in place (you must use a crane or lots of people to lift it into place versus building it in place).

Breaking tasks down into smaller tasks is done in pretty much the same way we write code by breaking a system down into sub systems down into packages down into classes down into methods down into blocks of code down into lines of code.

Software Engineering

According to Kinser

Coming up with estimates is hard. The more people you involve in estimation, the more answers you get that are different.

It is easier to estimate tasks that are similar to ones we've already done and know how long it takes. But we rarely build the same thing twice.

Also, if I estimate the task that you have to do, you may not do it the way I would so my estimate for you will be off. It is extremely rare for large projects (or even most small ones) to get the estimates from the people that will actually do the work. Since estimates are often needed upfront, they are usually generated BEFORE the people are assigned to the project.

The way this is compensated for is by having the tasks be as well defined as possible and as small as possible. These types of tasks are easier to estimate accurately. Also, the person doing the estimates is usually a senior technical person that has estimated work for other people before so they know not to make assumptions (without documenting them).

For my projects, I tried to have all tasks sized so that they didn't take more than two weeks at the most and no less than 3 days at the least, with each task averaging one week. If a task was estimated to be more than two weeks, we broke it down. If it was less than 3 days, we added it to related tasks that were a week or less. It is important to actually estimate what it will take to do the task and not just slap a one week estimate on everything and hope it averages out.

Once you identify the tasks, most projects layout the tasks in a sort of calendar called a project plan. One format commonly used is the Gantt chart (see example 4 on https://www.edrawsoft.com/gantt-chart-examples.html). These types of formats can be used for any large activity (as you can tell from the examples in the link provided). There are a lot of tools available to manage project plans. I believe that the most popular is still Microsoft Project (although it is not often used for Scrum projects in my experience).

The tasks on the project schedule can be linked together to show time dependency between them.

FS = Task B can't start until Task A finishes (most common)
You can't butter the toast until the toast is done toasting.
SS = Task A and Task B start at the same time (but could finish at different times)
All plates at one table in a restaurant should be served at the same time.
FF = Task A and B must finish at the same time (but could start at different times)
Vegetables in a stew need to finish cooking at the same time.
SF = Task B can't finish until Task A has started (very uncommon)
We shouldn't arrive at the restaurant(B) before it opens (A)

Tasks can have offsets too. If Task B can't start until Task A finishes but you also want a week gap in between (because the assignee is on vacation or a business trip; which you don't want modeled in a plan the customer sees) then you do a FS+5 days.

Dependencies may be due to technical dependencies (e.g., you can't image a computer

until it is delivered) or a resource dependency (e.g., same person must do two unrelated tasks) or a time dependency (e.g., a revenue report can't run until after the end of the fiscal year).

Like your class schedule for a degree, a project can have a critical path (or long pole) where the length of the project is determined. This is a schedule risk and anything that can be done to disconnect the tasks to shorten the path should be explored. For your degree, there is a list of classes you have to take. Some classes have prerequisites; that class is dependent on you having completed the prerequisite class(es). There is a sub list of classes that are chained together by subsequent prerequisites. This sub list defines the minimum time you can obtain a degree. Most BS degrees are set up to take 4 years, but the sub list normally keeps you from doing it in less than 3 years. Of course, sometimes you can get waivers or substitute classes/experience or take some in the summer which can shorten it to under 3 years. A wise student (or one with a good advisor) will identify this sub list (the long pole in your degree program) at the outset of your college experience so that you can get started on the prerequisites as soon as possible. The same is true with software projects.

The person that manages the project schedule is often the project manager. This position is complex, and a good project manager can make quite a bit in salary. There are certifications available for this type of position (PMP and PMI are two of the most popular ones) and some job descriptions require or at least list these certifications as preferred.

One of the biggest challenges in a project plan is resource balancing. To balance a schedule, all the tasks need to be assigned to someone on the project, AND everyone on the project needs to be fully loaded (i.e., always has something to do) while not being overloaded. Having experienced staff with a variety of skills helps because they aren't limited to just one kind of task. Balancing all the tasks while keeping to a targeted end date can be a seemingly impossible chore.

Task Estimation

In the beginning of the project, the Project Manager (PM) and one or more senior technical people will collaborate to identify the tasks needed for a project. Since most teams aren't assembled ahead of time and not everyone starts at the same time, many project estimates are based on titles or roles in the organization with the expectation that anyone who is capable in the defined role would be able to do the work in the estimated time. This usually results in more conservative estimates. Estimates based on the assumption of a specific person doing the work is a risk to the project as that person could leave for any number of reasons.

If the estimates result in a project taking too long, the PM could ask for more senior people for some tasks to help reduce the effort needed to complete the tasks previously assigned to more junior people. This will increase the cost because more experienced people tend to cost more than junior people.

Software Engineering

According to Kinser

It’s nearly impossible to identify all tasks in advance. At the same time, some tasks identified in advance may be overcome by events and not be needed. Some tasks will take longer than estimated. Some will take less time than estimated. Ideally, these balance each other out (discovered tasks can be added because defined but obsolete tasks are removed; tasks that take longer than estimated are offset by tasks that take less time).

For projects with an established team (e.g. an iterative project adding new features to an existing functional product), PMs may come to team members to help identify and estimate tasks.

Estimates provided by senior technical people should be reviewed/validated by a “second set of eyes” just like code is better if inspected by someone other than the person who wrote it. In the process of explaining their estimate, the estimator may change their estimate or questions from the inspector could change it (the change could go either way but usually results in a longer timeline). The explanation of the task and reasons for the estimate are often delivered orally so it is easily lost or forgotten (and not conveyed to the person who ends up doing the work).
The person providing estimates should ideally be involved in the discussion with the client on the scope and nature of the work as well as in the requirements analysis. They should also be able to contribute to the initial risk management plan.

If I have defined all the tasks needed for a development project, I could ask each of you to independently estimate each task in the plan. The resulting timeline would vary significantly. Most (if not all) of you would make a variety of assumptions based on your current level of knowledge and the development experiences you’ve had so far. Almost none of you would ask clarifying questions to better understand the full scope and nature of the project. And almost none of you would look at the requirements document before starting your estimations. This is simply due to experience and why, as a project manager, I only involved my more senior technical people in estimation.

You’ll see this take place when we do the group project at the end of the semester and again if you take SE II. The question then becomes, will you recognize what is happening and do something to improve things?

Progress Reporting (Definition of Done)

Once the project is laid out and work begins, the everlasting struggle of trying to determine the current status of the project begins. Are we on track to finish on time, on budget, with all the features expected/promised, and is the solution of sufficient quality? This is called Progress Reporting.

In order to determine the current status, the Project Manager has to have a way to collect/track status from team members. One common BAD way is to ask each developer how far along they are on their current task. In my experience, most developers will say they are almost done. If you ask them for a percentage, it's usually

more than 50% and never quite 100%. This is another variation of the telephone game because the developers may not know or not think about parts of the task beyond the code such as documentation, tests, scripts that need updating, etc.

To avoid this miscommunication, it's important to have a Definition of Done for each type of task. The Design phase tasks will have a different definition than the Implementation phase tasks which are different from the Test phase tasks, etc. Even tasks within the same phase might differ somewhat. For example, hardware installation tasks in the implementation phase might be defined as done differently from software tasks. Having these Definitions of Done documented at the start of the project is super helpful because it impacts the estimates being generated when the project plan is defined. It also serves as a training tool for the staff as they join the project. The definitions need to be quantitative, not subjective. If you can identify all the verifiable steps/deliverables involved in completing a type of task, you can check these off when each is finished. The defined steps should include the superset of all deliverables expected. If a step is not needed for a specific task then the assignee can mark it as N/A and get credit for it but this requires them to make that decision purposefully and not out of forgetfulness. The task is only "done" when all of these steps are completed. This will result in a much more consistent reporting across all members of the project team. One can assign percentages to each step. This is a quantitative approach.

For Software tasks during Implementation:

- Code is completed (all code is written and runs) = 30%
- Code is reviewed, tested, corrected, and checked in = 30%
- Documentation is updated (user guides, etc.) = 30%
- Progress tools are updated = 10%

If you just ask a developer what their percentage complete is, some may weigh coding more than testing and documentation, while other developers weigh testing the most, while others don't even count documentation. Thus, you get different interpretations of what a given percentage complete means. If they all say they are 50% done, the Project Manager has no idea what that really means in terms of progress. This is a subjective approach and leads to issues (unless the Project Manager only updates the schedule when the percentage is 100%).

Using time as an indicator of progress has similar issues. If a task is two weeks long and someone has been working on it for one week, are they 50% done? Not necessarily. They may be completely lost and nowhere near finishing, or they may already be done. The original duration was just an estimate. We don't really know how long something will take until it is done. Also, if a more senior person had spent a week on it, one would assume they had made more progress than a junior person would have in a week. So, it can depend on the task, or the person, but time elapsed is not a good indicator of progress.

Reporting to the Customer

Many of the people who become Project Managers like to "please" the customer (which is a nice way of saying they don't like conflict). So, they tend to report positive progress

even when they know things are not on track (hoping that a death march, or a superhero, or divine intervention will solve the schedule problem before the customer finds out). A few PMs I've met are overly honest to the point that they will throw their team under the bus when things start looking iffy.

I've found that a project goes best when you "sandbag" the initial set of tasks with overly conservative estimates. This gives the team time to go through some of the natural formation stages (Forming/Storming/Norming/Performing) and gives them time to ask clarifying questions. I also don't report progress based on time (a 6 week project is not necessarily 50% complete after 3 weeks; could be more, could be less). I also don't base it on the number of tasks (a project with 100 tasks is not 50% complete when 50 tasks are done because some tasks are harder than others, etc).

Progress reporting should always be combined with risk management and change management. Risk and change management are not just tracking a list of risks and changes, it includes mitigation plans, workarounds, and options for the customer to consider.
One way to sink a project really fast is to let customers and team members talk directly without mediation from the Project Manager or a technical lead. This is one reason why Scrum introduced product owners (a liaison between customers and team members). I will say that customers and team members can interact directly under very strict conditions; the team must be small (so information can be relayed quickly and reliably), the team must be mostly senior with experience dealing with customers, and the customer has to accept that they potentially introduce risk to a project when they work directly with the team (they can make things better too but they need to know it could be the opposite).

Reporting to the Team

Some Project Managers never report progress TO THE team. A lot of PMs I've encountered only collect status from the team but I trained my Project Managers to also report status to the team. Sometimes the Project Manager and technical leaders on a team forget the team is not in all the meetings they are and don't have access to all the correspondence they do. The team will do a better job when they have better information about how the customer perceives the project's status, and are reminded of upcoming milestones and past successes. That is to say, they do a better job when they see a purpose and value to the project, when they feel appreciated and recognized for the work they've done, when they feel the work ahead is achievable.

Change Management

Because we are building something that doesn't exist, we often miss things or the situation/environment changes which impacts the project. Oftentimes, customers change their mind about key aspects of the project or about the priorities of things. Change management is a process to keep the inevitable changes to a project from derailing it completely.

Changes are introduced most often by the customer but they can also be introduced by the team as they execute on the project and gain a deeper understanding of the work to be done. These changes by the team can result in either less work or more work; it can go either way. Some changes are suggested alternatives to work requested (these changes are almost always a reduction in work or at least a reduction in risk to the project).

Timing is important when managing change. No one likes to be put on the spot to agree/disagree to a change they haven't had time to research or consider fully. Unlike requirements at the beginning of the project, changes often have the luxury of time so that they can be fully defined, discussed, detailed/documented, and decided on before you have to act on them. Change can be looked at in a larger scope as opposed to on an individual basis. As a Project Manager, I often combined/offset a customer requested change with one I knew the team wanted which results in a WIN-WIN scenario.

Each model of SE (Waterfall, Iterative, Scrum, etc.) handles change in different ways as we'll discuss when we dive into each model. In general, Waterfall tries to do upfront work to minimize the amount of and the impact of changes during a project. Iterative tries to do this for each iteration of a project, allowing changes needed to be addressed in subsequent iterations so as not to impact the current one. Scrum, similar to Iterative, will welcome changes but will try not to let them impact the current Sprint (iteration). The big difference between Iterative and Scrum with respect to change management is the duration of their iterations and the fact that most Iterative projects "manage" change while most Scrum projects "welcome" change.

As a developer, you have the ability to instigate a change without conferring with anyone. You can just write the code to do whatever you feel is best. Don't do this. You may think your idea is the greatest thing since sliced bread but if someone else is paying you to build the solution, build what THEY are paying for, not what you want. You are getting paid to do a job, not to do whatever you want to do.

Risk Management

Like change management, Risk management is a process that allows the customer and project team to document known risks to the project and review them regularly in an effort to mitigate the risk (this could be an elimination of the risk or a reduction of its impact on the project). Risk mitigation can add cost/time to a project only to have the risk never materialize but the cost/time impact of having the risk occur would most likely be greater if not catastrophic (which is why it was classified as a risk). So, you might think of Risk Mitigation as a form of insurance. You have to pay for car insurance "in case" you have an accident. If you never have an accident, you still have to pay for the insurance.

But, if the accident does happen, having the insurance reduces the financial impact on you.

Generally, every risk isn't reviewed weekly. A pragmatic balance based on priority and probability should drive which ones are covered each week. Many projects start off short staffed, so a weekly review of staffing plans is normal until fully staffed then staffing isn't covered even though it is a lingering risk.

Risks that are highly probable and highly impactful to a project must be actively managed. For example, if there are a lot of unknowns around the scope of the project, the most solid requirements might be scheduled to be worked on first with an architecture that allows for modifications and additions (pub/sub vs RPC). In other words, the team may use a more loosely coupled approach.

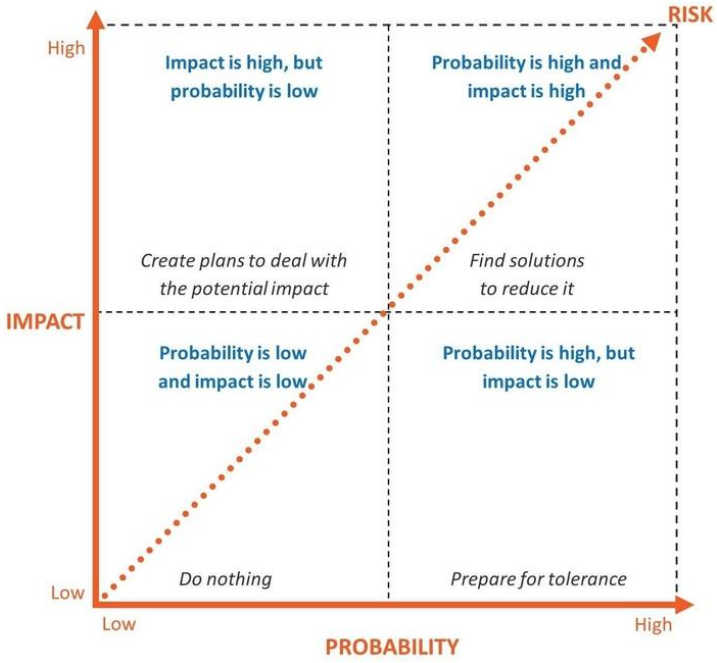
Risks that are highly probable but have a low impact to a project may only need a mitigation plan or workaround ready to put into place should the risk materialize. For example, a new version of the OS on which the solution is built may be scheduled to be released during the project timeline but it should be backwards compatible so the impact should be low. Some time might be added to the schedule to evaluate the solution on the new OS version.

Risks that are improbable and highly impactful would benefit from having a mitigation plan depending on how improbable. For example, a power outage would be highly impactful to a project but is low probability. Subsequently, the team might have USB (short term power backup) on critical circuits for any short term outages and a backup generator for long term outages.

Risks that are improbable and low impact might be documented but no action taken.

Software Engineering

According to Kinser



https://gohighbrow.com/identify-and-manage-key-risk-factors/

To be a Project Manager or not to be

Sometimes being proactive is also referred to as being fully engaged. If you are someone that waits until a due date to finish a project (or worse, to start a project), you should not consider being a project manager. If you, however, are someone that finishes an assignment as soon as possible so that you don't have to worry about it and realize this clears your plate for work that may arise unexpectedly, then you are a good candidate for project manager. Attributes of a good project manager:

- Proactive planning
- Proactive action
- Proactive communication
- Proactive tackling of the hard stuff like conflicts, risks, changes, etc.
- Proactively addresses topics/concerns/events/etc.
- Is equally focused on the customer's success and the team's success
- Solicits feedback and open to ideas from any involved party

Software Engineering

According to Kinser

Is able to and willing to make decisions in a timely manner
Shares the credit, but takes the blame
Takes team member's career goals into consideration in planning work
Looks for solutions, not scapegoats

A project manager is a leader, not a boss. They don't have to know how to do everything each member of the team has to do. They just need to know who knows how to do it. They need to know their team and how to lead them. They should be adept at techniques for problem solving so they can solve issues or facilitate others in solving the problems. They need to be able to handle a changing environment without stressing out.

ETSU offers a graduate course in Project Management. Austin Mahala, an SE I student in the Fall of 2023, discovered that the Project Management Institute has a student membership program: pmi.org/membership/student. Same access/benefits but at a reduced annual fee ($32/year in 2023).

Software Engineering

According to Kinser

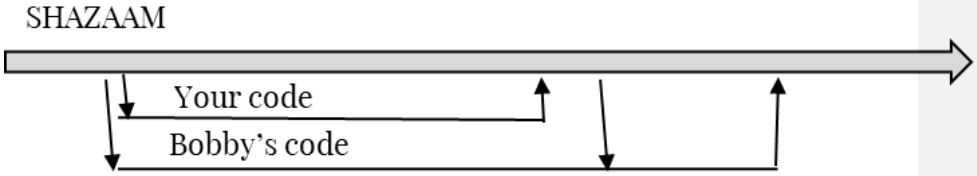
Chapter 6 - Source Control

Source control is like a database. When you write a stored procedure that modifies data, you usually start a transaction, make your modification, and then end the transaction. The “transaction” locks the data set so others can’t modify it at the same time. Also, if your modifications cause an error, you can rollback the transaction, undoing all your modifications.

Source control does this for source code and even more. In addition to helping the team be more efficient during development, the Version Control System (VCS) usually supports the management of the releases and versioning (to be discussed later) so the team can keep track of which changes go with which releases. There are also checkpoints and signoffs that can be configured into the system. For example, the VCS can ensure that all required tests pass every time a change is made to the product, automatically. Some can even support code inspections (e.g., static analysis) and workflows required before changes are made permanent.

Source Control is absolutely critical when teams are working on the same code set. Keeping track of changes in a newly created solution is fairly easy. Keeping track of changes to a product that has existed for years and is deployed in a variety of environments and written by a nearly countless set of people, can be problematic.

A project will likely store their source code in a Repository. A Repository is like a disk on your computer on which you have stored all your files, and it can also have a directory structure to organize the files. In VCS, those directories are referred to as Branches. By default, a repository has a “master” or “main” branch. This is like the root directory. Each branch can have one or more branches based off it. For example, a project will likely have a Development Branch where current implementations are being done. Each new feature being added is done in a separate branch off the Development Branch. These are called Feature Branches. When the work in the Feature Branch is completed (i.e., meets the Definition of Done; see Project Management chapter), it is Merged back into the Development Branch. Most VCS will handle the merge for you but if there are conflicts in changes it can’t figure out, it will alert you to a “merge conflict” which must be resolved manually.



In the diagram above, the project Shazaam has a Development Branch (the thick arrow) and Bobby creates a Feature Branch for his code. You create a Feature Branch for your code right after he does. Because Bobby is a bit slow (or maybe you are just fast), you finish before he does. So, you merge your code back into the Development Branch. Sometimes the merge process is called a commit. You performed a commit to the Development Branch.

Since Bobby isn't done yet, he needs to do a Pull to bring his feature branch current with any changes in the Development Branch (he's pulling in any updates made to the same branch that he started his feature branch from; in this case the Development Branch). Because your code was merged before he did this, Bobby "pulls" your changes into his Feature Branch. This allows him to resolve any conflicts he has with your changes and verify his changes work with your changes before he merges his code back into the Development Branch with a commit of his own. If everything works as it should, this process of pulling in changes and verifying things work should only take a few minutes. If something goes wrong the problems are isolated in your Feature Branch and are not in the Development Branch. This means the problems only impact you and not the entire team.

The VCS keeps track of every change Bobby made to any files in the Shazaam Repository. There is a timestamp with each change as well. In some courses at school, you may have had to add a header comment which listed the author of the file, the date created, and the date of last modification. All this is done automatically by VCS, so those comment blocks are redundant.

Once all the development is completed, the team may elect to create a Test Branch off the Development Branch. The Test Branch is like a snapshot of the Development Branch. This way they can keep testing the same set of code changes. If they report a defect, the team can fix the defect in the Development Branch without impacting the Test Branch. When enough fixes have been made, the team may elect to "pull" all those fixes into the Test Branch just like Bobby pulled in your changes into his Feature Branch.

After all the testing is done, the team could branch off the Test Branch to create a Staging Branch where all the load testing and stress testing will occur. This should happen if your team has a Staging environment that mirrors production. After that final set of tests pass and the solution is ready for production, they will branch off the Staging Branch to create a Production Branch. This way they can be confident that the solution going into production has passed all the tests conducted in the Test Branch and the Staging Branch.

After the first release, you must keep production separate from development. As more releases are made, they must be tracked and supported. This is where versioning helps

a lot. Over the years, people have learned that it is important to label each product release. If the product is large and consists of a collection of smaller products or subsystems or APIs, these will likely be versioned too (but independently of the larger product and each other).

Versioning is a way of naming each release logically so that you can readily see both the chronology of the releases but also their relationship to each other.

You probably have seen “versions” of versioning and not really paid much attention. Often the version is included in the Help menu or the About menu of an application. If you need support on a product, the help center will probably ask for the version you are using because that tells them which known defects have been fixed and which have not. It also tells them what features exist in the product you are using.

While there is no universal law about how to version a solution, the most common is called Semantic Versioning. It consists of three values; [Major].[Minor].[Patch]. The Major value indicates significant changes in user experience either by new features/capabilities or a new look-and-feel or a combination (e.g. Windows 10 vs Windows 11). The Minor value indicates an update or upgrade to the Major release that precedes in the version. These new changes are significant but don’t really affect the user experience. No new training will be required and it is always backwards compatible with other Minor releases associated with the same Major release (e.g. Version 5.4.<patch> is backwards compatible with 5.3.<patch>, regardless of the patch value). This brings us to the Patch value. The Patch value is increased if a fix is required to a specific Major.Minor release before the next Minor release can be made. This may be due to a security flaw, a requirement related defect, or a critical usability issue. Any patch release is backwards compatible with all other patch releases for the same Major.Minor release (e.g. 5.4.3 is backwards compatible with 5.4.2 and 5.4.1 and 5.4.0). Taking the latest patch normally includes all the previous patches in the same Major.Minor release.

Versioning Example:

The Orange company releases a new product, Portal_1_0_0
They begin work on a new feature in Portal_1_1_0
Before they finish, a critical security flaw is discovered in Portal_1_0_0 so they release a patch, Portal_1_0_1
This fix is pulled into Portal_1_1_0 which hasn’t been released so no changes in version number is required.
Portal_1_0_0 still contains the flaw because you don’t change the code in a released version, you release a new version; Portal_1_0_1.

Bug fixes made in earlier versions (the patch example above) must be propagated to all subsequent versions (it was added to Portal_1_0_1 and Portal_1_1_0) because the bug probably exists there too.

Versioning is critical to software development. Fortunately, we have tools to help us track them (imagine having to do this manually by keeping software in some type of file structure).

A Baseline is usually a phrase that indicates a planned release version (sometimes called a frozen branch or protected branch). Sometimes Baseline means that we are going to have a code freeze on that branch (no more modifications will be allowed, you must branch off it in order to continue development). For example, if your team has been working on a demo and you finally have it together and ready to go but the demo isn't for 3 more days and the team needs to make changes unrelated to the demo, you would baseline/freeze/protect the branch you did the demo development on (freezing it so as not to screw up your demo) and then branch off it to continue doing development .

Software Engineering

According to Kinser

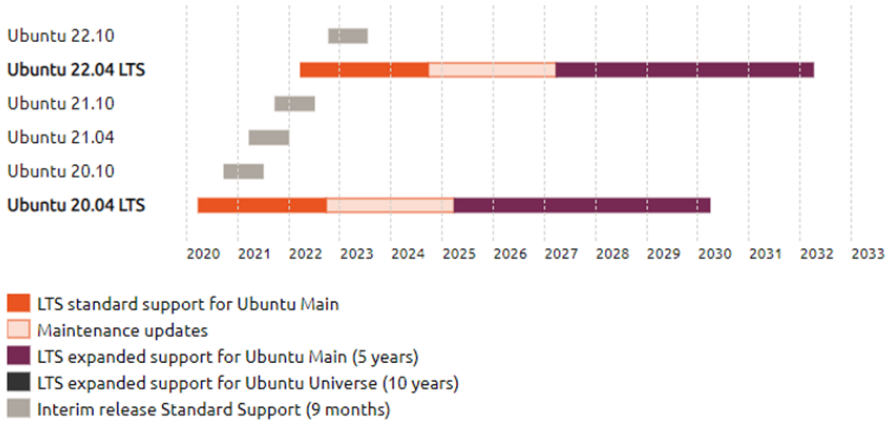
Real World Example

https://en.wikipedia.org/wiki/List_of_Linux_distributions

Linux has lots of “distributions” which is another way of saying there are many branches from the original version of Linux. Each of these branches evolve with use and therefore need to release new versions of themselves.

Software Engineering

According to Kinser

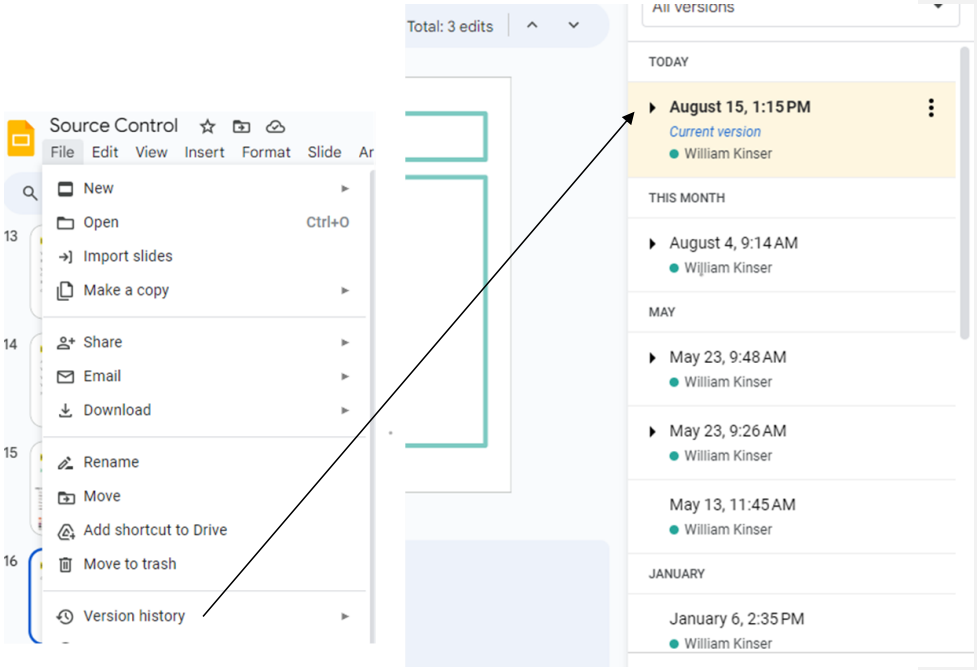


https://ubuntu.com/about

Ubuntu branched off Linux. For more info, see https://ubuntu.com/about . They appear to use Iterative development because they release updates every 6 months with major releases every 2 years; and patches as necessary.

Software Engineering

According to Kinser



If you use Google Docs, it automatically does versioning for you. Open a Google Doc, in the File menu, find “version history”. It will display each version. If you click on an old version, it will show you the edits made in that version. You can elect to make that version the current one, in which case it will discard all changes made after that version and you start anew with that selected version.

Nice Introductory Tutorial on GitHub

https://www.freecodecamp.org/news/how-to-use-git-and-github-in-a-team-like-a-pro/

This online tutorial above is a good way to get introduced to source control.

Software Engineering

According to Kinser

Chapter 7 - SDLC

Whether you realize it or admit it, all software goes through the same lifecycle. In fact, almost all problems that have much complexity to them will go through this lifecycle if the goal is to have a robust solution to the problem. First you have to define the problem well enough to identify a solution. Then you have to define how that solution can be built. Once built, it is important that the working solution meets the original intent before being made available to users. Once it is in use, there is upkeep and fixes that will most likely be needed over time. Some solutions are more dynamic and receive periodic upgrades and updates to keep it current with changing needs.

The Lifecycle describes the normal phases that software/hardware solutions go through in order to become a reality. While some texts only include Requirements, Design, Implementation, Test, and Release; I include Support/Maintenance and Upgrades/Updates because a solution isn't really a solution if it doesn't exist beyond the Release phase. Even satellite systems need to be maintained. Some of the Mars rovers would have stopped early on if they didn't have the ability to accept patches to their original implementation.

I also distinguish between Support/Maintenance and Upgrades/Updates for what probably feels like arbitrary or academic reasons. Support/Maintenance is the minimal amount of changes needed to keep a production system running (released as Patches). It primarily includes bug fixes but may include changes needed to support changes in third party products that the solution depends on (e.g. changes in case it depends on Windows10 and now needs to depend on Windows11).

Upgrades/Updates covers changes of any size that results in some new behavior/feature and modification to existing behavior/features (released as minor version number increments). These changes are additions to the original design and not the correction to any defect in the solution.

Software Development Lifecycle consists of the following phases:

Requirements	- Defines "What" is needed in a solution
Design	- Defines "How" the solution should be built
Implementation	- Makes the solution a reality
Test	- Verifies the solution does "What" it was supposed to
Release	- Makes the solution available to users
Support/Maintenance	- Keeps the solution working as long as it is used
Upgrades/Updates	- Improves the solution after initial release

Each SDLC Phase has some type of deliverable as a post-condition:

Software Engineering

According to Kinser

Requirements	- Requirements Set
Design	- Architecture/High Level/Detailed Level/Traceability Matrix
Implementation`	- Code/Scripts/Hardware/Traceability Matrix
Test	- Test Plan/Test Cases/Traceability Matrix
Release	- Release Procedures/Scripts/Release Notes
Support/Maintenance	- Release notes and/or a ReadMe file
Upgrades/Updates	- updates to all of the above

As described in Problem Solving, a problem is something without a clear solution. Software Engineering solutions are created by analyzing the problem and breaking it down into smaller problems until they become tasks to be executed. The SDLC is the process followed to do this.

In addition to the lifecycle phases, there are lifecycle models/methodologies. The SDLC models represent different ways to apply the SDLC phases based on the type of work that needs to be done. The most common SDLC methods include Waterfall, Iterative/Incremental, and Scrum. We'll cover these three in greater detail in later chapters. We will also spend a lot of time on Agile (which is a project management philosophy) as it can be overlaid on these three models. Some other models we won't cover include XP (Extreme Programming), Spiral, Build/Fix, DevOps, and Kanban. We'll cover DevOps and Kanban in 4350 but as extensions of Scrum, not as stand alone methods.

A quick summary of the waterfall model is that it is the birthplace of the SDLC. In waterfall, all the phases are executed in sequence completely before moving to the next phase. Waterfall is a very formal process with specific artifacts in each phase. Some consider it to be very rigid. The approach is used most often in life critical, mission critical, or business critical solutions where all the capabilities of the solution are needed at the same time. It is sometimes referred to as the Big Bang approach.

Iterative development is so similar to Incremental, that I just lump them together. In the iterative model, the SDLC phases are executed in a series of mini-waterfall efforts called iterations. The project starts with a nearly complete set of requirements that are then distributed across a series of releases roughly equal in duration (e.g. every 6 months). Each iteration executes the phases like waterfall but on a subset of the overall set of requirements. This provides more ability to adjust or change requirements than can be done in Waterfall while still leveraging the formal processes. The difference between iterative and incremental can be found in what is released each time. In iterative, the releases build on each other from a functionality perspective but may not be usable by the target users until the last iteration. Incremental does something similar but each release is meant to be a usable release.

Scrum is currently the most popular method, especially for internet-based solutions. Scrum inherits a lot from the incremental method but doesn't have as rigid a process to follow. The duration of the releases (called Sprints in Scrum), happen rapidly. Each sprint can be a week or two up to a couple of months in length; all sprints on the same

project will have the same duration. A big claim to fame of Scrum is its acceptance of and ability to respond to changing requirements.

Agile is a project management philosophy that can be applied to any method but is most commonly associated with Scrum. Agile is a mindset, a point-of-view, an approach. It does not have defined roles or specific processes that must be followed. Yet, it is extremely powerful when executed well.

The Requirements Phase

Most everyone has an opinion. However, the opinions of the stakeholders matter a lot. The opinions of the client's decision makers matter most. Like with interviewing for a job, there are "gatekeepers" and there are the people that actually decide things. Stakeholders are often "gatekeepers" with respect to requirements. They typically have an area of specialization that they focus on (e.g. marketing, sales, support, etc.) and rarely have, as individuals, a holistic view of the solution. Thus, having the appropriate breadth of stakeholders helps ensure the requirements are robust. Stakeholders rarely have the ability to decide cost and schedule. This means they also can't really define scope or priorities for the solution, but they have strong input on the scope and priorities for their area of speciality.

Having too many stakeholders can lead to stagnation. If at all possible, strive to have one stakeholder from each department/topic/category so the team doesn't have to mediate between stakeholders. Another risk is that not all stakeholders buy into the solution and may be resistant to help move it forward. Office politics can come into play as well.

The decision makers are the people that have ultimate authority to decide costs, set holistic priorities, agree to dates, and approve contracts/purchases. Ideally, there is only one decision maker with a backup/surrogate (someone that has the authority to act in the absence of the decision maker). The more decision makers there are, the harder it can be to get critical decisions made in a timely manner.

With both stakeholders and decision makers, the best way to get the most out of them is to be prepared in advance of meeting with them.

Requirements Gathering

Any SE problem is broken down into Requirements that define what a solution will do to address the problem. This is commonly referred to as Requirements Gathering or Requirements Elicitation. It is rare that an SE problem is so simple and straightforward that one can immediately start coding it. By asking a variety of clarifying questions, the analysts can identify major requirements which can be further broken down until each requirement is understood well enough to create a design for it.

Software Engineering

According to Kinser

The most helpful questions are open-ended questions. Questions that get the stakeholder to talk about the solution.

For example:

"What problem or opportunity does this solution address" is a question that gets to the heart of the solution. Some customers forget that the engineers weren't part of the myriad of meetings that led to getting the project approved.

"How does the business solve this problem at this time" is a great open-ended question that can lead to an hour long explanation filled with valuable information and will likely lead to follow up questions like "And what do you like about how that works today" then "What do you not like about how that works today".

"Is there an existing solution comparable to this one? If yes, what are the key differentiators" is a question that can address hard to define aspects of a solution like "usability".

Asking these clarifying questions will help the team understand the "words behind the words". Although a requirement is only a single sentence, it typically embodies paragraphs of information. These types of questions can be used to define the requirements needed for the solution.

Requirements are a textual description of what the product/solution needs to be able to do for it to be deemed successful. They should only define WHAT it needs to do, not HOW. Requirements are difficult to write and just as difficult to interpret. There is a great temptation to define HOW the solution should be built as a requirement. This usually leads to false starts and issues later on because the original problem wasn't fully broken down and the proposed HOW does not address all that is needed.

Requirements are either Mandatory or Optional. Mandatory is often denoted in the statement by using MUST or SHALL or SHOULD. Optional is often denoted in the statement by using CAN or MAY or COULD. Regardless of how it is denoted, it must be absolutely clear which are mandatory and which are optional as this affects the scope, cost, and schedule. Think of it like buying a car. It is mandatory that the car conform to transportation regulations (e.g. seatbelts, brakes, etc.). It is optional if the hubcaps spin after you come to a stop or if the seats are heated. The base price of a car reflects all the mandatory requirements. As you include the optional requirements, the price goes up and the delivery may take longer.

Asking "If we can do everything else in the requirements set but not this requirement, would the solution meet your needs?" If the answer is Yes, then the requirement is optional.

Requirements are either Functional or non-Functional. Functional requirements describe WHAT the solution must DO. Non-functional requirements describe WHAT constraints exist on the solution. Constraints are most commonly performance related or address the "ilities"; Portability, Security, Maintainability, Reliability, Scalability, Reusability, Flexibility.

Example: The solution MUST support up to 100 concurrent users.
Example: The solution MUST respond to user actions within 1 second.

Some non-functional requirements are process or project specific. For example, the solution may need to be written in C# because the company has that as a standard for all their products.

(http://codesqueeze.com/the-7-software-ilities-you-need-to-know/)

If a requirement set has no non-functional requirements (or very few), it is almost always a sign that it is incomplete and that there are probably more holes in it than immediately apparent. If the customer who defines the solution doesn't address performance or the "ilities" then they haven't thought it through enough or haven't included enough stakeholders in the creation of the requirements.

Some people get really caught up in the distinction between functional and non-functional. In my experience, what matters is whether the requirement is SMART (that is to say, a Good requirement) and whether it is mandatory or optional (as this impacts cost and schedule).

Sticky Note Brainstorming

One brainstorming technique that I have found effective in requirement gathering is the Sticky Note exercise. This works well with most any size group but anything over 20 people probably needs some adjusting. In this exercise, all the stakeholders are given a stack of sticky notes and a pen. There will be multiple rounds of brainstorming and organizing, with ideally no time limit. In each round, everyone will write down up to but no more than 3 requirements they believe the solution must meet to be successful. Each requirement is on a separate sticky note.

Once everyone is done the facilitator will ask each person to put their notes on the wall or whiteboard. As they do this, they should read the note to the group and briefly describe what it means in one or two sentences. We don't worry about how the requirement is written (as a User Story, using formal English, or just a snippet). Ideally, there is someone (not a stakeholder) who is capturing what is said in "minutes" that can be referenced later. This is helpful but not critical. Every person should read their notes even if they feel it is a duplicate of one already read because they might have a slightly different but significant perspective. After everyone has put their notes up, the whole group will go up and organize them. This usually seems like a daunting task but every group I've ever worked with ends up rising to the occasion.

To organize the sticky notes, they should try to eliminate duplicates and group related requirements together. Duplicate notes are stacked on top of each other. Related notes can be loosely grouped in no particular order or structure. The group can decide. They can also create a heading for the related items to help in organizing them. It is not uncommon for a group of related items to grow so large in subsequent rounds that the notes need to be broken up into "subtopics".

The facilitator continues to ask the group for another round until there is a clear indication of diminishing returns. This can be identified by fewer and fewer of the stakeholders coming up with three notes in each round. If half the group can't even come up with one, the exercise is over.

Why does this work? Why only 3 per person per round? Every group of stakeholders always has at least one person who is more extraverted than the rest and will dominate the brainstorming of requirements if left unchecked. To avoid "groupthink" this exercise gives everyone equal representation. Each person gets a turn to put forth their ideas. This also encourages the more introverted people to speak up as well. Limiting each round to 3 notes per person is not entirely arbitrary. It needs to be limited or the dominant person will, well, dominate with a whole litany of their ideas and never yield the floor. Keeping it to 3 makes it seem a realistic request to those that don't have a lot of ideas initially. Also, it minimizes the amount of time wasted on duplication. There will almost always be some amount of duplication but if everyone wrote down 20, there would be a lot more duplication because there would be more notes but also because, when limited to 3, people won't duplicate the notes from the first round in subsequent rounds. For the first few rounds, stakeholders usually focus on the items specific to their area of expertise because that is their comfort zone. Thus, the only duplication in the first few rounds typically arises from areas of overlap. Limiting each round to 3 also speeds up the organizing activity.

Writing Requirements

There are a number of different formats or styles of requirements with pros and cons for each. The ones I've used most are Structured English and User Story. The Structured English format is classical and has been in use for decades. In general, requirements are captured as *The System Shall <do some feature/function/behavior>*.

A "good" requirement is a statement, single sentence, describing WHAT the solution will do.

Example: The game MUST allow all players to move at the same time (Good).

Each statement should encompass a single idea (no conjunctions).

Example: The editor must support an Undo function as well as a Repeat function. These are two separate requirements merged into one (BAD).

A User Story format follows the pattern of As a <actor in the system>, I want < do some action> so that <some benefit or business value>. The same requirements could be written as a User Story.

Example: As Players, we MUST be able to move at the same time so that we can interact. (Good)

Example: As a Writer, I SHOULD be able to undo any action as well as repeat any action so that I can save time and effort. (BAD because it's two requirements in one statement).

Software Engineering

According to Kinser

Humans continue to struggle with written language. Consider “Let’s eat, Grandma” vs “Let’s eat Grandma”. The first is an invitation to a meal. The second is an invitation to be the meal. The difference? A single comma. It could make the difference between life or death (if you are Grandma).

All kidding aside, we struggle with our ability to convey intent, details, importance, nuance, etc. when the written form is the only way to pass on information. This is why so many “new” development models emphasize regular and direct communication (in person) between the Dev Teams and the customers/stakeholders.

Even then, if you don’t clarify your understanding to the other party, miscommunications and misunderstandings can kill an otherwise successful project. A good way to ensure you understand what the other party is telling you, is to repeat back to them what you heard but in your own words and by asking clarifying questions.

Requirements Analysis

Requirements analysis is like a design review or a code inspection. It is a form of verification that is performed to improve the quality of the requirement set. If the requirements are too vague, missing key elements, in conflict with each other, or not realistic, it will cause problems later on in the project. The best time to identify and fix any issues with the requirements is at the outset. If it is done later, it takes more time and costs more money because work will likely have to be redone or wasn’t needed at all.

Requirements should be SMART: Specific, Measurable, Attainable, Relevant, and Timely.
If this sounds familiar, this is the same approach I recommended for setting goals in the Jobs section. It works for goals and it works for Requirements.

Example questions you can ask related to SMART:

- What problem or opportunity does this solution address (SPECIFIC)
- How will you know if this requirement is met (MEASURABLE)
- Is there an existing solution comparable to this one? If yes, what are the key differentiators (ATTAINABLE)
- Who are the targeted users for this solution (RELEVANT)
- Will the system work if this requirement is not implemented in the first release (TIMELY)

Specific means that Requirements should not be vague and they represent an important aspect of the solution. Measurable indicates that it can be tested or verified. That is to say, there is a way to prove the solution includes or meets this requirement. Attainable means the feature is achievable within the scope of the project (e.g. installing cold fusion generators in every home is not achievable because cold fusion is still in the experimental stage). Relevant means that it has some value (either business value or value to the end user). Timely speaks to when the requirement is needed (e.g., requiring all states to install charging stations in every rest area on the highway is not timely

because there aren't enough EVs in use or with enough range to justify the effort).

If a requirement is not specific enough, it is often because it was written to encompass several capabilities or represent something complex. To address this, use problem solving techniques to break the requirement down into "derived requirements". The original can be kept to act as a group header for the derived requirements. Each of the derived requirements are then analyzed to ensure they are SMART. If any are not, break those down further, and so on.

Example: The solution MUST support multiple players simultaneously.
This is too vague. Perhaps it can be broken down to address multiple players moving at the same time, chatting with each other in real time, exchanging items, fighting together against a common enemy, etc.

It is absolutely critical that the requirement be stated in a way that can be measured, or verified. Otherwise, it will be nearly impossible to prove the solution does what it is supposed to once completed. A common requirement that is difficult to measure is "The system must be user friendly". What does it mean to be user friendly? How can you prove that it is user friendly? You would have to do case studies or user group sessions, set metrics, collect feedback, etc. Even then it would not be conclusive. You could say "The system must support 1,000 concurrent users". To measure this, you simulate 1,000 user sessions at the same time in a staging environment.

To be attainable simply means it is possible to do. This gets a little tricky as constraints are added. Given enough time and money, most anything is possible. Cold fusion is possible. It might take centuries and the entire global GDP, but it is possible. But it isn't possible on a 6 month long project with 15 people. Sometimes it is not possible to determine if the requirement is attainable until the design is completed. Beyond time and cost constraints, if two or more requirements are in conflict with each other, then one or all are not attainable. This requires a comprehensive and holistic analysis of the requirements set which can be time consuming. In many cases, looking for conflicting requirements in a subset of related requirements (e.g. all requirements associated with player actions, or all requirements associated with power consumption on a device) is sufficient due to the low probability of conflicts across categories.

For a requirement to be relevant is another way to say it is "in scope" to the effort and that it adds either business value or user value to the solution. For example, tracking how long each player has been online has business value, displaying that information to each user while they are playing might not. Capturing WHY the requirement is needed helps establish its relevance. This is built into the User Story format when properly written.

Whether a requirement is timely speaks to when it is needed. It could be a completely valid requirement with a business value but not something that is immediately needed. The timeliness of requirements can be leveraged when prioritizing or scheduling work. It is tempting to build a feature rich solution all at once but oftentimes, only the common features are needed in the initial releases. Advanced capabilities can be added on later as users become familiar with the product.

Software Engineering

According to Kinser

How do you know if a requirement is SMART enough? When the team feels confident that they can design a solution for it. They don't have to do the design to make this determination. They just have to understand the requirement well enough to be confident they can. Basically, they estimate that it can be designed.

Requirements can be static or dynamic in nature. As we discuss different lifecycle models, this is part of the differentiator for which model to apply. Structured English format is commonly used in models where the requirements are more robust and complete prior to development. They are stored in a Software Requirements Specification (SRS) document or set of documents. The User Story format is more common in models where it may not be possible to know all the requirements upfront, and changing needs are very common. In these models, the requirements are stored in a living document or an application that supports frequent updates.

The Design Phase

The Design Phase begins when the team starts working on HOW the solution will be built. The requirements set is a precondition of the Design Phase, because you need to know WHAT needs to be built before you decide HOW to build it. HOW the solution is implemented is documented textually and graphically in the Design phase. The design is often done breadth first and iterated on until a sufficient level of detail (or depth) is defined so that implementation can begin.

For simple problems like students normally encounter in a course, the design phase may be straightforward and happen quickly. In fact, you may think you can code a solution without doing a design. The reality is that it is impossible to build something without thinking about HOW you are going to build it. You probably figured out the design in your head without using a formal process (which probably led to having to rework the code because your design was not robust).

The Design Phase also follows the problem solving process. You take the problem (the requirements document) and you break it down into something more solvable/actionable, a design. Thus, the design is the solution to the requirements problem. If the team that is working on the Design is different than the team that did the requirements analysis, you will want to do some type of requirements review to bring the design team up to speed on the larger picture of the solution before they start breaking it down.

The formal process breaks the design down into greater and greater detail. The point of doing a design is not to have documented design. The point of the design phase is to break the problem down into smaller, solvable pieces until you get to the point where you can implement it.

- Architecture Design (Scaffolding for the entire solution)
- High Level Design (System/Application design and APIs)
- Mid Level Design (Subsystem and services design)
- Detailed Design (Class/Method design)

Software Engineering

According to Kinser

There are generally accepted techniques that exist for the design phase in Software Engineering. The unified modeling language (UML) defines a number of diagrams and graphics that can be leveraged throughout the design phase.
https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/
OR https://www.lucidchart.com/blog/types-of-UML-diagrams

Another technique commonly used is referred to as design patterns. These patterns define a set of commonly used designs based on certain attributes or goals. You've probably heard of Client/Server architectures. There is an architectural design pattern called Client/Server that describes this in a generic way. Designers can then apply the pattern, adding customizations and details as appropriate for their needs.

https://learningdaily.dev/the-7-most-important-software-design-patterns-d60e546afb0e
https://www.netsolutions.com/insights/software-design-pattern/
https://www.turing.com/blog/software-architecture-patterns-types/

The advantage of using UML and Design Patterns is that it establishes a common language that can be used during the design phase and implementation phase to communicate key concepts. Think about the old saying, "A picture is worth a thousand words." Including UML diagrams in your design can convey a lot of meaning and save a tremendous amount of writing. It can also reduce confusion and assumptions because you are using a language that spans models. UML is not confined to Waterfall models or Scrum models. They can be used for both software and hardware. Similarly, design patterns go beyond individual models and encompass both software and hardware.

As a new graduate, you may not be expected to create a design of any real complexity in your first job. You will likely be expected to be able to follow a design someone else created using UML and/or design patterns during the implementation phase of a project (or during a sprint).

My personal experience is that no team I was on at any level (except when doing military contracts) used all the UML diagrams, but they used some of them. No project used all the design patterns, but they did use some. When we used both of these techniques, we didn't leverage all attributes and symbols of UML nor all features of the patterns. We applied them in a pragmatic way.

Both techniques can appear in each level of the design as will be discussed below. Some diagrams and patterns can be used in multiple levels of design. For example, a Broker can be placed at the Architectural level to broker between applications or it can be placed inside a single application to broker between application components.

Architectural Design

Architectural Design is most often performed by a very senior technical person (sometimes called the Architect for this reason) or a very small group of senior technical persons. It may even occur concurrently with requirements analysis (which is my

preference).

Architectural Design outlines interactions between Systems/Applications and typically starts with a well-known architecture which is then tailored as necessary to meet requirements.

Examples: https://www.turing.com/blog/software-architecture-patterns-types/

It often begins to address the “ilities”, especially scalability and security. The architecture includes a data model; not to be confused with a database schema, the data model identifies the key data that are passed across the solution. It is a very high-level view. A database schema would be a detailed design version of the data model. Identifying how these key data items are passed around in a solution is represented as the Data Flow.

Systems refers to an application, or family of applications/services that together stand independently of the rest of the solution. For smaller solutions, this may just be a package or library.

Part of the activity of architectural design is to map which requirements are applicable to each system of the overall solution (traceability matrix) and to ensure all requirements are mapped. A traceability matrix is a table that maps the key design components to one or more of the requirements for the solution. Note: sometimes a requirement maps to multiple systems (meaning they all have to work in order for that one requirement to be met). This is harder to track and test. Refining the requirement into Derived requirements where each derived maps to a single application or system can help address this traceability challenge. Each derived requirement can then be verified independently (as each subsystem or application in the solution is completed). The sum of the derived requirements should equal the original requirement. That is, once all the derived requirements are verified then the original requirement is considered verified.

High level Design

For non-simple problems, the initial pass at breaking down the requirements usually results in the definition of the High-Level Design which often includes a Data Model. It starts with the architectural design and breaks it down into smaller more solvable pieces: modules, packages, service and subsystems.

I purposely say Data Model because it is a superset of the database. It includes data structures that are passed between systems or applications but may or may not be persisted to a database or other repository. Data Models, like the software design itself, can be defined in iterations or layers with increasing detail.

It is important to define the execution flow (sometimes called control flow) and data flow between systems in the solution at this stage so that each can be further refined concurrently (possibly by separate teams). A first draft of an API (Application Programming Interface) for each system may be possible at this stage but rarely will it be complete until the next level of design is done. The API for each system defines the contract of control and data flow. Like methods in a class, the API will define external

actions each component will support and the attributes needed as inputs and outputs for those actions.

For large projects that require multiple teams to build the solution, the API may align with the team structure. For example, if you are building a client-server application with a large data store (think retail site), there might be three teams; one for the client facing portion (the app or browser solution), the server-side business logic/controls, and the database team to organize/optimize/index/populate the data. The interactions between these three components of the architecture will be defined upfront as an API. Like a business contract, the API allows each team to work independently as long as they don't alter the specifics of the API.

At this stage of your academic studies, the need for the different "views" of UML is probably more obvious than when you first encountered them. It is more likely that you will "get it" if you research these now. Personally, I never found it necessary to use all the various views/diagrams. Some of the more useful ones in my experience are Interaction, Sequence, Use Case, and Activity. If a solution has a complex state engine, then the state diagram can be useful. I have also found that most developers can read these diagrams even if they aren't able to describe in detail how to construct them. It's kind of like the fact that most people can read a story even though they aren't necessarily able to define sentence structures/components or how imagery works, etc. You don't need to know which part of a sentence is a participle in order to read the sentence.

The whole point of the High-Level Design is to apply the principles of problem solving to the project; to break the design down into smaller, more solvable pieces. When you have defined all the systems, packages, services, modules, and APIs then you are probably done with High-Level Design.

Mid-Level Design

The Mid-Level Design activity is not as widely discussed as High-Level Design and Detailed Level Design. In most projects, the outputs of this effort are commonly split between high and detailed. The mid-level design outputs detail out the data, control, communication, and other interactions WITHIN a package, service, system, or subsystem.

If you think about a fighter aircraft, there are several major systems that would be defined during high level design like weapons systems, avionic controls, safety systems, propulsion, etc. All of these interact with each other (avionics are based on weight which is affected by how many or what kind of weapons are on board and how much fuel, while the weapons systems need to know if the aircraft is upside down or in a deep dive, or on the ground). Contrary to the movies, most fighter aircraft cannot fire missiles while on the ground. Defining all this is very involved and when done you can't start detailed design. There is still a lot of design work to do before you can detail out the classes and ERD. For example, what is the protocol all weapons have to use to communicate with the weapons systems on the aircraft, so it knows what kind of weapon it is, how to arm it and

how to target it and how to release it. These would be mid-level design items.

During the mid-level design, you may identify classes or modules but stop short of detailing them out. The UML diagrams commonly used during this activity include: Interaction, Sequence, State, Activity & Object diagrams.

Detail Design

This is the level of design you have done in your studies thus far. It includes Class Diagrams where you define member data, methods, parameters for the methods, inheritance, composition, etc. The detailed design is where you create an Entity Relationship Diagram (ERD) for the database structures you need to persist including keys, foreign keys, triggers, data types, etc.

In previous classes, you probably had to do a detailed design to the point where you were probably thinking to yourself, “this is a waste of time, I’m practically writing the code without writing any of the code”. And you probably were. In fact, there are some tools that will take this level of design and convert it into code so well that all you have to do is add the logic inside the individual methods.

Most of the popular IDEs will do the opposite. They will take code and reverse engineer the detailed design from it, creating the class diagrams, etc. This is sometimes called Round Trip Engineering because you start with a design, generate code, modify the code during maintenance, then regenerate/update the design from the code, thus coming full circle. The reason this feature exists is that one of the easiest things to neglect is the connection between detailed design documents and the code implemented. That connection breaks often because the implementation will evolve as it is created and even more so as it is maintained and updated over time.

Sometimes, when the same developer is creating the UML diagrams AND the code, they will just write the code and then reverse engineer the UML from the code. This might be considered cheating but it does result in a design that matches the implementation. I won’t name any names. And, I plead the fifth so as not to incriminate myself.

Now, what could go wrong with this idea? Well, having the detail design match the implementation does what? It simply documents how it is implemented.
It does not ensure that it is implemented to do what it is supposed to. Does the new design map to the high level design which, in turn, maps to the requirements. Our document might be accurate with relation to the code but our solution may be off base.

This is a helpful technique or trick to have in your pocket but use it with the full knowledge that it DOES NOT make what you wrote CORRECT.

Software Engineering

According to Kinser

Implementation

Once you have the design phase complete, all you need to do is write the code (the fun part), right? Wrong.

The implementation phase includes more than just code. You have to create stored procedures to define and populate your database(s). In a classroom, you probably open a tool and manually execute SQL commands to create tables, define triggers, insert rows of data, etc. In a professional setting, this has to be done using stored procedures (kind of like a script for a database) because you have to perform these same actions in a variety of environments. If you try to do them manually, there is a very real possibility that a step will be missed or performed incorrectly. A stored procedure will perform the task the same way every time. In addition, the stored procedures can be version controlled just like code can in tools like GitHub (we'll cover these concepts in the Source Control chapter). You cannot put an entire database in a source control system because the database is a binary image.

The implementation phase includes more than just code. You have to create scripts to set permissions, create accounts, create file structures, create/populate resource files, and other system settings and constructs. Scripts are used for these types of tasks for the same reasons given above for stored procedures. While you could do the tasks manually, it is prone to error when the same tasks have to be repeated over and over. Scripts can also be managed in a source control tool like GitHub.

The implementation phase includes more than just code. You have to generate documentation such as user guides, programmer guides, installation guides, etc.

Environments

The implementation phase includes more than just code. You have to set up your environments for development, test, staging, and production. You've probably only ever used a development environment and then submitted your code to the instructor. In a professional setting, code is initially developed in the development environment and promoted to the test environment as key components or stages are completed. Once functional testing is done, the solution moves to the staging environment where non-functional testing often occurs (e.g., load testing, failover testing, stress testing, etc.). Eventually the solution moves to the production environment. We'll discuss what happens in the environments more in the chapter on testing. Just be aware that if these various environments don't exist prior to the implementation phase, they will likely be created during the implementation phase.

Engineering Code vs Writing Code

For the coding part of Implementation, there are engineering practices that should be followed. We don't just start slapping code together to see if it works. Most professional

settings are creating solutions that are complex and require many people to work on them, so it won't work for everyone to just sling code around without a process. The SDLC is core to every modern SE Model/Process (e.g., Waterfall, Iterative/Incremental, XP, Scrum, etc.).

As mentioned earlier, it is impossible to write code without a design. A robust design depends on well formed requirements. Most classroom problems are simple due to time constraints and you might have done the design in your head (consciously or subconsciously). This won't work on a project. You need to study the design that was generated prior to this phase so that you have not only an understanding of the specific elements you will implement but also how those elements fit into the larger solution.

You need to follow coding standards. These are created so that all the code has a common look and feel in order to make it easier to maintain over time. Remember, in the professional setting, the solution you will work on will likely have to keep working for years and will need to be maintained and updated by someone other than yourself. If your team doesn't have a coding standard defined, write one. If they have one that no one uses, update it to make it usable. If they have it and use it, then use it.

Following a coding standard doesn't stifle your creativity. It helps you avoid common mistakes. It helps you understand other people's code more quickly. It helps avoid conflicts when merging code from different authors. It helps with long term maintenance. The coding standards typically address things like naming conventions for files, classes, methods, variables. It defines indentation, white space, error codes vs status codes vs exceptions, when and how to log events, class sizes, and so on.

Commenting Code

One of the most misused and underused aspects of coding is commenting. You've probably been told to "comment your code". Were you ever told how to do it effectively? I've seen student submissions that have a big header block with author name, date of creation, date of last change, comment about the last change. This type of commenting is a holdover from the 60s and 70s. Source control system eliminates the need for this type of comments.

If you think comments in code are useless, then you are doing it wrong.

To write effective comments, describe why you wrote what you wrote. If someone was sitting next to you trying to learn what your code does, what would you tell them? "This is a for loop." No. You would tell them what the code does and why you are doing that logic in this area of the code. This is what makes for a good comment. If they don't know what a for loop is, no amount of commenting is going to help. If you clutter your code with useless comments, no one will be able to find the useful comments.

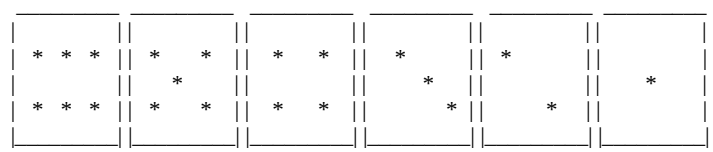
Be selfish when writing comments. Unless you want to be stuck maintaining the same code your entire career, document it well enough so that another person can maintain it, enabling you to be promoted to bigger and better things. Write to your audience: the

developer that maintains it, the developer that upgrades it, and you (because 6 months from now, you won't remember this code and might not even recognize it as your own).

Write your comments to be read by someone else. Don't write it like a shortcut note to yourself. Use complete sentences. Avoid abbreviations only you know. Avoid sarcasm. I often write my comments first as an outline for the code I will write later. I don't write pseudo code. I write prose. I focus on the critical elements of what I have to do. I don't write things like "add a setter for myStatus variable". I will write something like "collision detections only need to account for common geometric shapes such as circles, rectangles, and triangles."

Another example of commenting I did for a Yahtzee program:

> Each die is 12 chars wide * 7 high * 2 die + 2*7 CF/LF
> NOTE on design: I took a line by line approach. So I draw the top line of each die in turn. At the end of each line, I place CF/LF to go to next line on the display.
>
> There are only 3 lines that depend on the die values (first, middle, third). And the third line is just the reverse of the the first line, or a mirror image (since I show a six as two horizontal lines): reference images below



> Notice that only odd numbered values have a dot in the middle row and there is only one dot there ever.
> Also, all even numbered values have a dot in the upper left corner

Can you tell what language I implemented the solution in? How and Why don't usually depend on the language unless you are using an uncommon aspect of the language.

Happy Path

Don't just code the happy path. The happy path is the logical path through your code to do what you want as long as everything works properly. Almost every block of code you will write will require inputs (as parameters into the code, as data read in by the code, as variables known to the code, etc.). If you are using Objects and Pointers, you need to check for NULL and make sure it is clear who "owns" them (as in, who should destroy them). If you get what you expect from these inputs things work. This is the happy path.

The unhappy path is the logical paths through the code that are executed when you don't get what you expect as an input. How you handle the unhappy paths is critical to the quality of the code you write. This is often called error handling. If the error is possible and probable (e.g. human enters invalid data) then you should handle the error

gracefully and keep running. If the error is possible and improbable (e.g. can't connect to the database) then you handle it with exceptions so that your program can either handle the issue and keep running or exit in a controlled fashion. Only runtime or system exceptions should result in a core dump (like out of memory).

An Example (in pseudo code)

```
// parse a 14 character string of numbers into 3 substrings
String[] parseStringIntoThree (String inputString) {
        String[] subStrings = new String[3];
        subStrings[0] = inputString[0..5]; // get first 6 numbers
        subStrings[1] = inputString[6..9]; // get 4 more
        subStrings[2] = inputString[10..(inputString.size()-1)]; // get last 4 numbers
        return subStrings;
}
```

There is

- no validation on inputString to determine if it is not null
- no validation on it to ensure it has 14 characters or that the characters are numbers.
- no indication of what is returned if there is invalid inputs
- no indication as to whether the caller then owns the array and its contents returned.

The majority of unhappy paths should be identified during the design phase. If you look at how Use Cases are defined, the sequence of actions in the use case outlines the happy path. But, there are "alternate pathways" as part of the use case structure which identify what should happen if the happy path can't be followed. Test Driven Development (TDD) is big on identifying both the happy and unhappy paths by writing test cases for every path. In TDD, the code is not "done" until all the tests pass which means all the paths are correctly implemented (ideally). Still, the coder must be vigilant because change is constant and the needs might have morphed since the design or the tests were created.

Addressing both the happy and unhappy paths is key to building a robust, fault tolerant, well engineered solution.

Logging

Logging refers to writing out information about an application's behavior to a text file or a database. Logging can help engineers to understand how the application works in production, troubleshoot bugs, monitor performance, and audit user actions. Logging can also facilitate debugging by providing contextual information about the application's state and execution flow. A company's coding standards likely define when to log information, what information to log, and where to store it.

The larger and more complex an application is, the more helpful effective logging can be in quickly diagnosing the problem and implementing a quick resolution. This amazon article (https://docs.aws.amazon.com/prescriptive-guidance/latest/logging-monitoring-for-

application-owners/introduction.html) does a good job describing the kinds of events that should be logged and how often etc. It doesn't really address logging levels. Most logging tools support different levels of logging. The reason you have different levels is so that you can filter the log files quickly and to allow you turn different levels on and off at runtime. If you log absolutely everything all the time, the log files will grow quickly and become too big to manage easily.
Another great article is https://betterstack.com/community/guides/logging/log-levels-explained/ which explains the various levels of logging.

1. Emergency (`emerg`): system is unusable.
2. Alert (`alert`): immediate action required.
3. Critical (`crit`): critical conditions.
4. Error (`error`): error conditions.
5. Warning (`warn`): warning conditions.
6. Notice (`notice`): normal but significant conditions.
7. Informational (`info`): informational messages.
8. Debug (`debug`): messages helpful for debugging

Normally, your runtime environment will only have the first 4 levels active. In the dev and test environments, you may elect to have all the levels on (but you don't retain the logs for very long).

Some tools allow you to send each level of logging to the same or different files. This can be useful in that you retain the first 4 levels for long term reference but you only retain the last 2 (informational and debug) for 48 hours (for example). That way, if a critical event occurs you will know from the first 4 levels and may find important clues in the lower priority levels (notice, info, debug).

It is very important to be mindful of what data you log when events happen. It is possible to inadvertently disclose personal or private data. The Amazon link provides a pretty good description of the types of data you should NOT log.
https://docs.aws.amazon.com/prescriptive-guidance/latest/logging-monitoring-for-application-owners/logging-best-practices.html

It is very convenient to use logging tools but, like with any other third party tool, you have to be careful. Log4J is a common free utility for Java logging which has a vulnerability (https://en.wikipedia.org/wiki/Log4Shell) .

Code Inspections and/or Pair Programming

A code inspection (sometimes called a Code Review) is like having someone proofread your essay before you submit it to the professor. Having a second set of eyes look over your work is always a good idea. It may not be fun since most of us don't like to be called out for making mistakes, but it will reap benefits almost every time.

Software Engineering

According to Kinser

In my experience, a code review by an experienced person will find issues in the code faster than testing. This is especially true if the same person is writing the tests that also wrote the code being tested because any logical fallacies they had when they wrote the code, will still be there when they write the tests. That is to say, it's hard to find the errors in your own thinking.

For a code review to maximize its benefits,

- Code is submitted for review when complete but before it is committed
- Should be reviewed same day or next day at the latest
- Should be reviewed by someone more senior and then one other person (peer or junior to the person that wrote the code)
- Reviewers should take multiple passes through the code
    - Does it meet coding standards and is well commented
    - Does it fully meet the design
    - Is it well written (maintainable, extensible, follows good OO practices, etc)
- The author should be able to implement all changes in a day or two at most
    - If not, that indicates they need training or they submitted too much code for review

Some common mistakes that can often be found in a code review:

- Off by one (aka fence post error)
    - If you have 10 fence panels, how many fence posts do you need?
- Not initializing variables at point of declaration (don't count on the system)
- Not checking for Null when using pointers
- Assuming an array size instead of using the size() method
- Starting an array index at 1 when they start at 0 (or vice versa)
- Catching and ignoring exceptions (or just rethrowing with no action)
- Having more than one exit point from a method

Here's an interesting video Austin Mahala (SE I Fall 23) found on coding practices:
https://www.youtube.com/watch?v=GWYhtksrmhE (from Austin Mahala SE I Fall 23)

The person doing the code review should be the Devil's Advocate. This means you try to point out what could go wrong with the code as written. As the reviewer, you should be polite, non-threatening, supportive, encouraging, and open to being wrong. Everyone makes mistakes. We often learn best from the mistakes we've made. The reason a senior person should be doing the reviews is because they have probably made the most mistakes and will recognize them in the code. It isn't because they are necessarily smarter, just experienced.

Pair programming is a popular development technique especially when training is being

combined with an assignment. One person is the “driver” and one person is the “navigator”. The driver sits at the keyboard. The navigator is not a silent observer. They are the Devil’s Advocate for the driver. They collaborate with the driver on the implementation. They research the design, identify unhappy paths, perform real-time code reviews, solicit help from others as needed, etc. The navigator is tasked with trying to help the driver be as productive as possible. The navigator can be the one being trained or they can be the one doing the training. In fact, it is a good practice to switch roles periodically (whether halfway through the day or each day). In the medical field, they “watch one, do one, teach one” as part of their training. As the navigator, you can watch someone use a tool or technology or technique, then switch roles to “do one”. Later, you can teach someone else.

I’ve also seen a variant of the pair programmer where both individuals work at their computers, side-by-side. One person is writing the code and the other is writing the tests. They collaborate throughout the effort to ensure both are as robust and fault tolerant as possible.

Everyone makes mistakes. The important thing is to try to find them early on. Code Reviews are a good way to do this. Code reviews will likely become even more prevalent with the advancements in AI. Instead of having a person review the code or pair with another human, developers may pair up with an AI partner that helps write mundane code (setters/getters, simple methods/functions, etc.) and also reviews, in real-time, the human created code that is more complex.

Unit testing

You’ve probably been told to make sure you test your code before turning it in. Were you ever trained on the proper way to test it? Most students will “test” their code by running it. Only a few students will actually plan out different inputs to force the code to follow the unhappy paths. Even fewer will write a test driver; an actual executable that invokes the target code in a predictable way through its defined interfaces. This is often referred to as Closed Box testing.

In the professional setting, unit testing is the last one. Every class or unit of code has a corresponding test driver that invokes the target code in a variety of ways in an attempt to maximize code coverage (i.e., hit every possible logical path through the code at least once). These test drivers are configured in the source control system just like the target code is. The drivers can be collected in a suite of tests which can be automated as part of a regression test every time the solution is updated.

It is nearly impossible to achieve 100% code coverage by simply invoking the target code via a test driver. For example, how do you force an exception to be thrown in order to test the try-catch block in a method? In situations like this, open box testing is necessary. Open Box testing is when you go directly into the code and force a particular path to be followed. This can almost always be done manually. Sometimes it can be done programmatically.

Personally, unless you are building a mission critical, business critical, or life critical solution, I don't think unit testing is worth the effort. As stated earlier, if the same person that wrote the code, also writes the test, then it is likely any flaw in their thinking is mirrored in the test so it won't find it. Also, with the use of static code analyzers and proper code reviews along with functional and integration testing, I contend that the unit tests won't find anything of value that won't be found otherwise. With the advancements in AI code analysis and AI code completion, unit tests will be even less valuable in the future.

When are you done with Implementation?

As we will cover in Project Management, it is important to have a definition of "done" so that everyone working on a solution has a common understanding of what is needed in order to be "done".

Just getting it to work is not sufficient especially if you are taking a subjective approach. If it "works on my machine" but doesn't work when anyone else runs it, then it isn't really done even though "it works" (in an isolated case).

If the solution works great but if there is no documentation on how to use it or how to support it or how to maintain it, is it really "done"?

There is no universal answer to the question, when are you done. Each project has to define what it means to be done. Make sure you know what that definition is.

More often than not, the solution will have to satisfy the requirements (at least the mandatory ones). How do you know if it does? Use a Requirements Traceability Matrix. The team needs to be able to verify all the requirements have been met. The best way, the most reliable way, is to have a set of test cases for each requirement that demonstrates the requirement has been fully met (in the happy path as well as the unhappy paths). A matrix (or table) should have been created during the design phase to ensure that all the requirements had been addressed in the design. This same matrix can then be leveraged and extended to include a mapping of all the implementations for that design. If all of the design can be shown to have been implemented in the table and all of the designs are covered by the implementation then the implementation has addressed all of the requirements (if A=>B, and B=>C, then A=>C).

In order to test or verify this is true, the unit tests do not really play a role. Unit testing is primarily focused on ensuring that the unit of code performs as expected (as designed). In Test Driven Development, the tests are written before the code is. This is usually at the unit level but it can also be done at the requirements level as we'll cover in the Test Phase section of this chapter. So, at the end of the Implementation phase we have not done all of the requirements based testing (that happens in the test phase). Thus, the only way to know if we are "done" is to make sure we have implementations for all of the designs that map to all of the mandatory requirements. Yes, it's pretty much as boring as it sounds but it is necessary for formal projects especially Waterfall ones where we have one shot at getting it right.

Gold Plating is real. Some developers love coding so much that they could work on the implementation until forced to stop. They can keep optimizing the code or apply new techniques or new technologies, reworking the same code for a very long time. Perfecting the code for the sake of the developer's personal satisfaction is a recipe for project failure. On the other hand, not doing a quality job can result in costly maintenance and even project failure. How do you know if the code is "good enough"?

Test

One of my favorite questions in SE is: Who tests the testers?

In Waterfall, testing doesn't usually start until implementation is completed. RUP (Rational Unified Process) altered this practice by starting the test phase concurrent with the requirements phase. As requirements were defined, test cases could be written to validate them. This practice is probably where Test Driven Development got its start.

In Agile development, testing happens continuously. This is possible because Agile promotes sustainable development (Principle #8) and technical excellence (Principle #9). The automation of testing is key to these two principles. In order to automate the test, it has to be written using a testing tool (e.g. Selenium) or written as a stand alone executable that invokes the solution being tested through its interfaces. In both cases, the tests are almost always Closed Box tests. The question then becomes, what constitutes a test.

Verification and Validation

Testing is an overloaded term. In the test phase, testing primarily addresses both verification of the quality of the solution and validation of the solution back to the requirements. Verification testing is anything that evaluates the quality of the solution and the completeness of the solution. Verification can be done with visual inspections of the components in a solution (e.g., a design review, a code review, etc.). Some verification is done in each phase of the SDLC. Requirements analysis is a form of verification. Performing a design review and tracing the designs to requirements for completeness are both forms of verification. Interestingly, reviewing the tests themselves and tracing them to the requirements are forms of verifications. Some verification activities can also be automated (e.g., static code analysis, code coverage, code complexity, unit testing, integration testing, etc.). Any of the forms of executable tests that aren't validating requirements being met, are forms of verification.

Many development shops have an independent group (sometimes called Quality Assurance or QA) that is responsible for the validation activities. Validation tests primarily follow the pattern of "for a given set of inputs, an action is initiated, and the actual results/output is compared to the expected results/output". Validation tests are any testing which demonstrates that one or more requirements (whether functional or

non-functional) have been met. Most requirements involve at least two test cases for validation; one for the happy path and another for the opposite or unhappy path(s).

The test environment is where the QA team performs most of the automated testing. As features or key developments are completed, the solution is moved from the development environment into the test environment. The test environment isolates the testing activity from the constant changes being made in the development environment. If the code base is changing while testing is happening, it's much harder to identify what failed and why. Once these tests pass, the solution moves to the staging environment which mirrors as much as possible the production environment. In staging, load testing, performance testing, and a variety of other tests are performed (primarily around the non-functional requirements dealing with the "ilities" as discussed in the requirements chapter). If these pass, the solution can move to production.

Some of the most common types of tests:

1. Unit tests
   Consist of testing individual methods and functions of the classes, components, or modules
2. Functional tests
   Focus on the business or functional requirements at varying levels within the solution such as the unit, package, subsystem, or system level.
3. Integration tests
   Verifies that different modules or services used by your application work well together with respect to data flow and communications/connections.
4. End-to-end tests
   Replicates a user behavior with the software in a complete application environment.
5. Acceptance testing (not to be confused with Acceptance Criteria)
   Formal tests that verify if a system satisfies key requirements.
6. Performance testing
   Evaluate how a system behaves under a particular workload.
7. Smoke testing (aka Installation Testing)
   Check the most common Happy Paths. Often done before repeating more extensive testing as a way to ensure the system is set up correctly.

Defects

As a natural result of testing, defects will be found. After looking through the list of testing above, it is realistic to expect a large number of defects will be found. You probably found a few defects in code you wrote at some point. There are two strategies of what to do when defects are found:

1) Fix them as you find them
2) Document them all then fix them

I'm a fan of the second one because not all defects are created equal. Some are more significant than others. In fact, most Dev Teams have a classification system for defects that looks something like this:

Critical:
: The defect affects critical functionality or critical data. It does not have a workaround. Example: Unsuccessful installation, complete failure of a feature.

Major:
: The defect affects major functionality or major data. It has a workaround but is not obvious or is difficult. Example: The ability to upload files doesn't work but they could be e-mailed to support and uploaded manually.

Minor:
: The defect affects minor functionality or non-critical data. It has an easy workaround. Example: The logout button doesn't work but if you close the browser, the user is logged out.

Trivial:
: The defect does not affect functionality or data. It does not even need a workaround. It does not impact productivity or efficiency. It is merely an inconvenience. Example: A button label is misspelled; "Submitt"

A critical error, sometimes called a priority 1 error, is something that prevents the system from going to production or, if it is already in production, requires an emergency patch. While different teams have different lengths of lists or different definitions of severity, they all have the highest severity in common.

How you handle the non-critical errors is often subjective. I've been on projects where customers insisted on having no critical and no major defects outstanding before they would accept the solution. This, of course, led to many efforts to rationalize that some of the major defects weren't really major in reality. For example, who determines if the functionality is Major or Minor? Is every mandatory requirement a major functionality while optional requirements are minor? Or can some mandatory requirements be minor while some optional requirements (if implemented) be major? Is a derived requirement major or minor? Are non-functional requirements major or minor? Most contracts don't distinguish between major and minor when the requirements are written.

This is a great example of the type of situation where it is good to have a single decision maker who is authoritative enough to have their decision stand and also knowledgeable enough to make good decisions.

Sometimes the requirements are rather subjective, resulting in the testing group potentially taking a different interpretation than the development group. If the requirements were not SMART, this subjective conflict is more likely to occur. But people write requirements, and people are far from perfect. So, who decides which interpretation is right? Or maybe the right interpretation is different entirely? Again, it helps to have a single decision maker to settle these conflicts.

In my experience testers and developers didn't always get along. One company I was a developer at set productivity measures in place. Developers were measured on the number of defects found in the code they wrote. The more defects, the less productive

they are. Testers were measured on how many defects their testing uncovered. The more defects they found, the more productive they were. This put the two groups in direct conflict with each other. Testers would report a defect and developers would try to argue them away as not really being a defect.

When I was the director of development, I only tracked the defects that were found in production. If a defect of any kind was found and fixed before production, then it didn't exist because it never affected the user. This approach aligned the testers and the developers to find and fix as many defects as they could before the solution was released.

How will your team identify, classify, and keep track of the defects you find? Here is a good article on this if you are using the Scrum methodology: https://agilethought.com/blogs/4-ways-handle-defect-management-sprint/

Enhancements

In addition to defects being documented and tracked, it is important to document and track enhancements. Like defects, all enhancements are not created equal. Some enhancement requests are actually indications of a lack of training for the users. They request something that the system already does or does in a slightly different way than requested. Some enhancements are out of scope for the solution as it would be in conflict with existing requirements. For example, I had a user submit an enhancement request to be able to configure the inactivity timeout because they didn't like being logged out automatically after 5 minutes of inactivity. This conflicted with a security requirement. Other enhancements are great suggestions and can actually add more value to the solution than other planned upgrades. Because a lot of enhancement ideas come from users, it is important to treat them with respect and seriousness. The submitter should always be acknowledged and, if at all possible, kept current on the status of their idea. It makes for a happier customer base.

One more story: I took over as the director of a product development group where there were few formal processes in place. After about six months of productivity improvements, we started releasing fixes to reported problems in addition to adding new features (the previous director ignored defects if they weren't critical). As a result, we saw a significant increase in defects being reported by our customers. My CEO was unhappy, but I told him this was a good sign. The defects were valid issues and have existed for a long time. Customers just didn't waste their time reporting them because they saw nothing was getting fixed. Once they saw that we were fixing issues, they started reporting them again. After confirming my position with a couple of key customers, my CEO was more supportive of the process improvements I was initiating.

Release

Releasing the solution into production is not the end, nor is it usually easy to do without some planning in advance. Performing a release is like giving a speech. You have to

have a plan of how you will do it, all the steps involved, and practice, practice, practice. You don't practice in the actual production environment. You practice using the same deployment steps/scripts when you deploy into a staging environment because a staging environment mirrors (as closely as possible) your actual production environment, including current snapshots of databases.

Messing up a release into production can be a career limiting move. Plan it and practice it. And have a contingency plan and practice that too.

The release process can include scripts to do the actual release into the production environment. If not the first release of that product, it should also include scripts to handle upgrading the current environment to the new one. For example, if the current production environment has client data and history, this data needs to be ported to the new version (and possibly massaged to meet any new formats). There should also be a rollback script in case the release fails. A rollback script (like with Databases) will ensure the prior release can be fully restored in case of the new one failing. There are several tools like Jenkins and Docker that help manage the release process and orchestrate the script execution.

As the last step of a release, many development groups will perform a Smoke Test (or Installation Test) to verify the release was complete and successful. The smoke test is usually made up of read-only type actions that involve as many of the components/systems as possible. The key is to verify that all the components are running and communicating with each other. But, it is important to do this in a way that does not change any of the system states or data. For example, running a report from the UI will verify that accounts can be logged into, data can be retrieved from the persistent store, and displayed accurately to the user, and then that the account can be logged out of. This operation hits most of the horizontal layers of the solution while not changing any state or data.

Another story: As a developer with a reputation for fixing critical issues quickly, I was pulled into another project to help identify the cause of a production issue that the team couldn't replicate outside of production. After asking several clarifying questions, it turns out that a key production table was not populated and this caused a catastrophic exception to be thrown and the system crashed. They couldn't replicate it in testing environments or staging because all of these environments had test data in that table. We added a single entry in the table in production and the system booted without issue and ran as expected.

One last production story: In the same company, another project was being deployed as an embedded system but when they went to deploy it wouldn't fit on the target platform. None of the test environments had been configured to mimic the size of the platform. They were all much larger and only tested functionality. After 3 days of digging into the system, myself and three other developers were able to reduce the size of the applications so that it fit. To the surprise of many, we didn't make any code changes. We just removed unnecessarily link'ed in libraries. Turns out, one of the project developers had created an IncludeAll library that linked everything in because it was easier than figuring out which libraries were really needed and which ones weren't. Their approach

was copied by several other developers on that team. The end result was that almost every single executable was bloated. Now, if you were in charge would you insist that all the tests be rerun or would you waive that since no code was changed? Let me know if you think you have the right answer and why your answer is correct.

Maintenance and Upgrade/Update

If the solution will ever get used, it will have to be maintained because any solution of any real complexity or value has defects in it. A well implemented solution will have no high priority defects but it will have some defects. Part of the reason for this goes all the way back to requirements. It is nearly impossible to anticipate how the solution will actually be used. Also, the production environment may have evolved in the time between requirements and release. Also, designers and developers are human and prone to miscommunications and mistakes and assumptions.

If the solution is successfully deployed and meets the targeted audience's needs, you can be sure there will be a need to add enhancements thus additional releases with upgrades and updates will likely follow.

These two phases often last the longest. A solution that is being used has to be supported forever right? That isn't realistic. At some point, the solution or the older versions of it will need to be discontinued. This is often referred to as end-of-life (EOL) for a solution or version. Many product companies will announce that a solution or version is being deprecated first. Being deprecated means there is limited support. Often, no more defects except critical ones will be addressed and training will be discontinued. After a certain period of time, usually one year, the deprecated solution or version is declared EOL. This means there is zero support or training available for it. Microsoft has been around a long time and you can research how many of its solutions have been declared EOL. For example, they don't support Windows 3.1 or Windows 8.

Because these two phases last the longest, they represent the greatest cost to the supplier even though fewer people are involved than in the creation of the solution. It may take 50-100 people to build and release a solution, while it only takes 25-30 to support it initially and this may drop lower over time. However, the creation of the solution may only take a year or less, while support can continue for a decade or more. Any supported product requires more than just developers to support it as well. There are marketing, sales, training, technical writers, etc. involved during that long phase as well.

If a solution has strong demand, then the maintenance work is often done by a different team than the updates/upgrades and may occur on a different schedule. Maintenance involves defect fixes and minor enhancements. Updates/Upgrades involve the release of new features and capabilities or support for new platforms/environments. Maintenance releases might be done on a monthly or quarterly cycle with emergency patches done as needed. Upgrade releases will typically be slower to release, maybe annually.

One group I ran released every two weeks. We alternated between maintenance items

in one release and updates in the other, cycling back and forth in a predictable way. This worked because our solution was deployed in a single hosted environment. If a solution is deployed to thousands of client machines, this might not be the best idea because you don't want to change the look and feel of a product causing users to have to retrain on a regular basis. Thus, it might be better to keep maintenance separate from updates, allowing clients to choose when they take either at a time of their choosing.

SDLC Example

Example of the SDLC of a real-world solution: The creation of the 4250 course followed the SDLC waterfall model.

Requirements were gathered by extracting the Learning Objectives from the previous semester's syllabus (these are functional requirements). I then did some research online and talked with Dr. Dubay about learning styles and instructional practices which provided some of the non-functional requirements. I interviewed the 4350 instructors to find out what skills and knowledge they expected 4250 students to have going into 4350. These were additional functional requirements.

The high-level design draft was reverse engineered from the previous instructor's slide decks. I then mapped these (Requirements Traceability Matrix) to the Learning Objectives (LO), added in new topics to address remaining functional requirements such as Jobs, Tuckman's model, etc. I selected the Flipped Classroom modality and session structure (video, attendance, Q/A, activity, quiz) to address the non-functional requirements. I reviewed my high-level design with Dr. Bennett (key stakeholder) verbally and made a few adjustments based on his feedback.

The Detail Design included mapping the session topics to the semester calendar to address logical flow of instruction and to leave room for the semester project at the end. This resulted in the Course Outline document you see in GitHub. Notice it includes a second trace back to LOs. I also tracked in this document if the content had been reviewed for a final time (verification activity), which sessions had a quiz written for it, and what optional materials were associated with each topic (if any).

The implementation is where I wrote the slides and presenter's notes, and the in-class quizzes for each class session. Some topics required additional research to refresh my memory and to capture more current terminology. After the first draft of each topic was done, I did my first review of the content and created the class exercise for each. These exercises are documented in files named "Class Exercise - <topic name>" for ease of reference. After I finished each exercise, I tried to act it out at home by myself or with help from my wife in order to get a timing and find flaws in the instructions (Verification; a sort of unit test). I then sifted through YouTube videos for something fun and related to the topic that I could start the class with.

The final implementation task was to create the syllabus, setting Job Description, and

course details in writing for students; a sort of ReadMe file.

While verification was happening throughout (as described above), I did a final, tedious walkthrough of the entire course, making final edits. Concurrent with this Validation Testing, I solicited a review by two previous instructors of the course. For every topic, I now had a slide deck entitled <topic>, instructors notes entitled “Class Exercise - <topic name>” and a quiz “<topic> quiz”.

The final testing occurred in a Staging Environment which was a Sandbox account on D2L that Dr. Dubay set up for me. I used the D2L interface to create modules for each class session, attaching the appropriate slide deck(s) (optional and required readings), converting text quiz into D2L quiz, setting start and due dates, etc. This took longer than estimated due to the quirkiness of D2L. I viewed the content from multiple perspectives (Instructor vs Student, Outline vs Calendar, etc.).

To release the course, I exported my sandbox content and then imported it into the two sections of 4250 setup for the Fall ‘22 semester a few weeks before classes were set to start. I repeated the review from multiple perspectives as a type of smoke test to ensure all the dates and times lined up as expected (they didn’t and several adjustments had to be made manually). The last release activity was to post a welcome message to each section’s news feed.

Support/Maintenance occurs throughout the semester. A day or two prior to each class, I review the slides and the class exercise document to identify any adjustments needed. After each class session, I make note of any feedback from students. If a minor change is needed (e.g. spelling errors, clarifications, or suggested article/video links), I make it. If a more significant change is needed (e.g. changing a question on a quiz is not realistic once students have taken it; I have to wait until the semester is done), I add it to a document called “course updates” which is my defect/enhancement tracking tool.

Upgrades/Updates are identified throughout the semester by my observations but also from the course retrospective I conduct with all the students the last week of the semester. These are reviewed and approved items are added to the course updates document. I work on these changes between semesters so that they are fully implemented in time for the next semester. An example upgrade was the addition of Topic Quizzes to 4250 which were created to bridge the gap between the in-class quizzes (which are very easy) and the LC quizzes in 4350 (which are very hard), and to assess student’s understanding of key topics.

Chapter 8 - Waterfall

As mentioned at the start of the last chapter, Waterfall is a development methodology that has very specific roles and processes defined. The SDLC phases are executed in strict sequential order. Each phase is completed before the next one begins. Some examples of the types of projects that benefit from the waterfall approach:

Software Engineering

According to Kinser

An Airplane
A Satellite
A Mars rover
A Microwave
A Car
A pacemaker

All of these examples are deployed in a way that makes changes to them extremely difficult and rare. You may have also noticed that the majority of the examples are either Mission Critical or Life Safety; that is to say “they absolutely have to work” or people will die or great harm will be done to the business. ALL of the functionality has to be in the production release or it doesn’t fill the intended need. You can’t just implement the landing related systems of a plane, just the telemetry portion of a satellite, just the timer for a microwave, just the cruise control for a car, or just the timer part of a pacemaker.

Waterfall projects are typically a single pass through the Lifecycle model with one big release at the end. Thus, a lot of time is spent in each phase verifying and validating the artifacts for that phase before the project can move forward. There probably isn’t any such thing as a fast waterfall project (though I did run a few as project manager that were completed in 6 months).

The Waterfall method is one of the first models for structured development (vs the chaos of the early years).

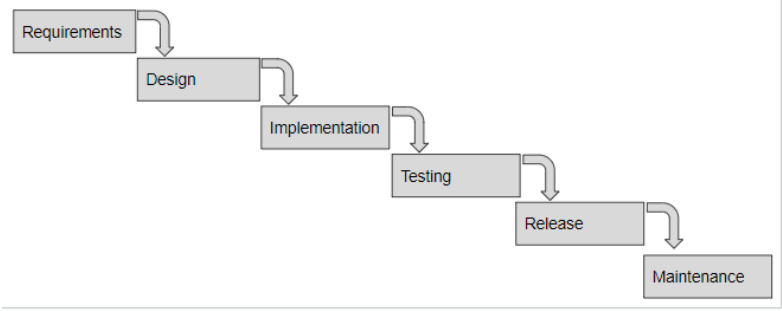
When I attended a training class for Scrum, the instructor lambasted Waterfall as a big mistake and the absolutely wrong way to build software. He said Agile was the best way. I argue that both have their place. The important thing is to understand the nature of the work you have in front of you and pick the best model (not the one with the most books written about it).

In many ways, all the models are just variations on the Waterfall model (including Agile/Scrum). You may notice this as we go along. Somewhere along the line, people pulled the lifecycle out of the waterfall model and made it independent of it. I would suggest they are one and the same.

Below is a pictorial representation from which waterfall gets its name:

Software Engineering

According to Kinser



Waterfall defines roles like Project Manager, Architect, Developer, Tester, Business Analyst, etc. It often depends heavily on formal project management activities. It defines artifacts for each phase:

- Requirements Stage
  - Software Requirements Specification (SRS)
  - Systems Specification (may be a section in the requirements document)
  - Acceptance Criteria
- Design Stage
  - Architectural Design Document
  - External API specification
  - Network Design
  - Systems Design Documents (one for each system in the solution)
  - Detailed Design Documents (one for each system)
  - System and Integration test plans
  - Acceptance test plan
- Implementation Stage
  - Software including scripts and stored procedures and unit tests
  - Network installation document and scripts
  - Acceptance, Systems, and Integration test cases documented
- Test Stage
  - Release Plan
  - Acceptance, Systems, and Integration test results for all pre-production environments

Chapter 9 - Iterative/Incremental

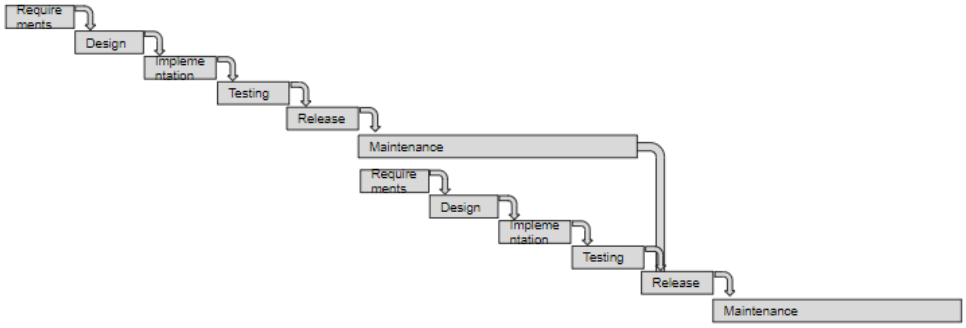
The iterative and incremental models often start out looking like a waterfall project, except there isn't a single release at the end. Instead, a series of iterations are planned of equal duration which build on the previous release. More often than not, these iterations follow the same formal process as Waterfall projects. The iterations often are measured in months and can even run as long as a year or two. Some examples of solutions that would likely use this method:

Software Engineering

According to Kinser

An Operating System (e.g. MS Windows)
A downloadable game
Accounting Software
An ERP like PeopleSoft or SAP

The first release of any of the examples here might have used waterfall to achieve some minimum of functionality (an MVP or maybe just a bit more) but it was expected all along that there would be additional releases that would expand on the feature/functionality of the product without additional purchase (i.e. they planned on upgrades and enhancements in addition to the normal fixes and patches). The original set of requirements are scheduled across multiple releases of the solution. You might think of it as a depth first approach where the full functionality of a few features are released then the next set, etc. (loop until done). With each iteration, the requirements can be refined, the design massaged, and the implementation refactored in addition to adding the new functionality.



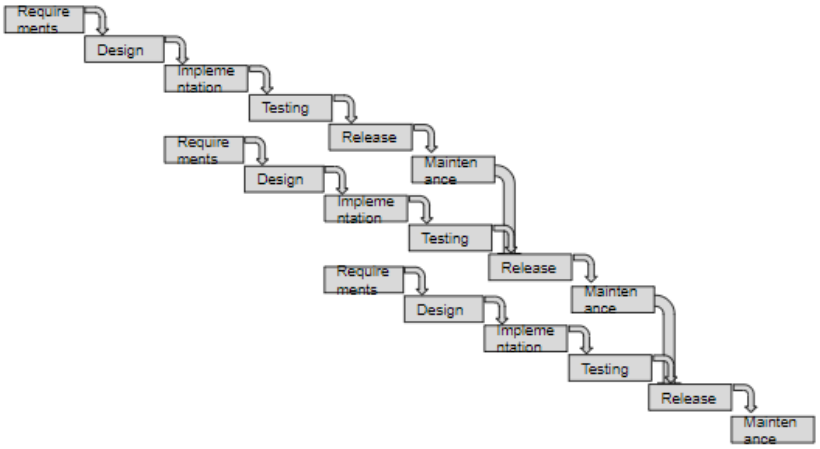
The examples above represent solutions that are routinely enhanced with expansions on existing features as well as new features not present in the previous version. These solutions evolve over time. However, the updates are not made frequently, usually annually though some might happen faster and some slower. Also, upgrades are usually manual and sometimes time and labor intensive. Thus, multiple versions may exist and have to be supported concurrently by the provider.

For simplicity, I've grouped Spiral with Iterative though some would argue they are different enough to be kept separate. I put forward that when compared to Waterfall and Scrum, the iterative and spiral models look very similar to each other. There are also other variations on Iterative that I won't call out for the same reasoning. Just know that there are LOTS of different models out there (and few are followed exactly as defined by their authors).

For example, there is a variation called Overlapping Iterative where the second iteration starts before the first one ends:

Software Engineering

According to Kinser



Notice how the requirements phase of the subsequent iteration starts concurrently with the implementation phase of the previous iteration. This works well because the staff involved in requirements are more often than not, starting to free up at the end of the design phase. Also, notice that the maintenance phase of the previous iteration ends with the release of the subsequent iteration. It is a bit of an oversimplification because not all phases are equal in duration even though each iteration is.

The majority of projects I ran as project manager fit into this category (i.e., iterative or overlapping iterative) because the requirements were defined ENOUGH to set the architecture and gain enough business value from the initial release to justify moving forward. The change management and risk management helped define the contents of subsequent releases along with the refinement of the requirements not yet implemented. By having a periodicity to our releases, we were able to scope how much could be done in each one.

Chapter 10 - Agile

I've chosen to place the Agile chapter in between Iterative and Scrum because it is a project management philosophy that is appropriate for both models. Some people call Agile a SDLC model but it has no defined roles or process and cannot be a model. Agile is a mindset, a point-of-view, an approach.

You can get more information on Agile at https://agilemanifesto.org/ but the key elements to understand are the Agile Manifesto and the 12 Agile Principles. The Agile Manifesto is:

Software Engineering

According to Kinser

"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 Process 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."

A quick history might help in understanding the manifesto. It was created in 2001 by a group of experienced developers who tried to find the common pros and avoid the common cons of the various SDLC models of the time. In a nutshell, they felt there had to be a better way of doing things but one model would never be a one-size-fits-all solution.

The important thing to remember with the manifesto is that last line "That is, while there is value in the items on the right, we value the items on the left more." So, processes and tools are important (especially in critical solutions) but they shouldn't be used so dogmatically that you burn out the people involved. Employee retention is rarely addressed in a SDLC model but it has tremendous value. Having experienced staff, familiar with your business model, your client's needs, and overall operations of organization and staff has tremendous value. If you have constant turnover, the rate of errors will skyrocket.

Documentation is necessary in order to be able to effectively communicate and retain information for the long term. We recognize that employee retention is important but if all the knowledge is in people's heads, disaster is simply one exit away. When dealing with customers, the "thud" effect is rarely reassuring to most clients (the "thud" effect is when you have printed out all the documentation the team created and drop it on the table in front of the client to show them all your "progress"). Clients are more impressed by seeing the solution, even if what you show them isn't the full solution.

America is a litigious country and no two independent parties will ever do business with each other of any significant value without a contract. So, it is important to have a contract, build a relationship of trust with clients so that the contract is more of a formality than a stick one side uses to beat the other into compliance. Collaboration sometimes means compromise. Each side should recognize that neither is perfect. The

key is to have regular, open communication where the focus is on the solution and not "winning" in contract negotiations.

That collaboration is usually based on the recognition that things change. In some ways, the software world changes very quickly. A five year project will find that the hardware originally spec'd out for the solution is likely to be OBE'd (Overcome By Events). Clients change their minds or discover a new need based on their ever changing market. While it is always good to have a plan for how the solution will be built, the more open all parties are to change and adapting that plan to change, the better off the project will be. I've been on more than one project as a developer where we were forced to work overtime and even holidays in order to achieve some "non-negotiable date" which eventually was negotiated to change once reality set in. In fact, I worked Christmas day several times the first five years of my career. That's the wrong way to run a project.

From this manifesto, 12 principles were defined. They are listed below. Do not memorize them. Instead, you should internalize them. I encourage you to understand them in the context of whichever SDLC model you are working with. One way to do that is to paraphrase each one in your own words. It will take some experience in the field before some of these principles will make sense. I have found all 12 to be quite accurate and useful.

1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4) Business people and developers must work together daily throughout the project.
5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6) The most efficient and effective method of conveying information to and within a Dev Team is face-to-face conversation.
7) Working software is the primary measure of progress.
8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9) Continuous attention to technical excellence and good design

enhances agility.

10) Simplicity--the art of maximizing the amount of work not done--is essential.
11) The best architectures, requirements, and designs emerge from self-organizing teams.
12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

I've tried to apply the Agile mindset to the way the two SE courses are executed. For the class exercises and projects the teams were not assigned. I asked you to form your own teams (with certain constraints). I did not assign roles. I asked the whole team to do the exercise and left it to each team to self organize. The course has a plan (as documented in the Course Outline) so everyone knew what to expect but we adjusted where necessary (e.g. some impromptu lectures or presentations, or videos). There was quite a bit of classroom time devoted to letting the teams get work done which created an opportunity for the exchanging of ideas (e.g., cross-team collaboration). Every session followed the same pattern (in SE I it was required reading, video, attendance, Q/A, brief lecture, and an exercise). Thus, each class is like an iteration that builds on the previous one. The lessons are more conceptual and less dogmatic (I don't ask you to memorize things, I ask you to understand them). Frequent feedback on weekly status reports helps each student know where they stand individually in the course and encourages small adjustments over the length of the course instead of late night cramming to catch up.

Hopefully you can see which of the above principles come into play in the description of the course plan. In a similar way, these principles can be applied to Waterfall projects, Iterative projects, XP projects, Scrum projects, etc. Agile is independent of the process you choose to follow. Agile is independent of team size or project complexity. It is a philosophy.

Chapter 11 - Scrum

Scrum is a SDLC model that was created to put the Agile Manifesto and Principles into a process. It does define roles and artifacts. It is a model. It is a methodology. Is it synonymous with Agile? No. It is entirely possible to ignore the people and get caught up in the dogma of the process. In fact, there are two competing organizations that have their own definition of Scrum: Scrum Alliance and Scrum.org. It turns out that the two people that created Scrum couldn't collaborate with each other and got a "divorce". They couldn't even agree on who gets the kids (Scrum) and we all suffer because of it.

Software Engineering

According to Kinser

Anyway, Scrum is a model that works well in projects that are not life critical or mission critical. It works for projects where the solution is rapidly evolving in response to changing markets and needs. For example:

- An informational portal or retail site on the web
- Cloud based solutions like Google, Facebook, TikTok, etc.
- Custom solution for a highly involved client
- Any Web enabled device that accepts downloads

These examples share the characteristic of being hosted remotely from the client or solely by a single client. They are generally NOT mission critical or business critical or life safety related. Sometimes referred to as a General Application. If the Google search engine goes down, it will frustrate some people, but it probably won't kill anyone. It probably wouldn't even cause Google much harm as a business (assuming it comes back up within a few hours). Same with Facebook (which does go down every now and then much to the chagrin of the sellers on their marketplace). Not the same reaction as when the power goes out across the state of Texas in the heat of the summer (you do know that the electric grid is managed via software written by people just like you).

Agile is, in my opinion, a natural evolution of the Iterative model that was enabled by advances in hardware and technologies like the internet. Hard to believe, but we did use the previous models prior to the internet (for many years prior). Agile does not require the internet but the internet did create a HUGE number of solutions that could easily be updated rapidly without negative impact to users (e.g. informational websites, shopping sites like Amazon, search engines …).

One of the critical aspects of Agile is for the customer (meaning the person(s) that decides the requirements) to be readily available if not co-located with the Dev Team. Another key aspect is for the team to be 5-11 people in size (small in SE terms) but they are expected to be experienced and cross-functional. We will cover more on this later.

Given the popularity of Agile and Scrum, you may already be familiar with them. You may even intern at a company that follows one or both. The reality is, just like with the other SDLC models, most organizations don't follow Agile or Scrum completely. They "tailor" it to their specific needs and situation. So, if you are already familiar with Agile or Scrum, you may find yourself having to "unlearn" some of those customizations. The key to successfully tailoring either is to fully understand them first so that you know what you are giving up and weigh that against what you are gaining.

For example, teams are supposed to be self-organizing. Some took that as self-forming. Most companies have a hard time letting teams form on their own. Companies are often organized like the military with a top-down command structure. Managers define who will be on the team. To self-organize in Agile terms is for the team to be involved in deciding what will be in a release and for each team member to pick which tasks they will take on. Managers have a hard time with this too because they think they know who should work on what. But if you take self-organizing away then it's "sorta" Agile but take a few more aspects away and it really isn't Agile at all.

Software Engineering

According to Kinser

If the organization says they are using Scrum, but their sprints differ in duration then they aren't really doing Scrum. If their dev team has an architect or technical lead, then they aren't doing Scrum. If the line manager (the person that decides raises) is also the Scrum Master or the Product Owner, then they aren't really doing Scrum.

In the Scrum model, decisions about scope/priority/severity can all be made very rapidly; questions of design or implementation can be resolved in real time; and releases can be made quickly (usually on a weekly or bi-weekly schedule but could be as often as several times a day). It can be almost mind boggling. You rarely start with a significant set of requirements, maybe just an idea with a handful of requirements. The architecture is flexible in that you can change what the architecture is. There is only enough design done to enable implementation to begin.

Scrum Roles

So, let's get into it. A Scrum Team has only three defined roles:

- The Product Owner - the voice of the business
- The Scrum Master - the process coach/mentor
- The Developers (aka the Dev Team) - responsible for the quality of the solution

The Product Owner is the voice of the business. They are responsible for working with the stakeholders and clients to identify and refine the requirements for the solution. They set priorities for any work that needs to be done: new features, enhancements, bug fixes, etc. Most Scrum Teams leverage User Stories for their requirements. The Product Owner will write most if not all of these, but this is not a hard and fast rule. They do need to be able to explain the User Story and write the acceptance criteria associated with it. The Product Owner is ultimately responsible for ensuring the solution meets the business needs.

The Scrum Master is an optional role (in theory). They are responsible for coaching the rest of the Scrum Team (the Product Owner and the Developers) on the Scrum model and the Agile mindset. They do this continuously throughout each sprint through one-on-one and group interactions. A common technique is guidance through queries like what therapists do. For example, if the Dev Team asks if a task is considered done, the Scrum Master might respond by asking "Do you feel it is done?" or "What criteria does the team use to determine if something is done or not?". The Scrum Master is not a project manager. They don't report to management on the team's progress. They are facilitators, coaches, mentors, advisors. They are not in charge. If the Scrum Team is at the Norming or Performing stage of Tuckman's Model, the Scrum Master might not even be needed anymore.

The Developers is somewhat of a misnomer. We often think of Developers as people that write software. They develop code. In Scrum, a Developer is someone that develops the solutions. They could be an IT person, a software developer, a tester, a UI/UX specialist, etc. Collectively, the Developers, the Dev Team, will work collaboratively to accomplish whatever is needed to build the solution. If no one on the team has the

necessary skills for a task, the team works to gain that skill. They are self-organizing in the Agile sense. The best person for a given task may not be the person that can do it best, but it might be the person that is most interested in doing it. For Scrum to work most effectively, the Dev Team should be composed of experienced persons where each is versed in a variety of skills. They need to be Hard Working, Self-Starters, that Learn Quickly and are interested in Success. That is to say, they need to be a self-motivated team.

The Scrum Team shares responsibility for the solution. I often refer to Scrum as “a communist lovefest” because there is no single person in charge. There is no project manager. There are no team leads. The whole Scrum Team works together in equal collaboration.

Scrum Artifacts

Scrum has several define artifacts:

- The Product Backlog
- The Sprint Backlog
- The Definition of Ready
- The Definition of Done
- Potentially Shippable Product

The Product Backlog is like a requirements document, except it’s not. It’s more like a collection, a sorted list one might say. Everything in the Product Backlog is referred to as a PBI (Product Backlog Item). A PBI could be a User Story, an enhancement request, a defect report, etc. The highest priority PBIs are at the top of the backlog. The prioritization is not strict. It is more common for a small group of PBIs to be considered to be of equal priority to each other. While they are interchangeable with each other from a priority perspective, they are distinguished from other PBIs or groups of PBIs. The Product Owner “owns” the Product Backlog. They are responsible for refining it; adding PBIs, removing PBIs, changing the order, breaking a PBI into smaller PBIs, defining the acceptance criteria for each PBI, etc. The backlog is a living document. I will note here that the backlog is supposed to be visible to anyone within the organization and that anyone should be allowed to add a PBI to it. The Product Owner still owns that PBI so they must work with the author to fully understand it and evaluate where it belongs in the backlog. The Product Backlog is like a project plan at a high level.

The Sprint Backlog is the list of tasks that the Dev Team will work on in each Sprint. The Dev Team owns the Sprint Backlog. They define the tasks that go on it. They update the status of the tasks. They manage the Sprint Backlog. Everything the Dev Team does during a sprint should be represented in the Sprint Backlog. Everything in the Sprint Backlog should be “doable” within a single sprint. The Sprint Backlog, like the Product Backlog, should be visible to anyone in the organization. This backlog is like a project plan at a detailed level. We’ll discuss how this backlog is populated later in this chapter.

The Definition of Ready is created by the Scrum Team to set the criteria which

determines if a PBI is ready to be brought into a sprint to be worked on. This can be a simple statement that the team has reviewed it and unanimously agrees it is ready. It can be much more complicated with metrics and measurements. It is up to the team to decide. Once defined, it is not static. It can be updated to allow for continuous improvement.

The Definition of Done is created by the Dev Team to set the criteria which determines if a sprint task is considered complete and ready for release (aka potentially shippable). Like the Definition of Ready, the Definition of Done can be a simple statement that the Dev Team has reviewed the task and unanimously agrees it is “done”. It can be much more complicated with specific tests that must be run (in one or more environments), code reviews performed, code analysis performed, etc. The Definition of Done represents the Dev Team’s commitment to quality. Because of this, it is not static. It is regularly reviewed and updated to allow for continuous improvement.

The final artifact is a Potentially Shippable Product. At the end of each sprint (and ideally, every sprint), all the tasks that the Dev Team marked as “done” are merged into a release that the Dev Team considers Shippable. That is, to the best of their knowledge, it works. It is not actually released until the Product Owner agrees that all the acceptance criteria for each of the PBIs included in the release have been met.

Scrum Activities

We’ve covered the Roles and the Artifacts of Scrum. Now we will describe the Scrum process that ties all these together into the Scrum Model. The Scrum process consists of the following activities:

- Sprint Planning
- Daily Scrum
- Sprint Grooming
- Sprint Review
- Retrospective

For context, it is important to emphasize that each sprint is the same duration. The Scrum Team determines this duration based on their needs and those of their client. Typically, it is one or more weeks but can be as long as a couple of months (see Agile Principle #3). Because they are fixed in duration, all the activities within the sprint need to be timeboxed as much as possible or they will eat into the time the Dev Team has to build the potentially shippable product. They need to work at a sustainable pace (see Agile Principle #8). This is referred to as their velocity (the amount of work they can consistently perform in a single sprint).

Every sprint starts with a Sprint Planning activity which is timeboxed and facilitated by the Scrum Master. The longer the sprint duration, the more time is needed in Sprint Planning because you have to plan more work. The opposite is also true. The inputs into the activity include a Sprint Goal and a list of PBIs that support that goal. The Sprint Goal is defined by the Product Owner prior to the Sprint Planning activity. The goal

speaks to the business value the Scrum Team is trying to achieve in this sprint. It provides context for the Dev Team to set priorities of their own tasks and to make decisions as the sprint progresses. The sprint goal is not a restatement or summation of the PBIs. In my experience, Scrum Teams that are still in the Forming and Storming stages of Tuckman's model, don't usually find much value in the Sprint Goal and often don't use them correctly.

At the start of Sprint Planning, the Sprint Goal and associated PBIs are presented and explained by the Product Owner to the Dev Team in priority order. The Dev Team collaborates with the Product Owner (by asking clarifying questions) to break each PBI down into tasks. Remember the discussion on Problem Solving. A problem is something without an obvious solution and a task is something that can be done. A PBI is usually not small enough to be a task by itself so the Dev Team must do a detailed design in order to define the tasks that must be performed in order to complete the PBI. How formal the team does the detailed design is up to the team to decide. After each PBI is broken down into tasks, each task is estimated by the Dev Team. The total of those estimates is compared to the Dev Team's velocity to ensure they don't sign up for more work than can be done, given what is known at that moment. If the estimates exceed the velocity, something must be traded out for a PBI that can be completed. If the estimates are below the velocity, something can be added. The Dev Team should strive to have a full sprint where they use all their time wisely to add as much value each sprint as possible.

At the conclusion of the Sprint Planning activity, the list of tasks is placed on the Sprint Backlog in priority order (based on the priority of the PBIs provided by the Product Owner). How the Dev Team manages this backlog is up to them. Many teams use a KanBan style board. The tasks in the backlog are NOT assigned to individuals during Sprint Planning. By adding them and working them in priority order, the team can better ensure they are maximizing value to the business (if something doesn't get worked on, it is likely of lower priority).

Daily Scrum is a short activity timeboxed to no more than 15 minutes and often takes less time. The Scrum Master usually facilitates this activity. In the Daily Scrum each Dev Team member will answer three questions; 1) what tasks they completed since the last Daily Scrum, 2) what obstacles to progress, if any, are they facing, and 3) what tasks will they work on between now and the next Daily Scrum. The Daily Scrum, as the name implies, occurs daily throughout the sprint at the same time, in the same location. It is meant to keep the members of the Dev Team current on each other's activities so they can coordinate, collaborate, etc. as appropriate. Every member of the Dev Team should attend every Daily Scrum. The Daily Scrum is limited to just stating the answers to the three questions. If discussions are needed, they should occur after the Daily Scrum and only involve the people required (not necessarily the whole Dev Team). The Product Owner can attend the Daily Scrum, but they are expected to be a silent observer. Similarly, the Scrum Master facilitates the activity, but they don't answer the three questions. This activity is just for the Dev Team.

The first Daily Scrum of a sprint occurs immediately after the Sprint Planning activity. Because the Sprint Planning activity does not include assigning tasks to anyone, the

Daily Scrum must take place for the Developers to select which task they will work on. Notice, they select what they will work on. No one should be assigning a task to someone else. It's a communist lovefest.

After that first Daily Scrum, the sprint is underway. What happens throughout the duration of the sprint is dictated by the Dev Team in how they execute their tasks (but the Daily Scrum is executed every day, at the same time, in the same place).

It feels a little out of sequence, but I place Sprint Grooming before Sprint Review for several reasons which hopefully will make sense after the following explanations. Sprint Grooming is the activity where the Scrum Team reviews the highest priority PBIs in the Product Backlog to determine if they meet the Definition of Ready. At a minimum, a PBI is only ready if the Dev Team estimates that it can be implemented in a single sprint or less time. The Product Owner can't determine this because they don't do the work. The Dev Team can determine this by doing something like a High-Level Design. They won't actually define the individual tasks needed (that is done in Sprint Planning as explained earlier).

Because Sprint Grooming occurs within the timeline of a sprint, it must be timeboxed. I recommend that teams fairly new to Scrum plan to have a Sprint Grooming activity every single sprint. This allows them to have a consistent velocity and to adjust to changes in the backlog as well as the volatility of their estimations.

The Scrum Master facilitates Sprint Grooming, but the Product Owner should know which PBIs need to be groomed prior to the start because the Product Owner "owns" the Product Backlog and all the PBIs in it. They know if there have been any changes in scope or definition of existing PBIs and whether there are new PBIs that have risen in priority. The Product Owner will present and explain the PBIs that need to be groomed in priority order. The Dev Team will collaborate with the Product Owner to understand the scope and definition well enough to decide if it meets the Definition of Ready. If it is too large, the Scrum Team works together to break the PBI down into smaller PBIs (kind of like Derived Requirements). Each of these smaller PBIs should be self-contained such that they can be released individually if necessary.

Sprint Grooming is always focused on future sprints. It should not be executed for the PBIs in the current sprint. My recommendation is to have enough PBIs groomed that you have enough to fill at least two sprints and no more than four. You always need enough for two sprints because if it turns out the Dev Team finishes a sprint backlog earlier than estimated, they need to be able to bring in work to finish out the remaining velocity. So, you need at least two sprints worth of groomed PBIs ready to be brought into a sprint. You don't need more than four because Agile promotes the ability to respond to change (see Agile Principle #2). As each sprint is executed, the Product Backlog shifts up because the highest priority PBIs should be getting implemented. Thus, the team needs to groom more every sprint for near-future sprints.

I will point out here that some teams will do their sprint grooming in between sprints. This works basically the same and all the above guidelines and caveats still apply. Most importantly, they should NOT be grooming the PBIs they plan to take into the very next

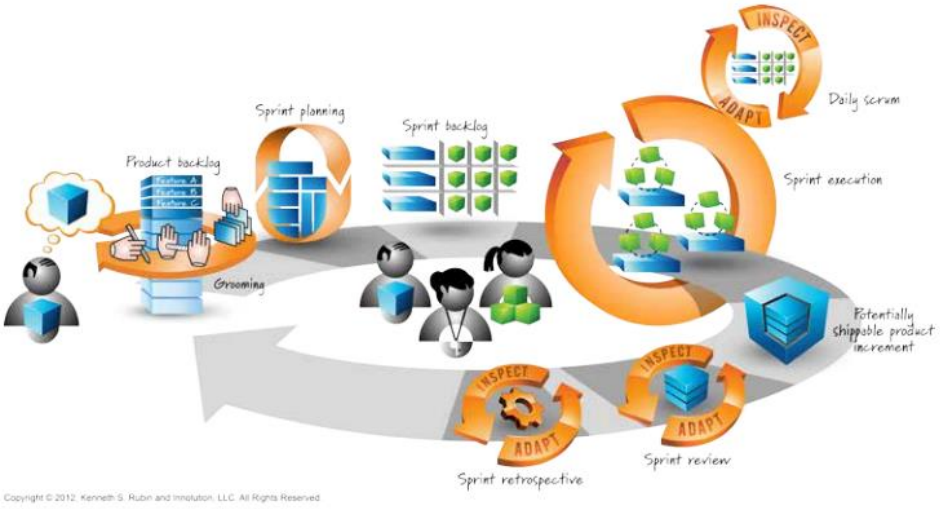
sprint.

The second to last activity each sprint is the Sprint Review. This is where the Scrum Team will evaluate the Potentially Shippable Product against the Acceptance Criteria. The Sprint Review is open to stakeholders and interested parties. While dialogue is encouraged, it is time boxed, and the Scrum Master needs to facilitate to ensure the goal of the activity is met. Ultimately, it is up to the Product Owner, as the voice of the business, to determine if the work from that sprint will be released to customers or not. If the Product Owner approves it for release, the Dev Team will execute that release after the Sprint Review but before the Retrospective. It can't be done before the Sprint Review because the Product Owner hasn't approved it for release before then. It should be done before the Retrospective so that the team can include their experience with the release and release process in the Retrospective.

The final activity in each sprint is the Retrospective. This is the activity that embodies Agile Principle #12. The Scrum Team will reflect on the most recent sprint. Each person will put forward what they feel went well (that the team should continue doing) and what could be done better. The Scrum Master facilitates the discussions. Ultimately, the Scrum Team picks one thing they want to continue to do well and one thing they want to improve. Based on this selection, they must then decide if changes need to be made to the process and what that change must be. For example, they may decide that code reviews needed to be done more quickly once code was ready for review, so they automate the notification of code reviews and have a timer to escalate to the team if it doesn't happen in a timely fashion. It isn't a real retrospective if all that happens is people point fingers at each other to do better with no plan of how to do that. That is called whining.

Software Engineering

According to Kinser



From ScrumAlliance.org

The above diagram is from the ScrumAlliance.org website. I like it but there are issues with it. I like it because it's easy to follow and visualize. The bald guy is the Product Owner that has an idea for a product and captures it in the product backlog. The pencil, ruler, and card catalog illustrations around the backlog represent how the Product Owner is always refining and prioritizing the PBIs, and the Dev Team estimates the higher priority items. These PBIs feed into Sprint planning where the team evaluates some set of PBIs and breaks them down into tasks which feed into the sprint backlog. The larger orange circle represents daily work which starts with the Daily Scrum and culminates with a Potentially Shippable Product. The Sprint Review evaluates this to determine if it should be released or adapted/changed. And the sprint ends its cycle with the retrospective where the team evaluates their own performance in the sprint.

What I don't like about the diagram is that it is missing the Release activity. Even if it is automated, releasing a solution to its production environment is no small endeavor and should be represented in some way. I also don't like that they illustrate the Grooming as happening at the start of the sprint. You should not be grooming things just in time for the next sprint. In my experience, that leads to rushed jobs and not enough time to coordinate with other parts of the organization like sales, marketing, support, etc. The Product Owner needs some amount of lead time to work with stakeholders to ensure their buy-in and support. Work should be planned, not a reaction to last-minute evaluation. I take this stance based on Agile Principle #9, technical excellence. I also

don't like that the avatars appear to be all white people and that the Product Owner is an old, white guy. Why can't they just be neutral?

Anyway, that's Scrum. It doesn't seem that hard, does it? Well, it is. It can take a team years to master. The fastest way I've found to get a team to be really good at Scrum is to start with a group of experienced people (with at least 5 years of industry experience though 10 would be preferred), with a track record of agility, give them lots of freedom and support (tools, tech, etc.) and sell them on the purpose and value of the work they are taking on. Making this happen is no easy task either, but it accelerates the benefit of using Agile and Scrum.

Now, let's check to see if you are really starting to internalize all this stuff. If Sprint Grooming occurs in the middle of a sprint, what does a team base their first sprint planning on? Think about it. The Product Owner owns the product backlog and decides which PBIs to bring into the first sprint. However, the Product Owner cannot groom the PBIs by themselves. Thus, there must be some type of initial grooming that occurs so that there are enough groomed PBIs in the product backlog so that the team can execute the first sprint planning activity. This need may be why the above diagram shows grooming prior to sprint planning.

So, when does initial grooming happen and is it the same as sprint grooming. Well, it must happen before the first sprint. So, sometimes people refer to it as Sprint 0. Neither Initial Grooming nor Sprint 0 is defined as a Scrum activity or part of the Scrum process. But it must happen, or you don't have what you need to start the first sprint. Some groups say, you just start the first sprint and figure it out. I say this approach violates Scrum and Agile because there is an expectation that every sprint generates a potentially shippable product. So, I say that Initial Grooming occurs prior to Sprint 1 but is not a sprint and is not timeboxed. It takes as long as necessary. Initial Grooming is more than Sprint Grooming. It starts with the Product Owner giving the Dev Team an overview of the project and desired solution. They explain the value of doing it well. They then conduct a walkthrough of the entire Product Backlog with the Dev Team, so the Dev Team has an idea of the "big picture" of the solution being asked for. The Dev Team should be asking clarifying questions throughout this process (because they are interested in success). As part of this process, they will likely define a targeted architecture design. Once the entire Scrum Team is ramped up and in agreement on what needs to be built, they then conduct a Sprint Grooming activity for the highest priority PBIs until they have enough groomed items for at least two sprints but no more than four. I recommend having closer to four, so they have plenty to get started with. Initial Grooming should not be timeboxed, and this includes the Sprint Grooming portion of it since it is not being done as part of a Sprint.

Software Engineering

According to Kinser

That’s the end of the content for CSCI 4250. If you aren’t in SE II, stop reading right now or you’ll get distracted with all the cool stuff we will cover in CSCI 4350.

Software Engineering

According to Kinser

CSCI 4350

The following chapters are written primarily for CSCI 4350 students and presumes the reader has completed CSCI 4250. These chapters will cover some of the same topics as earlier in the book but will take a more “robust” perspective.

Software Engineering

According to Kinser

Chapter 12 - SDLC more robustly

The Vision Phase

The vision phase is not commonly included in the SDLC. However, it has always been present in every project I have ever been involved in. I think this is because there are typically no software people involved in the vision definition process and it has traditionally been done by the business side of an organization.

When you start gathering requirements, what do you start with? There must be something there that tells you who the stakeholders are that can help define the requirements. There must be some forward-looking statement that justifies even pursuing the project (to justify the investment). That is the vision statement. The vision isn't any more perfect than any set of requirements you've been given in the past so ASK CLARIFYING QUESTIONS about it so that you understand the vision and the words behind the words.

In more recent times, some of the newest and biggest companies have been started by software people or in partnerships with software people. This means our profession is getting more and more involved in the business side of the development phases.

The vision statement is usually a formal document that addresses several key items needed to get a project started. An executive summary will describe the goals of the solution/project in terms of impact to the business (e.g. capturing market share, increasing revenue, decreasing costs, etc.), what the projected budget is, what the targeted timeline and key dates are, who the stakeholders are (either by name or by department), references to research that might have been done that led to the solution/project proposal, supporting market data to help justify the effort, and a summary of the competitive landscape.

SE II Requirements

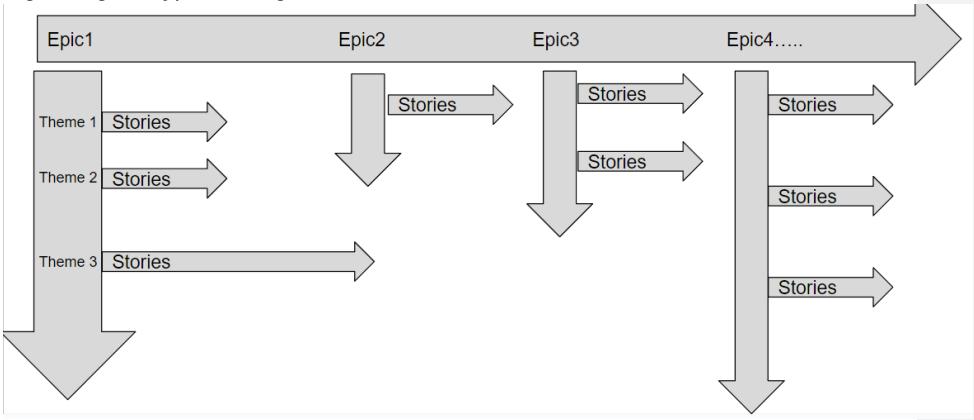
The precondition to the requirements phase is the vision document. As stated above, the vision document defines a very high-level vision for what the solution will do and why. This gives scope and direction to the requirement gathering effort. One of the essential steps in the requirements phase is to identify the stakeholders and decision makers.

Requirements don't magically appear nor are they obvious. It might be comforting to think that the "people in charge" know what they want and how to describe it. The reality is that they are no better at writing good requirements than developers are at writing good code. It takes practice. If that's not possible, someone that is well versed in requirements gathering and creation should be engaged to assist.

One of the biggest challenges in writing requirements is understanding who the audience is for those requirements. This is further complicated because there are

multiple audiences. The stakeholders themselves are one. The solution designers and builders are another. Whoever is funding the solution (aka sponsor) is a third, and there are probably others depending on the solution. Just these three will illustrate a key challenge in that they don't all speak the same language. The stakeholders are fully immersed in the business for which the solution will be built. The designers and builders are not typically acquainted with and rarely an expert in the same business space. The sponsor is normally an executive and acquainted with the business jargon at a high level but doesn't always understand the details of how things actually operate. Neither the stakeholders nor the sponsors usually understand the technology involved in creating the solution.

The stakeholders oftentimes have an idea of what they want but haven't normally really detailed out the lowest level needs. They usually think at a high to medium level of detail. One could call these Epics and Themes where an Epic is a major capability or functionality (e.g. navigation in a self-driving car) and that Epic is composed of multiple Themes representing components of that capability (e.g., a routing algorithm, a GIS system, a road condition alert system, etc.). The diagram below shows one way of organizing this type of categorization.



As requirements analysis is done, additional requirements are uncovered. These "derived" requirements could be at any level including a new Epic, but most of them will likely be at the level of User Stories. As each Theme is analyzed, multiple User Stories will be identified (e.g. a routing algorithm will need options for most direct route, fastest route, routes that avoid heavy traffic, etc.).

When conducting the analysis phase, it is usually helpful to identify who all the major actors are in the solution starting with the target user group. The use of storyboards is a common practice where each actor's involvement is described like a story or "a day in the life". This can help not only identify more of the User Stories that the actor is part of but also identify the dependencies that may exist between those stories. Some

developers create “personas” with great detail and an avatar to represent each actor in the system. For example, you might have a business application where you need to take into consideration the seasoned user that wants the most efficient way to get work done as well as the novice user that wants things clearly laid out for them to avoid confusion. These would be two different personas for the same actor role of “user”.

In 4250, everyone was encouraged to ask clarifying questions and there was a consistent theme of “there are missing requirements”. This was to prepare students for the real world where there really are missing requirements. They are missing not because someone is purposefully withholding them, but because they are either assumed to be known or were not thought of. Asking clarifying questions is one of the only ways to discover these before their absence has negative consequences.

One activity that I have found to be useful in requirements gathering as well as in requirements analysis is the Sticky Note brainstorming exercise introduced in 4250. This can also be used to brainstorm on the questions to ask during requirements analysis. In this case, you have the designers and builders writing the sticky notes for each question they have. There needs to be SME (Subject Matter Experts) in the room that can answer these questions as they are brought up. Alternatively, the SME can take ownership of getting the answers and providing them after the fact.

If you order the Epics in priority order left to right and identify the themes in priority order top to bottom, I recommend doing one or two passes across the board defining user stories. Then, focus on the highest priority Epic and detail it out as much as possible. This is consistent with the Scrum approach to grooming the product backlog. The initial grooming hits the entire backlog and subsequent sprint grooming focuses on highest priority items.

One thing to remember if you use the Epic/Theme/Story model is to NOT get hung up on whether a requirement is an Epic or a Theme or a Story. Epics don’t have to be written like a user story but they can be. The same goes for Themes. The value in the model is how it helps the team organize the requirements so that it makes sense and helps define a roadmap for development. Don’t get caught up in the semantics of the process, just focus on the value it brings. Remember the Agile Manifesto: We value individuals and interactions over process and tools.

Design

The more experience you get, the more you will come to value the design phase. I actually address Design in multiple chapters; Architectural Design and High-Level Design are covered in the chapter on MVP (Minimum Viable Product) and in the chapter on Grooming and the one about Emergent Architecture. Detailed Design is covered in the chapter on Scrum Planning.

As briefly stated earlier in the book, I have found that the design phase is most productive when tackled in a pragmatic way. There is no value in arguing over whether a diagram is a sequence diagram or an interaction diagram, or if you need a claw foot

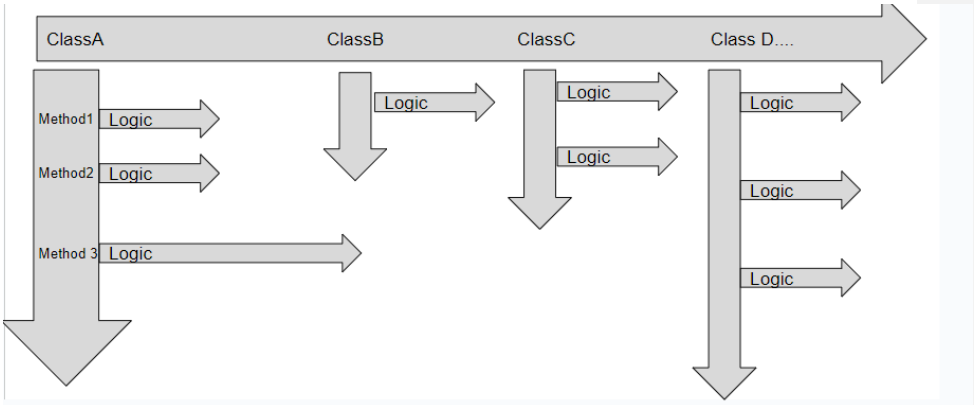
pointing this way or that. The key is to break the problem down further and further until it is understood well enough to take action on. Part of the value of Design Patterns and UML and similar tools/techniques is in the vocabulary it provides which, when shared by the team, helps communicate effectively and efficiently. If a majority of the team doesn't know UML, find something they do share in common as a means of communicating their designs.

Implementation

When I am coding a solution, I follow a similar pattern to how I do Requirements:

- I create a file for all the classes I might need (breadth first at the highest level)
- I create all the method signatures and stub out all the methods in the classes I will work on first
- I then write the logic in the methods as I implement a "slice" of functionality.

Notice that I do not fully implement a class, one at a time



When I create the files for the classes, this is when I do my comments as an outline for the class. So, when I come back to that class later on to add the methods, I have notes to remind myself what I wanted this class to do and how I had originally planned it to work.

As I create each method in the class, I'll add comments that outline what happens inside the method. So, when I come back to that method later on to add the logic, I have notes to remind myself what I wanted this method to do and how I had originally planned it to work.

Kind of like a fractal.

My technique or approach to coding evolved over the years. Originally, I wrote procedural code and only did the happy path. I got better at writing robust code. When I

started writing OO code, I did whole classes because that's what my architect told me to do. After a couple of years, I landed on this approach as it yielded the most productivity for me personally.

Notice that I might have identified methods in the early steps that I never end up implementing the logic for. Do you know which Agile principle this is related to?

10) Simplicity--the art of maximizing the amount of work not done--is essential.

Stubs

As part of testing, sometimes you have to "fake" results to force a specific logical path to be taken. These are sometimes referred to as Stubs. They are temporary code that is only enabled during testing. They can also help you if code you are dependent on isn't ready.

A stub can be enabled using environment variables. The default path should be to not execute the stub so that the absence of the flag yields the designed behavior, as in what you want in production. It can also be enabled via commenting (as in the example below). The risk here is that you forget to comment out the stub and send it into production so choose a behavior in the stub that is obviously not the happy path. It can be enabled via connecting to a different library of code. Etc.

```
        CALL rollDice ; roll the dice specified in bDiceToRoll, store in lpDice

; TEST CODE: in order to test the joker rule and yahtzee bonuses,
; comment out the line above "Call rollDice"
; and uncomment the 5 lines below so that you always roll 6s
        ; MOV lpDice, 6
        ; MOV lpDice+4, 6
        ; MOV lpDice+8, 6
        ; MOV lpDice+12, 6
        ; MOV lpDice+16, 6
```

In the example above, the MOV statements could have been inside the rollDice method as temporary logic before fully implementing the rollDice code. This would allow the rest of the program to be worked on and will likely be noticed if accidentally left in place because it would be unusual for a random function to generate all 6s every time.

Stubs are handy for testing but also as a risk mitigation tool when two or more developers are working on related code and they finish at different times. Those that finish early can use stubs as placeholders until the other developers finish their pieces. Similar risks exist here in that you might forget to take out the stubs once the other developer is finished. While development stubs can be used at a class or method level, it is more common in larger development efforts where an API is defined between systems or subsystems. If SystemA is dependent on SystemB but SystemB is not finished, then SystemA developers can stub out SystemB's API with default logic/values so they can continue with the development of SystemA concurrent with SystemB. Once SystemB is

ready, the real API replaces the stubbed-out version and system testing can begin.

Testing

In the 4250 section, there are a number of testing methods defined. You probably didn't do many, if any, of them with your SE I project. Running your code is not a testing method. To properly test something, you need to define a set of inputs and preconditions, perform some specific action, and then compare the actual result to a predicted result. That is to say, you need test cases.

For every requirement or user story you work on, there should be a design. As you design the solution you need to implement, you should be looking at both the happy path and unhappy paths. These are your test cases. When doing the design, if you decide that you need to handle the case where the user provides invalid inputs, then write one or more test cases where invalid inputs are provided and identify what you expect the solution to do in each case. For example, if a user attempts to login with the incorrect password, you probably will display an error message and allow them to try again. Do you force them to reenter the username first? How many times can they enter the wrong password before you block them and how long do you block them? Thinking through the test cases concurrently with doing the design will make both more robust.

Some testing approaches that can help you write good test cases:

- Boundary Value Analysis - test value at edges and on either side
  (input expects integer between 1 and 10 so test 0, 1, 10, 11)
- Equivalence Partitioning - where any item in a range is as good as any other
  (input expect integers so test a negative, zero, and a positive)
- Defensive Coding - test for the unhappy paths
  (i.e. try to break it through the interface)

Similar to what I said about design techniques like UML and design patterns, I don't spend much time arguing whether a test is a functional test, an integration test, a system test, etc. If there is a requirement, there needs to be testing done to fully validate that it has been met. How much testing and which technique you use is oftentimes a decision left to the team. More formal development, where the solution is life critical, mission critical, business critical, the amount of testing and the techniques are probably defined in the contract. For other applications, not so much.

This is a good time to remind you of the Agile Manifesto where it says,

- We value individuals and interactions over Process and Tools
- We value responding to change over following a plan

And the Agile Principles of

5) Build projects around motivated individuals. Give them the environment and support they need and trust them to get the job done.
7) Working software is the primary measure of progress.
10) Simplicity--the art of maximizing the amount of work not done--is essential.
11) The best architectures, requirements, and designs emerge from self-

organizing teams.
12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

If you are a hardworking, self-starter, who learns quickly, and is interested in success, you won't skimp on testing. Because you are a professional, you want to produce a quality solution (this is implied by the principles listed above). As a student you often feel like you have to turn something in even if it wasn't tested "to get the grade". In the workplace, you shouldn't release something (turn it in) until it is "done". It is better to be late than to claim false credit by releasing low quality work.

Later we will talk about DevOps and CI/CD. These topics promote verification and validation. They also emphasize the need to automate testing at all levels as much as possible. By doing robust testing in an automated fashion, the team will produce better quality faster.

Whose job is it to test the solution? I mentioned the QA (Quality Assurance) team of independent testers in the 4250 sections. This is somewhat of a relic of the old style of development and used on Waterfall and Iterative projects. The more Agile projects expect everyone to be involved at some point and at some level in testing the solution. It's a communist lovefest.

When you read about automated tests, remember that someone has to write them. They don't just show up. So, include in your estimates the effort needed to formally and robustly test your solution. If you can automate the tests as part of your task, then it will be much easier to identify any negative impacts future development has on your completed work. To automate your tests, you can write standalone drivers that can be independently executed against the code base. You could use tools like Selenium to automate testing through the user interface (https://mailtrap.io/blog/best-automation-testing-tools/).

Chapter 13 - MVP

Let's pretend we have a customer that would like us to build them a car.

If we are using the Waterfall method, we ask lots of questions to get a full, robust, and mostly unchanging set of requirements.

Why do they want a car? Answer: To get from point a to point b.
How do you get there now? Answer: by walking.
And so on.

Software Engineering

According to Kinser

Once we have all of the customer's requirements, we add in some because there are laws and regulations about what every car must have and must do. We define all the requirements for driving, transmission, radio, heat, ac, seat warmers,etc. It takes a couple of months to detail out all the requirements and we estimate it will take 3 years to build. We will start building a very detailed project plan to construct the chassis, engine, etc. Then we build each component and assemble it together.

After 3 years, we finish the car and drive it around a track to test it. After we fix some issues, we deliver the car, and the customer buys it. They need to buy gas each week, change the oil periodically (maintenance). We issue a safety recall (Update) and later they decide to add those cool spinning hubcaps (upgrade).

Using the Iterative model, we do the same as above except every six months we "deliver" something, but just to show progress since they can't actually use it. The first time we delivered the chassis and simple steering (no seats). We can demo it working (well, we can demo that the chassis is and that the steering system does something). The next iteration we add an engine and seats. The next time we deliver the transmission, etc. In the final iteration, we assemble all the components together.

After about 3 years, we finished the car, drove it around the track to test it, Yada Yada.

Using the Scrum model, if someone orders a car.

Why do they want a car? Answer: To get from point a to point b.
How do you get there now? Answer: by walking.

So, we build them a skateboard.

In Scrum, we don't start building the car. We don't deliver a wheel, then another, and then the steering column, then the seats, etc. Because each of these deliveries are NOT working solutions.

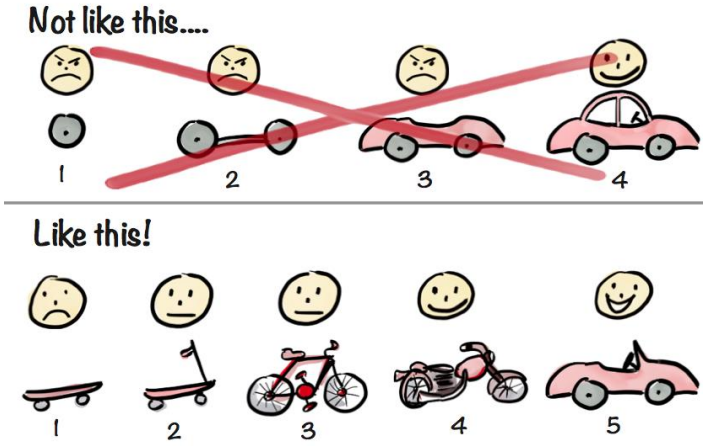
Instead, we would build them a skateboard right away. It's not the final solution but it is a working solution, and it will get them from point A to point B.

Next, we might put a small electric motor on it based on their feedback that it is too hard to make it go. Then we deliver an electric bike, then a motorcycle. It is entirely possible that upon delivery of the motorcycle they inform us that their needs have been met. They didn't really need a car. They just thought they did.

A minimum viable product is one or more vertical slices through the solution that provides a working feature(s) even if it is not robust, but it adds value to both the user and the business. It provides enough value to the user that you can collect meaningful

feedback that can drive future development. The use of MVPs is much more common in Agile projects than traditional waterfall or iterative development. So, this chapter assumes an Agile mindset in the discussion of MVP.

In the “build a car” scenario, we first gave the user a skateboard. This was an MVP. It gave them a way to get from point A to point B without having to walk the whole way (which was how they made the trek before). The skateboard was a working solution. It was a viable means of transportation. It wasn’t what they ultimately wanted but it addressed the same need as what they wanted, albeit minimally.



Henrik Kniberg

https://blog.crisp.se/2016/01/25/henrikkniberg/making-sense-of-mvp

I like this example because when I talk about a vertical slice through the solution, some people think of the top row in the diagram. They imagine a vertical slice is a portion of the final product like a car tire. But the bottom row illustrated a vertical slice better.

As you define your requirements and refine your product concept, you will probably have an idea of what your final solution should look like. Will it be a hosted solution, a downloaded application for mobile devices, an enterprise application, etc.? This is your target architecture. Will you build out that architecture upfront or will you evolve it over time? This goes back to the car/skateboard analogy. You may plan to ultimately have a downloaded application for mobile devices, but your MVP doesn’t have to be that necessarily. Knowing what you want as your final architecture is helpful so that you don’t do anything to make it harder to reach that goal but that final architecture may not really be needed until your solution hits milestones like market capture or a certain number of

users or some key feature that can only be supported with that final architecture. Building what you need now and allowing the architecture to evolve is further discussed in the chapter on Emergent Architecture.

I bring it up here because your MVP's architecture may be very different from your target architecture and it's important to understand this as part of your MVP activity.

Chapter 14 – Robust Source Control

I introduced you to source control slowly in 4250 by having you edit the attendance file from your forked repository. I also gave you links to materials and exercises online that help explain source control. I then walked you through an exercise step by step (unusual in comparison to how I led the other exercises). The chapter on source control in the 4250 section of this book describes how to do source control robustly. Ultimately, you should have leveraged source control on your semester project. Did you do it robustly? Probably not.

To do source control robustly in 4350, your team should have a shared repository where each dev team member is a contributor. When working on a project, I don't recommend having each team member "forking" off one person's repository like we did for attendance. Everyone should do coding in their own feature branch that stems from the common development branch for that sprint. The development branch should be tagged/named for the specific sprint. The feature branches should be tagged/named for the task from the sprint backlog that corresponds to the work being done. Each developer should pull in changes made by the other team members regularly so that their feature branch is current. Each developer should be working in small batches and completing their work in one sprint or less.

As tasks are completed and the sprint review is nearing, the dev team should have a test branch they can use to perform their testing and potentially allow the product owner to do some early acceptance testing. After the sprint review, the code should move from testing to a staging branch which mirrors production as much as possible (meaning that it should be deployable and not on a local personal machine). Once the release process you used to deploy to staging is verified to work (and you should follow a process), you can move the code to a production branch (or rename the staging to be production). The production branch should be tagged/named using semantic versioning.

Code reviews should be integrated into the submission process. Each time a developer is ready to merge their feature branch back into the development branch, the source control system should wait for code reviews to be performed and the subsequent corrections made before continuing. The developers should be submitting test cases or test drivers along with their code.

As you get more working software in place, your team should add the execution of automated regression testing and/or smoke testing to the submission process to ensure new changes didn't break existing working code (because the developers probably only tested their changes and nothing more). The release process described earlier should include automated testing as part of the release (testing before you release it and some non-state changing test after you release to ensure the installation was performed properly and the solution is in a good working state in the production environment).

If you had trouble following the above content then I suggest you reread the source control chapter in the 4250 section of this book.

Do I expect your team to do all the above in sprint 1? No. But the above is one example of a more robust way to engineer software solutions and it should be a goal for your team by the end of the semester. There are other processes that are even more robust but the above is attainable given where you are in your training.

Chapter 15 - Product Owner

The Product Owner is a member of the Scrum Team. The Product Owner is the voice of the business. They own the Product Backlog. They are responsible for the vision of the solution. They need to be readily available to the Dev Team during sprint execution. The Product Owner is NOT a Project Manager. They do perform some project management tasks such as planning but other project management responsibilities are distributed among the rest of the Scrum Team.

As a member of the Scrum Team, the Product Owner (PO) is just part of the communist lovefest that is Scrum. They need to have an Agile mindset. If they do, the more engaged they can be with the team on a daily basis the better. The Product Owner is not in charge of the team. The role of Product Owner is often filled by a business analyst or product manager type person and may not necessarily have a technical background or a background in project management.

As the voice of the business, the Product Owner needs to be adept at negotiating with stakeholders, requirements gathering, requirements writing (usually in the

form of User Stories), organizing the product backlog based on a combination of business needs and technical dependencies, and making decisions. While it is nearly impossible to simulate well in a classroom setting, the Product Owner in a professional environment will spend a lot of their time in meetings. They meet with stakeholders inside and outside the company. They articulate to all, a common vision for the solution and help manage expectations of progress. You might say that they are also the voice of the team to the larger audience of stakeholders. This last part is a bit controversial because in scrum the Dev Team is supposed to be able to talk directly with customers and stakeholders. The reality is that too many of those conversations can lead to misunderstanding, sowing confusion across the organization. These conversations can also negatively impact the productivity of the Dev Team. The Product Owner can run interference for the rest of the team.

The Product Owner is in charge of the Product Backlog and the vision that drives it. The Product Backlog is like the SRS (Software Requirements Specification). It contains the functional and non-functional requirements of the solution. It is a living document and often changes in some fashion on a daily basis. Anyone in the organization should be able to add a Product Backlog Item (PBI) to the backlog. When this happens, it must be easily identifiable by the Product Owner as being new and requiring review. The Product Owner would meet with the author to gain a deeper understanding of the PBI (i.e., the words behind the words) so that they can own that PBI going forward. Composing the acceptance criteria for the PBI is one way to paraphrase back to the original author what the intent is.

While the Product Owner is not the project manager, they do have the responsibility of high-level planning by ordering the PBIs in the backlog. By putting the PBIs with the greatest business value at the top of the backlog, the Product Owner is scheduling work to be done. Unlike traditional project management though, the order is a little soft and fuzzy. The PBIs may not end up getting worked in the same order they appear in the backlog. This is because the

Dev Team is responsible for the detailed planning of each sprint. The Product Owner will bring a Sprint Goal and a list of PBIs that support that goal to the Dev Team at the start of the sprint; during Sprint Planning. How many PBIs and which ones will be determined by the Scrum Team. See Sprint Planning for more information about Sprint Goals.

The Product Backlog does not consist of just requirements. A PBI can also be a bug report, an enhancement request, a refactoring idea, a research topic, etc. It is the Product Owner's job to balance all these different types of PBIs into a robust plan for the solution. If the Product Owner only focuses on functional requirements, the solution will become fragile due to technical debt and unresolved defects in the code and design. If the Product Owner only focuses on the technical aspects of the solution, then it won't be responsive to the user's needs.

The Product Owner also is responsible for scheduling the Sprint Grooming activity. The Scrum Team should always have enough groomed PBIs to fill at least two sprints but no more than four sprints (see Sprint Grooming for more details). Because the Product Owner owns the Product Backlog, they should be tracking which items have been groomed by the team sufficiently to be brought into Sprint Planning. Some scrum camps recommend having the Scrum Team create a Definition of Ready (similar to the Definition of Done used for sprint tasks in the sprint backlog) to outline what it means for a PBI to be "ready" for development. That is to say, that the PBI is "sprintable". My minimum definition is that the PBI (by itself) can be implemented by the Dev Team in a single sprint or less. If the Product Owner feels that there may be fewer than two sprints worth of PBIs groomed and ready, they can inform the Dev Team that a Sprint Grooming activity is needed. The best time to do this is during Sprint Planning so that the Dev Team can account for the impact to their velocity in that sprint. If the team is doing a good job of filling their sprint with PBIs (matching their projected velocity to their target velocity) and the Product Owner asks for a Sprint Grooming session, then some planned work won't get done because the team will spend

time grooming instead. Thus, it's best that the Product Owner be proactive in planning when Sprint Grooming is needed and working with the team during Sprint Planning to make it happen.

Note: It is not the responsibility of the Product Owner to make the PBIs "sprintable". They are focused on capturing the requirements accurately. The acceptance criteria can aid the dev team in understanding the PBI. Having said that, as a scrum team works together more and more, the Product Owner will gain insight into what the team needs in the PBIs to help them better break them down. They will also get a feel for how much work the team can do in a sprint such that the Product Owner can "refine" PBIs on their own that will make them more likely to be sprintable. At the same time, the Dev team will gain a better understanding of the PO's vision and style of writing such that they can dig into PBIs more efficiently and effectively over time. By working together, a motivated scrum team that is interested in success will converge on their understanding of and appreciation for the different roles being played.

Advice for Student Product Owners in SE I and II

Since you are not in a business scenario, the business value you should use to prioritize the PBIs in the Product Backlog is one of learning. Which PBIs will help the team learn the topics the best. Remember the topics are Problem Solving, Source Control (feature branching, versioning, merging, etc), Project Management (planning, scheduling estimation, risk management, change management, defect tracking), SDLC (requirements, design, implementation, test, release, maintenance, upgrade/update), Agile (12 principles), and Scrum. Teamwork, collaboration, brainstorming, and engineering excellence are all inherent in the above topics.

If you can, make it fun. Learning comes first. Fun is an optional non-functional requirement.

Software Engineering

According to Kinser

You can have goals for any sprint be something like, "Gain a better understanding of how to release to the Amazon Cloud" with PBI of "Deploy a simple app to AWS” with an acceptance criteria of “Document in writing and/or video to explain how to deploy to the Amazon Cloud and prototype a release". You could also have a sprint goal that is more feature driven, “Enhance hunt player experience” with a PBI of "Add admin ability to vary how hunt tasks are presented to players" with an acceptance criteria of "Each player in the same hunt has potential to see the hunt tasks in a different order, but all players see the same complete list of hunt tasks." or "The admin can define tasks as a question/answer, multiple choice, single selection, or matching." In either case, select previously groomed stories that support the goals and the ability to achieve the acceptance criteria.

Your goals can be defined in advance and used to prioritize the product backlog so that they are groomed in the order you want to apply the goals in future sprints. However, you should be Agile in your planning, recognizing that the Dev Team may progress at a different pace than you planned (faster or slower) and you need to adjust your plans accordingly. Remember that you are not a project manager.

If you notice that the team is finishing their sprint tasks early, look for simple stories or defects or research items they could pull into the sprint. The Product Owner does this to be proactive in case the Dev Team asks for more work. You don’t dictate to them that they do any of these additional items. Also, you should not suggest any items that have not been groomed by the team.

Similarly, if you notice that the team is not going to finish everything they signed up for in sprint planning, help them identify which items are optional or low priority as early as possible so that the highest value items get completed and anything that is not finished is of lower value.

Software Engineering

According to Kinser

You can change or replace any of the user stories for the class project you like but please stay true to the overall vision of the product. You can break stories down into very small vertical slices without the team's involvement so that you bring stories into the grooming session that are much more likely to be sprintable. Teams respond better to being able to complete multiple stories in a single sprint. It's a psychological thing (part of project management).

You aren't grouping the stories into sprints specifically. It's not like a project manager job where you assign things to an iteration. Instead, you are creating a plan of action, order of attack. You go through the backlog and you put the highest priority items at the top. They don't have to be exactly in priority order (just all the high priority have to be above the medium priority which are above the low priority).

The highest priority items need to be detailed out by you and the team during backlog grooming sessions. If you have too many high priority stories, it will be hard to focus. If you have too few they won't have enough work for the next 2 sprints. It's a bit of a guessing game but time will align things. Make sure you have acceptance criteria defined for your high priority PBIs prior to Grooming if at all possible.

PLEASE NOTE: the team will most likely identify some research stories they need to add to the product backlog as well because there are foundational activities that need to take place and the effort for these needs to be accounted for in the velocity.

For example: They might identify the following items

- Define how they plan to demonstration the acceptance criteria have been met for the Sprint Review
- Define the steps needed to deploy the server application and web pages and persistent store (if any) to the target VM

Define the verification steps needed to ensure the deployment was successful
Define the verification steps to needed to ensure the application is running properly
Define scripts to populate the database in production (this can also be used by each developer to populate their local environments during development)
Define scripts to export database contents so it can be pulled back into a test environment for debug purposes

As the PO, you need to open to these needs and work with the team to understand the priority of these. However, if all they do is foundational stuff there will be no features to release at the end of the sprint. If they don't do enough foundational work then you can't release the features.

As the PO, you can manage expectations. Do not bully your team. Do not let your team bully you. Every sprint needs to have some functional improvement and some process improvement.

https://www.youtube.com/watch?v=502ILHjX9EE A video an SE II student found that has been popular
https://medium.com/agileinsider/top-13-tips-to-become-a-great-product-owner-bb366c678a84 (article that also references the above video)
And lastly, https://www.infoq.com/articles/great-scrum-team/ .

Chapter 16 - Scrum Master

The Scrum Master is responsible for coaching the rest of the Scrum Team (the Product Owner and the Dev Team) on the Scrum model and the Agile mindset. They do this continuously throughout each sprint through one-on-one and group interactions including facilitation of the Scrum Activities.

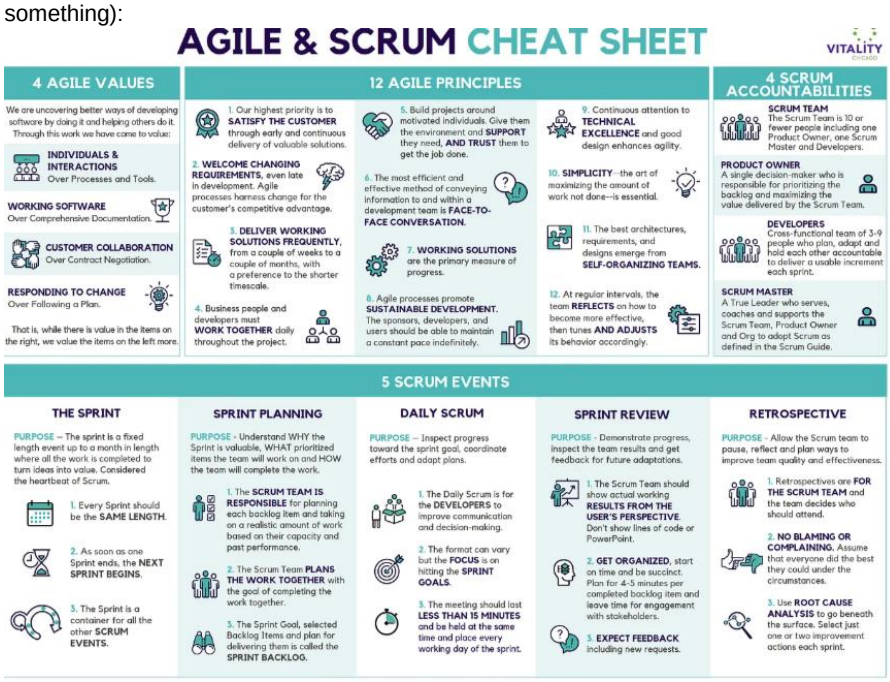
Software Engineering

According to Kinser

During the semester project in SE II, the Scrum Master may feel like they aren't contributing as much as the developers. This should not be the case because there is a lot of things the Scrum Master should be doing.

Most students that take on this role tend to focus on facilitating the meetings by having an agenda and prompting the team members during the meetings. This is a small part of the job. The Scrum Master's goal is to build a Scrum Team that doesn't need them anymore. There is no recipe for doing this because each team is unique and the dynamics of the team can change over time as members evolve or face personal challenges.

First and foremost, the Scrum Master helps the team DO Scrum. This means the Scrum Master must know how Scrum works, the order of the activities, the inputs and the outputs, etc. Beyond the mechanics of Scrum, the Scrum Master must be well versed in the Agile principles. I recommend having a cheat sheet for Scrum and Agile like the one below (Note that Sprint Grooming is missing so add that with a sticky note or something):



https://medium.com/leadership-and-agility/downloadable-agile-principles-scrum-tip-sheet-6710ba0da2c3

The Scrum Master guides the team towards doing these things well. They don't expect or demand the team do them all right initially. Scrum Masters are patient but insistent.

They will allow the team to explore the process and tailor it to some extent because this shows ownership by the team which is a good thing. They will keep nudging the team to do it better each sprint with specific directions for improvements (sometimes those directions are delivered to the team as a group and sometimes to specific individuals in a 1-1 discussion; other times the directions may be given through a series of guided questions).

As the team puts Scrum and Agile into action, the Scrum Master is also looking at the team formation process (Tuckman's model of Forming, Storming, Norming, and Performing). They should know where the team is in this process and be working to help them advance forward. This can only be done by understanding how teams form (e.g. watching the interactions between all the team members and working with each person to improve their relationships with each other to build trust and respect and empathy).

The Scrum Master should be willing to put out some ideas for the team. Some students struggle with this because the team may not like the idea. Very few people like having their ideas shot down or dismissed. The Scrum Master needs to overcome this. For example, I explained in 4250 the importance of avoiding group think so I provided the class with the sticky note brainstorming technique so that everyone had a chance to put their ideas forward without a dominate personality controlling the direction of ideas. The Scrum Master can do things like that. In the Daily Scrum, they could suggest the team use a "talking stick" or pass a rubber duck around. Whoever has the stick or duck should be the only one talking and should have everyone's attention. This visual aid helps the team keep interruptions to a minimum.

For more good advice and guidance on the role of Scrum Master, I recommend this article: https://www.infoq.com/articles/great-scrum-team/ .

Now I will give you some specific advice for what the Scrum Master should do in a sprint.

Prior to Sprint Planning, the Scrum Master should work with the Product Owner to have a sprint goal and a list of already groomed PBIs that support that goal. The Scrum Master can be a sounding board for the Product Owner in this process and ask questions like "how does this goal reflect the business value you want in the sprint" or "what if the team can't do all these PBIs in the same sprint, which ones are the most important to you". The Scrum Master doesn't tell the Product Owner what to have as a goal or which PBIs to pick.

The Scrum Master can publish an agenda for the Sprint Planning meeting. The purpose of an agenda is to remind the team of the meeting, it's purpose, and what they should expect to happen, how much time they have, and what the desired outcome of the meeting is. By publishing this in advance (by a day or two but not much more than that)

the team will be better prepared and the Scrum Master won't have to take as much of the meeting time explaining what needs to happen in the meeting. They should still verbally summarize the agenda at the start of the Sprint Planning meeting.

During the Sprint Planning meeting itself, the Scrum Master will be a silent observer. However, if they see the team rushing through the process and not getting the value they should, the Scrum Master can step in. For example, if the team starts estimating a PBI the Product Owner provided the Scrum Master could ask "Does everyone understand what needs to be done in order to complete this PBI or do you need to talk it out (maybe someone can diagram the detailed design on the whiteboard)" or "Is the PBI really small enough to be a task on the sprint backlog or do you need to break it up into a couple of tasks". When the team does estimations, the Scrum Master should make sure it is done in a way that does not allow one or two people to dominate the decision-making process. Everyone on the team should be able to provide an estimate even if they don't have the skills to do the work themselves. I recommend the team have cards with 1, 2, 3, 4, and ? for the amount of hours it will take to do the work. The ? can be used when someone doesn't have the skills or they feel the task is too big to be done in 4 hours or less. The team can use this approach or something different as long as it allows for everyone to participate in the estimation process. When the estimates presented vary wildly, the Scrum Master can ask the highest and lowest estimates to explain why the voted the way they did (not in an accusatory way but simply to gain an understanding of the thought process behind the estimate because the team members with outlier estimates may have important information the rest of the team doesn't have).

Also, during Sprint Planning, the Scrum Master should keep an eye on the clock. This is a timeboxed activity. Halfway through the time allotment, the team should be roughly halfway done with filling their sprint backlog with estimated tasks (halfway is defined as the total of the estimates is half of the projected velocity of the team for the sprint). This alludes to a key process point: the team should be evaluating each PBI in turn. They should not be defining all the tasks for all the PBIs and then doing estimations. This way, if they run out of time there will still be tasks defined on the sprint backlog.

It may be helpful to the team for the Scrum Master to remind them of their Definition of Done during Sprint Planning so that their estimates include all the work needed not just the coding part. Did they include all the testing in the estimate that they agreed to do in the Definition of Done? All the documentation? Did they include the merge effort and allow for things to go wrong?

At the end of Sprint Planning, the team should a sprint backlog with enough tasks to match their target velocity. It rarely happens in 4350. Oftentimes the team feels it necessary to extend their Sprint Planning beyond the timeboxed window. The Scrum Master shouldn't fight them on this initially, but they should help the team understand the

ramifications of doing this. The more time they spend in planning the less time they will have to do the work they are planning. Also, if it took too long to define the tasks it may be because they didn't break the work down enough during grooming or thought it through enough during planning which means it will likely take longer to do the work than they estimated because they are missing some steps. If this happens, the Scrum Master can remind the team of this scenario at the start of the retrospective so the team can brainstorm on some ways to avoid it from happening again. If it still happens again but it isn't as bad as the last time, this means the team is getting better so allow them time to further improve before pressing the issue further.

The Scrum Master might consider sending out an agenda for the Daily Scrum for the first several of these events to help the team come prepared. As they do the Daily Scrum more and more, the agenda may not be as helpful so it can be discontinued then. The Daily Scrum should go quickly but also be effective. This is a chance for everyone to share what they have done, what obstacles they face, and their next steps. There is no need to go into detail of how they are implementing something. But there is a need for the team to know what each other is doing. This activity is more effective in a workplace environment because it will happen daily and people will be working full time on the project. For the semester project, it will feel less effective. This is okay. Getting used to the process has value. The Scrum Master can help address these points with the team as they come up.

Each team should dedicate some time to doing Sprint Grooming in each sprint. There is a low probability that the students at this stage will be doing the grooming so well that there isn't a need for Sprint Grooming each sprint. The Scrum Master should try to help the team groom each PBI well. This may take a couple of sprints because the team may need to see the results of grooming in their Sprint Planning before they realize what is actually needed in grooming. For example, some teams will groom a PBI by reading it out loud and then asking if it is "Sprintable" or not, then moving on to the next one. What they should be doing is reading the PBI, discussing it, asking questions to better understand it, maybe do some whiteboarding to outline what is being asked for, talking through some high-level design such as an interaction diagram, or listing the classes that will need to be touched, etc. Without some discussion and team wide understanding of WHAT is being asked for and HOW it will be implemented, there is no way to realistically determine if it is sprintable. All of this information needs to be captured in some fashion and linked to the PBI so that when it is brought into Sprint Planning the team will remember what they agreed to and can detail out the tasks more quickly. The Scrum Master can help the team do this by asking questions like "The team agrees this PBI is sprintable, but does everyone feel you could pull this into Sprint Planning and detail out the tasks or are there some unknowns that need to be addressed first" or "How many people on the team feel they could implement this PBI on their own (if only one or two people raise their hands, ask them to describe how they would do it at a high level

so the whole team has an understanding)". The Scrum Master can use Sprint Grooming as a good place to remind the team that "Being Agile" isn't about getting as many things done as quickly as possible. It is a communist lovefest. It is about the team. As you build the solution, you should also be building the team into a self-organized, cross-functional, self-motivated, and highly effective team.

The Sprint Review is similar to Sprint Planning. The Scrum Master can work with the Dev Team in advance to help them be ready. All the working features (tasks that meet the definition of done) should be merged into a single development branch and be demonstrable. The Scrum Master may elect to publish an agenda to help every be prepared. The Sprint Review should start with the Product Owner reminding everyone of the Sprint Goal and the PBIs the team brought into the sprint (not the full list the Product Owner presented, just the PBIs the team committed to doing that sprint). Then someone runs the solution, showing that the acceptance criteria for the PBIs completed has been met. The Sprint Review is not a code walkthrough event. You don't show the code or the database schema or talk about the details of the implementation. You don't talk about what didn't get done or what went wrong during the sprint. The Scrum Team should be performing a validation activity akin to customer acceptance testing to show that the solution has met the requirements embodied in the PBIs committed to. The Sprint Review should end with some feedback from attendees that will help the Product Owner improve the Product Backlog.

After the Sprint Review, the Product Owner should decide if the Potentially Shippable Product should be shipped or not. This decision should be based on how well the acceptance criteria has been met. The Product Owner should not ship the product if it is "good enough" or "close enough".

The last activity in the sprint is the Sprint Retrospective. The Scrum Master will treat this much like the Daily Scrum. They may elect to publish an agenda for this activity for a while just to help everyone be more productive. They should remind everyone this is for the Dev Team primarily. The Scrum Master facilitates this meeting to ensure each member is able to participate equally and no single voice dominates. The goal of the retrospective is for the team to brainstorm on ways to improve their processes, the quality of their outputs (code, documentation, etc.), and the way the team works together. The Scrum Master might consider reminding the team of the list of items from the last retrospective, especially the changes the team elected to do in the most recent sprint. As mentioned earlier, the Scrum Master may remind them of how Sprint Planning went (it's interesting how quickly people forget things from just a week ago). This can be done easily and quickly by simply saying "As you start to thing of the things that went well and the things that could go better, think back to how Sprint Planning went in your view, how Sprint Grooming went, how the Sprint Review went, how the tasks flowed

across the sprint backlog during the sprint, etc.". If the team is using a burndown chart, the Scrum Master can remind them of the final outcome of that as well.

I do recommend that the team use the sticky note process for brainstorming in the retrospective. Each developer writes down up to but no more than 3 things that went well during this last sprint. They also write down up to but no more than 3 things that could have gone better. By having everyone write these things down first before any discussion happens, the team will get a better collection of inputs. Once everyone is done writing these down, the notes should be put on the wall or whiteboard in two columns (one for things that went well and one for things that could go better). Group duplicates or closely related notes. Then have each developer vote for which item in each column they think the team should focus on going forward. Some teams give each person a single vote for each column. Some teams give each person three votes for each column. The Scrum Master can try different strategies. The goal is to identify the items that will help the team the most if they can focus on them. After the vote is done, the team needs to decide what they are going to do differently in support of the items that got the most votes. This could be changes to the Definition of Done, or in the way grooming is done, or how tasks are defined in sprint planning, or the columns in the sprint board, etc.

When the team is actually doing development, the Scrum Master should be observing the team. Is everyone working on a task? If not, why not. Is there an obstacle that the Scrum Master can assist in resolving for them. If someone looks frustrated, the Scrum Master can proactively seek to help them. The Scrum Master can also be reviewing their notes from the Sprint Planning or other activities to see how they can nudge the team in right direction when those activities happen again. The Scrum Master could be collaborating with other Scrum Masters (and encouraging the Product Owner and Developers to do the same type of collaboration).

At the end of the sprint, the sprint backlog should be empty. The team should not carry items/tasks over to the next sprint. Every sprint should start with a new sprint backlog and a new sprint board.

So, if you are the Scrum Master and you don't feel that you contribute as much as a developer, I hope that the six pages I wrote on the role helps you see that there is a lot to do.

Chapter 17 - Dev Team

In the previous two chapters I provided instructions for how to perform the roles of Product Owner and Scrum Master more robustly. I will do the same in this chapter for the role of Dev Team. The Dev Team is a subset of the Scrum Team and consists of the

people that “develop” the solution. This does not mean everyone on the Dev Team is a coder because it takes more than just code to make a solution work. So, I will be talking about how the Dev Team does Scrum in an Agile way, and not talk about how to be a better coder.

The success of a Scrum Team is heavily dependent on the Dev Team. They need to buy into the Agile Principles and the Scrum methodology. A single naysayer can severely hinder the ability of the Scrum Team to realize the benefits promised by Scrum. If enough people on the Dev Team don’t drink the Kool-Aid then chaos will likely reign. So, decide up front whether you are interested in success or not. If not, then come talk to me and I will find a way for you to complete the course without being on a team that wants to succeed.

The Dev Team needs to be committed to doing Software Engineering, not just coding. Look back at the 12 Agile Principles and find the principles that talk about the attributes that describe the Dev Team or the members of the Dev Team. You should find quite a few. These principles help describe what it is to do Software Engineering. Combined with Scrum, which helps describe how to do Software Engineering, the Dev Team is well equipped to put out a quality product.

As you approach the semester project, keep in mind that the goal of the project is to provide you with the opportunity to put into practice all the things we’ve discussed or will discuss in the course. It is not about the features or how many features you will implement. It’s about how you will implement them. You should strive to implement them in a robust way. What does that look like?

It starts with Initial Grooming. The Dev Team should be working together to understand the big picture of the solution that will be built in the coming sprints by asking the Product Owner clarifying questions as they step the team through the Product Backlog. The Dev Team should not argue with the PO, but they can offer suggestions or alternatives. The Dev Team must respect the PO’s decisions though. If the Product Owner doesn’t take the Dev Team’s suggestion, the Dev Team can’t pout and do what they want anyway. They need to follow the PO’s directions for what a features to build and when to the best of their ability.

Initial Grooming is more than just the product backlog though. The Dev Team needs to be deciding as a team what their development processes are going to be. How will they do source control (See the chapter on Source Control in the 4350 and 4250 sections of this book for my recommendations)? What is their Definition of Done? Remember that the definition should be attainable; it’s good to have lofty goals but have a plan that gets you to that goal versus setting the goal for the first sprint knowing you will fail to achieve it. The Definition of Done represents the team’s commitment to quality. Are you going to

do code review? How many? What does a code review entail? Are you going to do testing? What kind? Just unit testing or will you do regression testing or other kinds of testing (remember from 4250 that there are LOTS of kinds of tests)? Will you automate the reviews as part of the source control process? Will you automate your testing, your deployment, your processes, etc.? One team said they were going to automate all their tests. I suggested that they automate one test first so they could get an idea of the effort involved and work out some of the kinks before committing to doing all of them. This is an example of having a plan that works towards a lofty goal.
Initial Grooming is a good time to decide as a team how you plan to groom the PBIs, how you will break tasks down during Sprint Planning, what the columns will be on your Kanban board (the Sprint Board in Trello or Jira), which tools you will use (Trello or Jira, AWS or Azure, Docker or etc. etc.

As you decide these things keep in mind that your decisions are not final. You just need something decided so the team knows how to get started. You have the chance to evaluate and change these decisions every sprint (but only change things on sprint boundaries not during a sprint if you can avoid it).

As a Dev Team, you should be striving to make the Scrum Master role unnecessary. My point being, the Dev Team should work together to improve their ability to BE Agile and DO scrum well. Don't just wait for the Scrum Master to tell you what to do. One of the best ways to be really good at Agile and Scrum is to seek to understand WHY they are the way they are. Why does Scrum timebox the sprint and most of the meetings? Why does the Daily Scrum consist of those questions and not others? Why is grooming separate from planning?

Going into Sprint Planning, the Dev Team should have an agreement with themselves on how they will break down PBIs into tasks that can be put on a sprint backlog. What information does the Dev Team need in order to do that and did they produce that during grooming? Will the Dev Team capture any design decisions or drawings they've done as part of the grooming or planning process? Where will this be stored and how will it be associated with the PBI so that it can be found when needed?

The Dev Team needs to be conscious of the time available to do sprint planning. The more time you spend doing the planning, the less time you have to actually do the work you are planning. The better you plan the work, the easier the work will be to do. Finding the balance is magical but possible. Try to avoid distractions and unnecessary conversations during Sprint Planning. Treat it like a job. Be professional. Seek a positive outcome. Don't settle for low quality efforts.

My recommendation is that the Dev Team break down each PBI in priority order, one at a time. Take the highest priority PBI and talk through the design. This could be one

person leading the group or a subset of the team. Whoever is familiar with the feature and area of the product impacted can lead the discussion. It should be a discussion, not a lecture. There are two goals of this discussion. The first is to find holes or errors in the plan so the correct tasks are defined to be worked on and you end up with a better-quality product. The second goal is educate the whole team on what needs to be done to implement the feature even if they won't be doing the work themselves (it is a form of cross training). This second goal is important in a workplace because you are trying to build a highly functioning team (one that is at the performing level of Tuckman's model). Also, if the people that end up doing the work hit an obstacle, the rest of the team has some idea of what they are trying to do and may have ideas of how to help. The Dev Team should effectively be doing the detailed design work for the PBI in order to define the tasks for the sprint backlog. Try using the whiteboard or sharing a document or other means of capturing the discussion so that it isn't left to each person's ability to remember the details later on.

After the team figures out all the tasks, then they need to estimate the effort for each task. This should be done before moving on to the next PBI. The estimate should be arrived at in a way that avoid group think. Even though one or two people have the clearest idea of how to implement it, everyone should participate in estimating the effort. I recommend using the planning poker approach (where everyone has 5 cards labeled 1, 2, 3, 4, and ? to represent how many hours it will take to do the task). Each person selects their estimate and then everyone reveals their vote at the same time so that no one person influences the estimate. Total up the estimates to see how close you are to the team's target velocity. If you are close (+/- 10%) you are done with Sprint Planning. If the next PBI is too big for the remaining amount of velocity, the Dev Team works with the Product Owner to find something that will fit. This could be one or more bug fixes or research items from the Product Backlog. Whatever is selected should have been groomed by the team already and (if possible) should support the sprint goal.

At the end of the Sprint Planning activity, the Dev Team should have all the tasks in their sprint backlog and marked as "to do". If you list the tasks in the sprint backlog in priority order (based on the priority of the PBI they correspond to) then the Dev Team can work them in that order. If they can't get every task done, it will be the lower priority tasks that don't get done.

The next thing the team does is a Daily Scrum so that each person can put their name next to the task they will work on and move it to the "in Progress" column (or equivalent) of the sprint board. It is important (REALLY IMPORTANT) for each person to put their name next to the task they are working on. If you don't put your name on it and/or don't move it out of the "to do" column, there is a high probability that at some point someone else will start working on it and it will be a waste of that person's time and negatively impact the velocity and the morale of the team. Also, by having the person's name next

to the tasks, anyone with a question or comment about that task knows who to go to. Everyone on the Dev Team should have their name next to a task, and just ONE task. Do not sign up for multiple tasks. This is called hoarding and doesn't end well for the team. If someone on the team cannot do any of the tasks by themselves due to a lack of knowledge related to the PBIs, they should pair up with someone so they can start to gain that knowledge. You don't have to be an expert in something in order to be the Devil's Advocate (remember that phrase from SE I?).

If there are tasks that are closely related and someone or a pair of developers feel it will work best if they do all those tasks, then my suggestion is for them to put their name on the related tasks but leave all but one in the "to do" column. This can serve to tell the team that you plan to do those tasks but haven't started on them. If they want to pick up one of those tasks, they should coordinate and collaborate with you. Or, they could pick some other task. The team should agree that any task in the "to do" column is fair game for anyone to pick up whether it already has someone's name on it or not. There should be no hoarding of tasks.

If the Dev Team uses the sprint board effectively (putting their names on tasks and moving them to the correct columns at the appropriate times) it will help avoid a lot of confusion and problems that arise in team-based development. It doesn't take more than a few seconds to update the sprint board and it helps keep the whole Scrum Team informed of what is happening. A lot of tools can also leverage the sprint board to automatically build the burndown chart. Some tools can use the task id to tag the feature branch in your source control so that when you create the feature branch it automatically advances your task to the "in Progress" column and when you commit your code the task is automatically advanced to the "done" or "committed" column for you. It takes some time to set this up, but it can pay dividends every sprint. This is part of what Continuous Integration is made of (the CI in CI/CD).

Speaking of automation, you can also set up the "actions" in GitHub to include the code reviews as a gate to completing a commit so that an alert goes out to the team that a review is needed; when it is done you get an alert to look at the feedback and make any changes needed; and then the commit can advance. You can also include any automated regression tests as part of the commit process such that the commit will halt if the regression test fails. This will help keep you from committing code that breaks already working features unknowingly. If you go crazy with automation (drinking all the CI/CD Kool-Aid), a successful commit can cause the development branch to be deployed into a testing environment and a series of automated tests could then be run. If all those pass, a new automated workflow will deploy to a staging environment for ilities testing, and ultimately could deploy to production. You might want a few manual override steps in there (like having the Product Owner sign off on the changes before going to production). But you could automate things such that every successful commit ends up

in production without much manual effort. This is the CD of CI/CD (Continuous Deployment). This might be a good PBI for someone that isn't a coder to work on. The students with a concentration in IS are all about building workflows like the one described above.

Both SE I and SE II are set up to give students time in class to work together on their projects. The time is class is best utilized for the Scrum activities and any collaboration that involves the whole team. This way the team doesn't need to find a window of time outside of class when everyone is available. This does NOT mean that you don't need to work on the project outside of class. It just means that the time you spend on the project outside of class can be spent on doing the implementation type work that is needed. The only coordination you need to do with others is if you are pair programming with someone. Just like everyone has different learning styles, as we discussed in SE I, most people have different working styles. Some people work better alone in a quiet place. Others do better with lots of noise in the background (like music or people). Other people do better work when collaborating with someone else. I've tried to set up the projects to support as many of these working styles as possible.

As the Dev Team works on implementation of tasks, don't be afraid to collaborate with other teams in your section or the other section. You can share ideas and designs and code. This is not uncommon in the workplace.

The Dev Team oversees the sprint backlog and the sprint board. It is not the Scrum Master's job to keep these updated. Part of managing them is to work tasks in an order that will result in a potentially shippable product even if every task doesn't get done. It is very common (in this class as well as in the "real world") for some tasks each sprint to not get completed. To ensure you have a potentially shippable product, I recommend creating tasks in vertical slices so that each task or each group of related tasks composes a working capability. You should be starting with a shipped product so only you can make it not be potentially shippable. Another important step is to test your changes. Test the happy path. More importantly, test all the unhappy paths. It is the unhappy paths that will usually be the thing that breaks a product and makes it unshippable.

This leads us to the Sprint Review. In the real world, the Sprint Review is often a "dog and pony" show for stakeholders. A dog and pony show is a demonstration of best of breed. That is to say, the Sprint Review is when the Scrum Team shows off their progress to the stakeholders. It normally is lead by the Product Owner and facilitated by the Scrum Master with the Dev Team in a supporting role. In our situation, your Sprint Review will be when the Dev Team demonstrates their completed tasks to the Product Owner so that the Product Owner can confirm that the acceptance criteria has been met and decide if you should deploy the solution or not. You should not talk about tasks that

didn't get completed. You should not go into details of how things were implemented. You shouldn't be displaying code (it is not a code review or a code walkthrough). It's possible that the Product Owner reviewed each change as it was made and already knows if the acceptance criteria was met or not, but they won't likely know that it all works together from the same baseline until the Sprint Review. I do recommend that you run the solution in a test environment (not on an individual's personal laptop) if possible. If you can setup a staging environment in AWS or Azure that looks just like your production environment, that would be a great place to demo the Sprint Review from.

After the Sprint Review is over, the Dev Team should deploy to production if the Product Owner approved the release. In a sophisticated work environment, the Product Owner should be able to pick and choose which PBIs that were implemented should be released and which should not. I don't expect that from students. Just know that it is possible to be that granular. The Dev Team should automate the deployment process as much as possible. I recommend doing so incrementally. The DevOps movement would have measurements in place to time the deployment process and assess the quality of the deployment. A mature Dev Team would have the ability to rollback a deployment if necessary and restore the previous version without negatively impacting persisted data.

What does it mean to deploy? Most students have deployed an application at some point or installed one on their laptop (like Visual Studio). When you set up your production environment you will do a "greenfield" deployment. This means you will put the solution into the environment for the first time. You may have to create certain directories, files, databases, etc. After that first deployment, your next deployment is really an upgrade. You shouldn't have to delete everything and reinstall it all over again as a "greenfield" deployment. You should only install the pieces that change. One really important thing to do is not destroy or overwrite production data. There is almost always some dynamic data that is created or updates as the solution is used. There are also often log files that the solution generates as the solution is used. These things SHOULD NOT be erased or modified with each new release. Thus, the deployment process changes after the first release.

The Sprint Retrospective belongs to the Dev Team. This is an opportunity to assess the Dev Team's processes and make improvements. Because you do this every sprint, the changes you make don't need to be dramatic. They should be incremental. As talked about in the Third Way of DevOps, it is a good time to try new ideas and experiment. At the end of the retrospective, the Dev Team should have decided on actions they will take next sprint to improve the quality of the product or the quality of their processes or the quality of their teamwork.

Software Engineering

According to Kinser

Chapter 18 -

Commented [WK1]: Pick up here

Chapter 19 - Grooming (Initial and Sprint)

In 4250 we talked about two different but related Scrum activities; Initial Grooming and Sprint Grooming. In this chapter we will dive into each more robustly. Initial Grooming is not strictly speaking part of Scrum. Sprint Grooming is part of Scrum. Initial Grooming happens once before your first sprint can begin. Sprint Grooming only "needs" to be done when there aren't enough groomed items in the product backlog. I like what Essential Scrum by Pearson promotes which is some grooming upfront (what I call initial grooming) and then "ongoing as necessary" (which I call sprint grooming).

In both cases, it is a forward-looking exercise meant to prepare materials for the team. In both cases, there is an element of high-level design occurring as well as effort estimation (at a high level). They are both a group activity involving the entire Scrum Team (PO, SM, and Dev Team). They both require a product backlog to be populated in order to be possible.

Neither Initial Grooming nor Sprint Grooming involve requirements gathering or definition. That is an activity that must precede either grooming efforts. Ideally the backlog is not only populated but has already been refined to some level such that the items in it are in some type of priority order. This is where Epics and Themes can help to organize related PBIs (Product Backlog Items) and these can be prioritized by group if not individually.

Initial Grooming

Imagine that your team was just formed, having never worked together as a group and collectively you are assigned to work on a project that you know nearly nothing about. Sound familiar?

How do you gain enough understanding that you not only feel confident that you can do the work but are actually motivated to make it happen (Agile Principle #5). Seeing that the project has a purpose and that there is a plan/vision for what it is going to be when finished is helpful in building a team and starting with good morale.

A big chunk of this responsibility falls on the Product Owner at the outset of a project. The Product Owner needs to have spent enough time with the product backlog to be

able to speak to it authoritatively. They need to be excited about the potential of the project. They need to have “buy-in” that this is a worthwhile endeavor. This enthusiasm by the Product Owner can be contagious. The opposite is also true. If the Product Owner is low energy, has no excitement about the project, then it will negatively affect the team.

The initial grooming activity is only needed if the team is not familiar with the product backlog or project. Since the first Scrum activity in a sprint is sprint planning, the team needs enough “sprintable” PBIs to fill that first sprint’s velocity. I recommend that you have at least two sprint’s worth of PBIs groomed prior to starting the first sprint.

So, if the team already existed as a team, is already familiar with the product concept/vision and the product backlog has enough groomed and sprintable PBIs for at least two sprints, then the Initial Grooming activity is not needed. If any of these conditions are not met, then Initial Grooming is probably a good idea.

If all of them are not met (the team just formed, the product is new to them, and there are no sprintable PBIs) then Initial Grooming is absolutely needed. Why? One of the key ideas behind Scrum is that the team can maintain a sustainable pace (Principle #8) and that they are motivated (Principle #5). It is hard to achieve these two things without some type of onboarding activity that allows the team to get to know each other in a casual and low-stress environment and for them to spend as much time as necessary to become familiar with the project goals, purpose, value, and vision. It is critical to the team’s ability to enter Tuckman’s Forming stage without going directly into the Storming phase.

During the Initial Grooming, the scrum team will become familiar with the full product backlog together so that they have an idea of the scope of the project. They will learn about each other as they ask questions and propose ideas. The output of the activity is a groomed product backlog with enough sprintable PBIs for at least two sprints but no more than four sprints.

Even if the team has worked together, they must establish a new velocity for their sprints. It won’t be the same as a previous project because the solution is different. So, while the dev team can define a target velocity for their sprints, they won’t have a consistent or accurate feel for what is truly sprintable until they have executed several sprints. Convergence of actual velocity to targeted velocity can take half a dozen or more sprints to achieve even with an established team. Thus, by having at least two sprints worth of sprintable PBIs, even if they overestimate there is probably enough work to fill that first sprint. There is no need to groom more than four sprints-worth because a lot of things can change in that amount of time in terms of priority. At least 3 shippable releases will have been achieved and customer feedback could alter priorities and/or PBI definitions.

So, when the Product Owner starts the initial grooming session, they really need to focus not only on communicating the vision and plan for the project but also the morale of the team. Do they really need to read every single story in the backlog out loud to the dev team? No. How many stories do they need to read? As many as it takes to give the team a pretty solid understanding of the direction of the project. Knowing what is going to come into play later on can help the dev team prepare the architecture for those stories (or, at least, not make it harder to implement later on).
The danger of talking about far in the future features and capabilities is that some of the developers may want to work on those stories more than current stories. But, the dev team should be disciplined enough to stay focused on the stories at the top of the backlog. The stories that the Product Owner has as a priority.

Because you want to start the team off on the right foot, it is best not to timebox the initial grooming session. Some of the reasons for this is:

1) it is the first time the team is looking at the backlog and they probably have LOTS of questions (or should)
2) if you timebox it too short individual team members will make lots of assumptions
3) unanswered questions and assumptions create risk
4) it's part of team formation.

That being said, you can't take forever doing it. Get an overview of the product's grand vision by walking through the full backlog once and then focus on the highest priority stories that the team will tackle in the first four sprints (at most). The overview is the part that really makes the Initial Grooming different from Sprint Grooming. Understanding how the near term PBIs fit into the big picture is essential to empowering the team to create the best architectures and designs (Principle #11).

When grooming a PBI to determine if it is sprintable or not, the dev team must perform some amount of high level design. How formal the design is will be up to the dev team. It could be whiteboard diagrams, UML diagrams in a tool, or a formal document that is updated. Going through the design process is part of technical excellence (Principle #9). In order to estimate the effort to build the PBI, the team has to have some amount of agreement on what the scope of the work is. This is what the high level design helps achieve. If at all possible, the dev team should somehow capture their design(s) and associate them to the relevant PBI because it may be a couple of sprints before they actually work on it and it is a good idea to not lose the work already performed.

If the PBI is determined to be too big for a single sprint, the scrum team works together to break the PBI down into derived requirements or sub-stories. Each sub-story is a new PBI on the product backlog and must be groomed to ensure it is sprintable as well. Ideally, these sub-stories will be vertical slices of the original so that each can be delivered in separate sprints and add value. When all the sub-stories are completed then the original PBI can be validated to ensure it is also completed. As the dev team works through the process of grooming the PBIs, the Product Owner should be paying

attention so that they have a better understanding of what the team takes into consideration when deciding if a PBI is sprintable or not. The Product Owner can then use this knowledge to independently refine the product backlog.

There is no rule that says the scrum team has to break each PBI down in the order they appear in the product backlog. For example, if the first PBI is determined to be too big to be sprintable, the Product Owner can ask the dev team to consider the next PBI and the Product Owner can work on breaking down the first PBI later, without the team's involvement. Remember that the backlog is roughly in priority order, not strict priority order. In many cases, the first several PBIs are of equal importance and may be ordered arbitrarily. Or, they may be ordered based on functional dependency between them. In the latter case, it may not be feasible to skip to the next PBI unless the dependency can be adjusted.

One of the biggest challenges a new scrum team has when grooming the backlog is to know when to stop talking about the PBI and move on to the next one. A lot of new teams will blend sprint planning work into the grooming activity by defining individual tasks that need to be performed for that PBI. That is out of scope for grooming and the Scrum Master (SM) should step in to help the team avoid this scenario. The dev team only needs to design enough of the PBI to feel confident that it can be executed in a single sprint. That's it.

Properly groomed means the Product Owner has acceptance criteria defined, the team feels it is doable in a single sprint or less, and key elements of the Words behind the Words are captured like related stories, bounds to scope, etc.

It's hard to simulate real world situations in the classroom. In SE I, I gave everyone the product backlog at the same time so the Product Owner had the same lack of familiarity as the dev team. Not ideal but we were just learning the basics of Scrum. Now, we are ready to take it up a notch. So, in SE II each team builds their product backlog in order to practice requirements writing and analysis. This leaves the Product Owner and Dev Team with the same familiarity with the product backlog. However, the team is newly formed and you haven't groomed your backlog yet, so initial grooming is probably helpful.

Sprint Grooming

The sprint grooming is optional in that you only need to ensure the team feels there are at least enough properly groomed stories for at least two sprints and no more than four. Sprint Grooming occurs during a sprint, typically closer to the Sprint Review than it would be to Sprint Planning. The team should know grooming is needed during Sprint Planning so that they allocate time to do the grooming otherwise it will negatively impact their velocity and sprint backlog items may not get completed. The Product Owner should know going into sprint planning if there are enough sprintable, previously groomed, PBIs on the product backlog for at least 2 sprints not including the ones they

are bringing into sprint planning (they know this because they own the product backlog and were present during grooming activities and should be tracking what's been groomed and what hasn't).

If sprint grooming is needed, it should be time-boxed, and a task should be created for the sprint backlog to account for the effort much like a research task. It needs to be time-boxed in order to avoid negatively impacting the team's sustainable pace (i.e. their velocity in that sprint). My recommendation is that teams plan to do sprint grooming every single sprint for the same time-boxed amount of time. This makes a more predictable impact on the team's velocity which is very helpful in the early sprints until velocity convergence can be achieved. After that, they can switch to an on-demand approach if they prefer.

Some scrum camps believe the sprint grooming does not require the entire team. I am not one of those people. I think anytime a product backlog item is being groomed, the entire team should be present. By only having a subset of the team present, you are creating a hierarchy within the team and there shouldn't be any. It's a big communist lovefest. Everyone will benefit from hearing the discussion around each PBI.

Sprint Grooming will do the same High Level Design work described earlier in Initial Grooming. Sprint Grooming is always forward looking. That is to say, it isn't about grooming PBIs for the current sprint. It is about grooming PBIs for future sprints. My recommendations of ensuring you have at least two sprints worth of PBIs groomed and no more than four still holds. It is a moving target because each sprint, the team implements one or more PBIs off that list. So, a team should be grooming every other sprint at a minimum.

Sprint Grooming can include more than just grooming the highest priority PBIs. If the team has a Sprint Grooming session planned but already has four sprints worth of sprintable PBIs, or if they find extra time at the end of a Sprint Grooming session (since they are time-boxed), the scrum team may elect to help the Product Owner with some backlog refinement. They could look through the full product backlog quickly to see if any PBIs have become OBEd (Overcome by Events) and are no longer needed. They might find some PBIs have been partially implemented and their scope has changed. They might identify some PBIs that could easily be completed given the current state of the solution and this may lead the Product Owner to raise them in priority.

I use the word "could" a lot in the above paragraph because those actions are not strictly part of Sprint Grooming, but I have experienced some of my teams getting value out of the exercise even though product backlog refinement is primarily the PO's job. It is a big communist lovefest after all and helping each other is almost always a good thing.

Some of my Scrum teams have made sprint grooming a permanent part of every sprint. This gave them a more consistent velocity because the same amount of time was spent on sprint grooming every sprint. If they had properly groomed enough PBIs for at least 4 sprints, they would help with backlog refinement. Having this consistency also removed the feeling that sprint grooming was a disruption to their flow work.

Software Engineering

According to Kinser

Chapter 20 - Scrum Planning

The title says Scrum Planning, not Sprint Planning. This chapter is about how project management is applied to the Scrum process. How is the project's schedule created, managed, and executed in Scrum. If there is no Project Manager, who manages it? How do you know if things are going well or not?

The above slide is from the optional reading in SE I. The bullet points are SE Model and SE process independent, but they are slanted towards Waterfall-like projects in how they are presented. But let's look at it again now that we have some Scrum Method experience.

Scrum defines the team structure (PO/SM/Dev Team) and bounds the scheduling (via sprints).
The Product Backlog, Sprint Backlog, and Sprint Board are part of the Scrum management tools. Some teams also elect to use a burndown chart, but this isn't strictly required by Scrum. These all support managing the project through documenting a plan and monitoring progress against the plan.

We talked about estimating stories to be Sprintable during the Grooming activity. We talked about estimating tasks to be <= 4 hours during the Sprint Planning activity. We haven't talked about how to estimate the project. When will all the critical features be done?

The last two items in the slide above are often addressed by the project plan/schedule (in Microsoft Project for example) which we did briefly talk about in SE I. In Scrum, this is the combination of the two backlogs.

In Scrum, the Product Backlog's order is kind of like a schedule. However, it is more loosely defined because it is not in strict order of execution. The higher priority PBIs are near the top. The lower priority PBIs are near the bottom. We normally don't groom more

than four sprints worth of PBIs at any given time, so we don't know if the rest of the PBIs in the Product Backlog are sprintable or not. This uncertainty can become a problem in planning when a solution will achieve a certain milestone or capability.

As you experienced in SE I if you used a burndown chart, it is not always obvious during a sprint to know with certainty if everything planned will get done and your sprints were only one week, and your teams were small. Imagine how hard it is if the sprints are two weeks long and there are multiple scrum teams working on different systems of a larger solution.

It's a Balancing Act

All Projects have four major constraints on how a project is managed. You can change the size of one of the constraints and it will affect at least one if not all the others. The four constraints are cost (e.g., people, hardware, tools, facilities), schedule (time and complexity of organization), scope (amount/complexity of features), and quality (validation/verification).

Assume you start with the correct balance of staff and time for the given set of requirements to achieve the level of quality that you want. (BTW, this is nearly impossible but let's just pretend)

If you add staff, it adds cost. But it might result in greater quality or increased scope. It will increase the complexity of the schedule. If added late in the project, it will likely increase the schedule timeline as well.

If you add scope to a project, it will increase cost and schedule. It also increases the amount of verification and validation needed, thus increasing quality.

Software Engineering

According to Kinser

If you try to reduce cost and schedule while keeping the same amount of scope, then quality will suffer.

Balancing these 4 things is primarily what a Project Manager does.

In Waterfall, Iterative and other “Plan-based” models, a project manager uses a project schedule/plan to define the tasks and timelines and staffing assignments for the entire project to its projected conclusion. They then collect status daily to track if everything is going according to plan. They use change management and risk management processes to adjust the constraints in real-time with the project. The larger the scope or the longer the schedule, the harder this is to do. A project manager in Waterfall or Iterative projects can leverage the Agile philosophy to be more effective at the process (this is primarily how I ran projects). However, some contracts are written by non-agile persons which constrains the PM’s ability to be agile (e.g. older Defense Contracts).

In Scrum, there is no Project Manager. The responsibility to balance these constraints falls on the entire Scrum Team. The Product Owner is responsible for Scope of the project (they define the Epics/Themes/Stories and prioritize them). The schedule is defined by the Dev Team during grooming (by breaking stories down to be “sprintable” and during sprint planning by defining the tasks). The Scrum Master manages and facilitates the process. Change management is usually under the purview of the Product Owner by how they arrange the product backlog (anyone can add things to the backlog, but the Product Owner prioritizes each item against the others). Risk Management falls across all 3 roles, by identifying risks, defining mitigation tasks, and planning these into sprints. Labor costs are determined by the number of sprints.

If you add more sprints, you increase cost and schedule. If you add more stories, you increase scope (to maintain scope, the Product Owner adjusts the priority of the stories).

Some teams use Story Points assigned to the Product Backlog items to define the longer-term project schedule and its cost. Read up on Story Point and see if you believe this works. (On ETSU’s OReilly’s account you can find Rubin, K. S. 2013. Essential Scrum: A Practical Guide to the Most Popular Agile Process. Upper Saddle River, NJ: Pearson Education, Inc. (see chapters 6 & 7).

Common Fallacies

Adding more staff doesn’t make things go faster. In fact, it almost always slows things down. When companies add more staff to solve a problem, they usually do it at the last minute which is absolutely the worst time. When you add staff, the existing staff must bring them up to speed. This slows down the work the existing staff were already doing. It doesn’t start helping with the workload until the new staff are up to speed and productive on their own. The more complex the problem, the longer this takes.

Working more hours doesn’t make things go faster. It might help in short bursts. If it is expected for weeks on end, it can cause fatigue and attrition. When companies force

staff to work extra hours (nights, weekends, holidays, etc.), it is almost always done when the project is already lost. They also rarely adjust the scope. They also fail to consider the impact on quality. As staff begin to burn out, the quality of their work goes down. They are less motivated. The volume of work may also go down even with the extra hours. Consider a sprinter that suddenly has to run a marathon. They won't maintain the sprinting speed for the duration of the marathon. They might not even make it to the end. The impact of attrition to a project is significant. The investment in time to get the lost staff up to speed is lost. The time in between when they leave, and their replacement arrives, is completely lost productivity (not counting the ramp up time of the new person and the negative impact that has on existing staff). If there is enough attrition, morale of the whole team goes down and so does productivity from each member.

Spending lots of time upfront on the infrastructure and architecture of a solution (especially a large one) often delays the building of the key elements that are of business value. Subsequently, when those key elements are built, changes are made to the requirements around them (which is inevitable), then the architecture must be changed too. The solution rarely needs to scale thousands of concurrent users on day one of the release. The reality is that architecture can be made to be flexible and extensible. Later in the semester we will talk about Emergent Architecture.

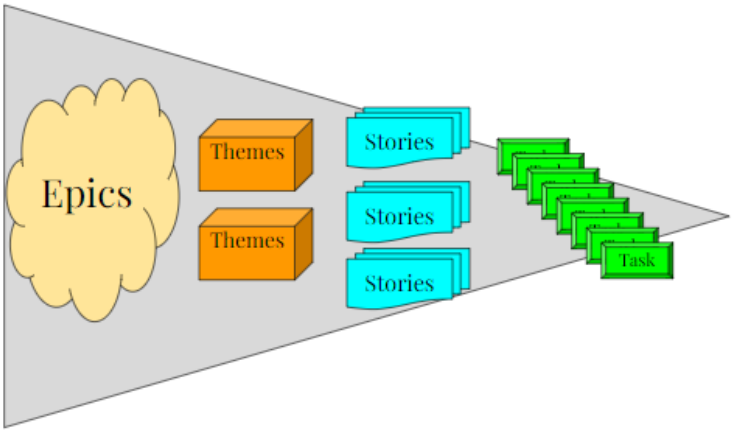
These three fallacies are part of what drove the creation of Agile.

The Cone of Uncertainty

The Cone of Uncertainty is like Tuckman's model. It only defines what is, it doesn't do anything to fix it. At the beginning of a project, there are a lot of unknowns and risks which create uncertainty regarding the success of the project (whether the full scope can be achieved by the target date with the given staff to the desired level of quality). Basically, the further away the deadline, the less certain you can be that you will finish in time. The closer you get to the deadline, the more certain you can be that you will meet it. You only know how long something takes after you finish it.

Software Engineering

According to Kinser



Another way to look at it is, the larger and more vague the problem, the more uncertainty there is. If you can break the problem down into small, more solvable problems, and break these down into tasks, the more certainty there is in achieving success. Epics are vague and big, and full of uncertainty. Themes are more defined but still too big to estimate with accuracy. Stories are better defined and small enough to be sprintable (which is a level of accuracy). Tasks are the most detailed and specific enough and can be estimated pretty accurately down to how many hours it will take.

https://en.wikipedia.org/wiki/Cone_of_Uncertainty
https://blog.tsl.io/understanding-the-cone-of-uncertainty-in-agile-scrum

You can tell if a company, as a whole, has adopted the Agile philosophy or not simply by one question: do you estimate the completion date for your Scrum projects before you start the first sprint? If the answer is yes, then they are not fully Agile.

I say this because it is not possible to know the completion date of a Scrum project before you start. Many companies that do, try to use something called Story Points. If a story point is the amount of work one team can do in a single sprint (i.e. their velocity), then you could, in theory, groom the entire Product Backlog and assign each PBI some number of story points. If it is a large problem, and the team thinks it will take 3 sprints to complete then it is assigned 3 story points. Thus, you groom the Product Backlog and add up the story points and it tells you how many sprints it will take to complete the project. Right?

This doesn't work. Not all PBIs are equal is size, definition, accuracy, etc. If a PBI is an Epic, it is too big to estimate without breaking it down into themes and then into stories. If a PBI is a Theme, how many stories it needs to break down into is not known, and what the effort of each story is not known. Some PBIs may be research items, for which there is no understanding of effort until the research is done.

Compound these obvious issues with more subtle ones. A PBI further down the list

might be estimated at X number of story points now but changes in the solution between now and when it is actually worked on can cause that number to increase or decrease. It's entirely possible that it won't be needed anymore at all (e.g., the customer might change their mind, or it might end up being in conflict with something else of greater value). The further down the Product Backlog a PBI is, the more uncertainty there is as to what the state of the solution will be at the time of implementation so it's impossible to estimate what the effort is.

Then there's the math behind how productivity is calculated. If the sprint is 2 weeks long, then each sprint has 80 hours * the number of developers on the Dev Team, right? (WRONG). You have to account for the Scrum activities, other meetings, interruptions, training time, holidays, vacations, sick time, etc. etc. Some of these are known (e.g., Scrum activities are timeboxed and holidays are known in advance) and some are not (e.g., pregnancies are hard to predict more than 9 months in advance, attrition, sick time). Remember how your sprint was affected last semester by university break(s) and missing classmates? Did you accurately predict these?

Most planners use .6 as a general guide for work that takes 6-12 months, so 80 hours becomes 48 hours:

1 Story Point = #dev * 48 in a two-week sprint

Try explaining all this to stakeholders.

This is why, as an Agile project manager on waterfall and iterative projects, I set my max task size to 2 weeks and my minimum task size to 3 days and my target task size to 1 week; assuming a .6 work factor (meaning, my technical leads and I broke our schedule down into tasks whose size was within this range). This did result in a fairly accurate plan, but it wasn't the only factor involved in ensuring success; change management, risk management, proactive problem solving, etc. were also involved.

There is a scenario where the Story Points approach works:

IF the Product Owner has defined all the critical User Stories for a project

AND

IF the Dev Team grooms the entire Product Backlog

AND

IF the Dev Team is consistent in their estimations

AND

IF the Dev Team's velocity is established

Then and ONLY then can a completion date be estimated.

But we know from the history of software engineering and the Agile philosophy that all the requirements cannot be known upfront. We also know that the longer the project

timeline, the less likely the Dev Team will remain static in its composition. We also know from our experience in SE I, that a newly formed team cannot accurately estimate what is sprintable. We also know from our experience in SE I, that it would take a very long time to groom an entire product backlog (remember your product backlog in SE I was only 1 paragraph from a 6-page document).

Agile Principle 5) Build projects around motivated individuals. Give them the environment and support they need and trust them to get the job done.

From a Scrum Perspective

Assuming a solid first pass at a project's requirements were made (potentially using the Sticky Note brainstorming technique). Now, we want to plan the work. We may need to pull related stories together to establish the initial set of releases.

For example, the Product Owner might have marketing and customer related stories identified as Themes in an Epic for a new product rollout. Customer Retention incentives are related to customers and marketing but add no business value until you have an established customer base. Capturing new customers would be a higher priority. Retention is only required if a drop in return-business is identified as a factor in continued revenue growth since capturing new market share is often the main driver at the start of a business project. So, in this case, the Product Owner might pull out the stories related to capturing market share quickly into a new Epic and move it to the highest priority position in the product backlog.

From a planning perspective, Epics are like large scale efforts done on a quarterly, biannual or even annual basis; Themes are collections of related user stories that have the greatest impact when done together and usually take multiple sprints to accomplish (i.e. a month or more). An epic contains multiple themes. A theme contains multiple user stories. User stories (which are ideally "sprintable") contain multiple sprint tasks. Thus, we break a large problem down into smaller pieces until they are solvable.

I like to have a single Epic to keep my highest priority stories in. However, it is also reasonable to pull stories from more than one Epic each sprint. It is up to the PO.

Risk Management is built into Scrum. If a PBI is identified as being risky or "at risk", the Scrum Team can explore options during the Grooming activity. They may want to add some research PBIs to the backlog to research different ideas or technologies or tools, and thereby reduce risk to the original PBI. They may want to break the risky PBI into multiple PBIs and deliver some portion that is less risky or not risky at all, in order to gain market feedback.

- If the story is high risk but low value, then it would be lower in priority in the backlog.
- If it is low risk and low value, it might be far down in the backlog.
- If it is high risk and high value, then you want some research stories towards the top of the backlog that help mitigate the risk before you need to work on

the story itself.

If it is low risk and high value, you might not need to do anything except note the potential risk.

In all the above cases, the level of business value the story drives the time spent on it and it's priority relative to other stories.

Another resource on this topic: https://www.scrum.org/resources/blog/managing-risk-scrum

Change Management is what Scrum is all about. By having short sprints, the team can respond quickly to changing needs. By staying focused on the highest priority items, they don't waste time planning out things that may never be needed. This gives them more bandwidth to respond to change.

The Product Owner should be working on the product backlog daily. Interviewing stakeholders to define new requirements or further refine existing ones. Evaluating the business value of the entire product backlog on a regular basis, causing stories to be shifted in priority order. Taking the results of recent grooming sessions and applying them to other stories in the backlog (Note: as a Product Owner works with a team for a period of time, they will begin to anticipate the information the dev team needs during grooming and begin to provide that information proactively).

Even if the Product Owner has a draft list of sprint goals, the Product Owner should be evaluating the order and even the definition of those goals on a regular basis. Thus, fulfilling the Agile Manifesto statement of valuing "Responding to Change over Following a Plan". It's good to have a plan but it's even better to be flexible with the plan.

Another aspect of Change Management in Scrum is that anyone can add items to the product backlog. (https://www.scrum.org/forum/scrum-forum/5826/who-allowed-add-items-product-backlog). Depending on the tool used to manage the product backlog, this can be done a few different ways. The key is that it is clear the item is new and has not been approved by the Product Owner or properly prioritized. Only the Product Owner should set the story in the proper order. If they didn't add it, the Product Owner should spend time researching it before bringing it to the Dev Team in a grooming session. Now, just because anyone can add an item to the product backlog does NOT mean that anyone can edit or remove items from the backlog. Only the Product Owner should edit, move, remove, or otherwise alter stories on the product backlog.

The Root of Agility is centered on the following:

- Change your mindset from robust set of features to robust solutions
- Work in small batches
- Merge frequently
- Vertical Slices yield valuable software

Instead of trying to add as many features as you can, focus on trying to make the features you do add to be more robust; better engineered. As a team, don't work on something that takes more than a single sprint to implement. Break it down. Don't try to

multi-task. Individually, define and work on tasks that can be completed in less than a day or two days at most. As you work, use feature branching. Pull in changes at the start of your day, after lunch, and at the end of your day. As you finish your task, pull in changes again. Pulling in changes this frequently minimizes the merge conflicts your code will have with others. Only push your changes if you are confident that it meets the definition of done (that is, only submit/commit working code). As you break work down in grooming and in planning, break it up based on vertical slices. This allows you to complete tasks that add value by themselves and even more value when combined with others.

We have already covered the Grooming activity in the previous chapter. Suffice it to say that the Product Owner needs to have a plan coming into the Sprint Grooming session, with a specific list of items that need to be groomed. The team should not be spending time trying to figure out what has been groomed and what needs to be groomed. The Product Owner should also provide an overview and explanation of each PBI to be groomed. This includes explaining to the team how this PBI fits into the vision of the product, adds value, and aligns with or is related to work done thus far. This gives the team context for the grooming activity. The Scrum Team should be applying the Root of Agility during the Grooming activity.

At the start of Sprint Planning the Product Owner should present and explain the sprint goal. This, of course, means they need to have defined the sprint goal prior to the sprint planning activity. Product Owner should present (in priority order) and explain the PBIs they selected to support the sprint goal. This explanation should include the business value of each, how each supports the sprint goal, and the acceptance criteria for each.

The Sprint Goal is NOT a restatement of the user stories selected. This is the most common mistake Product Owners make. They should set the goals in advance and then pick the stories that support those goals. This is how they can determine which stories are highest priority in the Product Backlog. The Sprint Goal is NOT a compound sentence. That would mean there were multiple goals. This causes more confusion than it helps. There should be one goal. The Sprint Goal is not the team's goal. The team's goal is defined by the Agile principles. We want to build quality software that adds business value. This is the team's goal for every sprint.

The Sprint Goal should be a business value driven goal for that sprint/release. This helps the Dev Team make decisions and tradeoffs throughout the sprint (because we know nothing ever goes exactly as planned).

PO presents backlog items that support the sprint goal. Because the Product Owner doesn't know what the team can accomplish in a sprint:

They should present all the PBIs related to the goal

They should have them already selected and ordered based on priority to the business
They should have acceptance criteria already defined for each PBI
They should be able to succinctly explain each PBI as part of larger product story and show how it flows from the last sprint
They should explain the value to the business
They should be open to discussion and compromise to a point

Sprint Planning is not the time to determine if a story is sprintable. This should have been done in a previous grooming session. The Product Owner is responsible for being consistent with their priorities. If a story is a priority, it should have already been groomed by the team to ensure it is ready to be brought into a sprint prior to being presented in sprint planning. (This rule gets broken more often in the workplace than it should)

Most POs will have sprint goals tentatively planned out several sprints in advance (like I did for SE I). Personally, I would have a draft list for all the stories associated with the Epic I am focused on delivering. I would order them based on value to the business and functional dependencies. Since POs don't necessarily know if everything they want can be done, they have to be flexible but consistent. Having a draft plan is good. Being able to adjust that plan depending on what the team can deliver and what feedback is received in sprint reviews, is being Agile.

The Product Owner should respect the team's time by coming into sprint planning with a clear agenda (goal and stories with acceptance criteria already to go). However, they are not allowed to dictate to the team (i.e. they can't say,"this is what has to be done in this sprint so find a way to get it done no matter what"). The Product Owner has to be open to feedback from the dev team. As the dev team breaks the stories down into executable tasks with estimates, the Product Owner must respect their estimates. But, the Product Owner can keep the team focused on the goal (some developers will try to add sprint tasks in that aren't needed for the stories selected or in support of the goal because they feel it is important to do or it's left over or they think it is a great idea; some of these may be valid and the Product Owner might allow it but they don't have to).

Remember, the Product Owner defines the vision and priority of work to be done. The Dev Team determines the amount of work to take into the sprint. The Scrum Master should facilitate the discussion and compromising between the Product Owner and the Dev Team.

If the Product Owner knows they need to do some more grooming in that sprint, they should remind the team to create some sprint backlog tasks for the grooming session.

The Scrum Master should remind the Scrum Team of the outcomes of the most recent Retrospective. The team decided on some things they want to keep doing and some things they want to do better. They also decided on how they will do things differently in order to support those decisions. It's important to remind them of those changes because it likely will affect their estimations for effort this coming sprint.

The Dev Team then breaks the highest priority PBI down into executable tasks. This is effectively, the execution of the detail design work that was discussed in SE I. The detailed design should leverage the Root of Agility, especially "work in small batches" and "vertical slicing". Once they feel they have all the tasks defined for that PBI, they should estimate the effort for each task. This should be totaled and compared to the projected team's velocity for the sprint.

I've known some teams that skipped the estimation portion entirely. They focused, instead, on just defining the order work needed to be done. These were well-established teams that had been working together for years and working on the same product for the same amount of time. Their sprint backlog would include some updates/upgrades to features but were dominated by customer support tickets. Some of these were defects in the product. Some tickets required manual intervention in a customer's data. Some tickets had minor changes in layout. They just worked them in the PO's priority and released every sprint. I tell you this because it is important to know that every sprint team can vary some of the concepts we discuss but the core remains unchanged if you want the benefit of Agile and Scrum. In this case the core was: The Product Owner sets the priority of work; the Planning happens at the start of the sprint; the dev team decides how much they can do; the sprint is timeboxed and ends with a shippable solution.

Since your teams haven't worked together for years or even seen this solution before, you should try to follow the steps outlined above.

Scrum Team discusses the PBIs. Notice that I didn't say Dev Team discusses the PBIs. The goal of the discussion is to fully understand "The words behind the Words". This discussion is often blended with the next step. Hopefully, the Product Owner is able to communicate the Words behind the Words in a concise and timely manner (answering any questions the team has). If the team doesn't ask any questions, the Scrum Master should prompt them some (e.g., "What other information do you need before you define the tasks for this story?", "How many tasks do you think this will take?", etc.).

The Dev Team may not have questions until they start breaking down the story into tasks. As they define tasks (which is effectively performing a detailed design), they may have questions they didn't anticipate before. This is part of the discussion. Which is why I say this step blends with the next one (defining tasks and estimating them). Sometimes the estimate depends on choices the dev team wants the Product Owner to make. For

example, in BucHunt, some teams had tasks associated with managing multiple hunts even though the first epic only had one hunt as its scope. The Product Owner should have pointed this out because Agile principle #10) Simplicity--the art of maximizing the amount of work not done--is essential. Another example, some teams in SE I created tasks for team play. This was also not necessary because a web app doesn't preclude multiple connections using the same access code unless you design it to be that way. Thus, team play (where multiple people use the same access code) doesn't require any coding. Some teams created tasks to generate access codes where they had to also ensure each access code was unique. They could have just use the player's phone number (which was already known).

How do you break a PBI down? Well, during grooming, you had to do high level design to get large stories down into Sprintable stories. Now, you do the detailed design for each story (one at a time in priority order). You only have to detail enough of the design so that you are ready to implement it. Add all these tasks to the sprint backlog.

Quick tangent: The sprint backlog is NOT an extension of the Product Backlog. It is used by the Dev Team to manage their work in a single sprint. The product backlog is managed by the PO. The sprint backlog should be blank at the start of every sprint. There is no carryover from the last sprint. If items were not completed in the last sprint, they should have been added to the product backlog. Also, the last retrospective might have altered the layout of the sprint board so make sure this is done before you start developing.

Total up the estimates for all the tasks in the sprint backlog. If the total is less than the targeted velocity, then keep going. If it is more than or within 2 hours of the targeted velocity, you are done with planning. Planning poker (as a reminder) is where all the developers (not the Product Owner or SM) write down their estimate for that item without showing it to anyone else. Once everyone has estimated, they reveal their estimates. The highest and lowest estimates are explained and then we do it again for the same item until the estimates are consistent or the same.

After several sprints, a team will naturally converge to their sustainable velocity (which really means their productivity levels off as they get into a groove). Sprints are short enough that the team can look at the burn down chart during retrospectives to see how far off they were in their sprint planning. They will know if they were accurate or not at the end of the sprint even without the use of the chart because they will have either had tasks left incomplete (meaning they overestimated their velocity or under estimated the effort for the tasks) or they will have completed everything early (meaning they under estimated their velocity or overestimated the effort for the tasks).

Software Engineering

According to Kinser

Some teams use hours when they estimate, some use T-shirt sizes, some use colors, etc. It doesn't really matter. Let's say you use hours. If the team estimates most sprint tasks at an average of 2 hours each (likely since we have set 4 as a max, above which we break the task down into more tasks), then they work on them. They take into the sprint as many tasks as will add up to their target velocity. If they are overly optimistic with their estimates, they will end up with some tasks not done. If they are overly pessimistic, they will finish all of them earlier. The team will naturally adjust their estimates the next sprint even if the average is still 2 hours for each task because the team will adjust slightly what they feel they can do in those 2 hours. They aren't really adjusting the 2 hours because the total of all the tasks still has to come up to the projected velocity for the sprint. It's psychological or magic or peer pressure…. But it does happen. The longer the team works together, the better they get at estimating what they can do in a single sprint even though the tasks each sprint are different.

Remember the cone of uncertainty. The smaller the task, the more confident you can be in your estimate. By keeping the sprints short, we keep the tasks small, and therefore our confidence in our estimates will grow.

Which is better, to take on more work than the team can do in a sprint in hopes that things go faster than you expect or to take in less than you think you can do in case something goes wrong?
ANSWER: Neither. Take in as much work as can be done based on the estimates. The burndown chart and retrospective are where you learn to adjust your estimation abilities so they eventually will be accurate enough that everything you take in will get done. This takes time and adherence to the process. If you try to adjust on the fly, all your data is skewed, and you are just using your gut feeling instead of empirical evidence.

If a team member doesn't have the skills to work on any of the sprint backlog tasks, DO NOT create tasks for them to work on. This team member should pair up with other developers that do have the skills. This creates a learning experience for both. The first developer gains some new skills. The second developer strengthens their own skills by teaching someone else (Doctors have the famous saying, Watch One, Do One, Teach One).

In SE I, some teams assigned tasks to specific people during planning. This was not correct.
Since the team is breaking down the stories that are in priority order (based on what the Product Owner provided), then the tasks in the sprint backlog are naturally in priority order also (since you should break each story down one at a time). When planning is done, the next activity is to do the daily scrum (the standup). This is when each developer will pick a task off the backlog.

Software Engineering

According to Kinser

Chapter 21 - Daily Scrum, Peer Reviews, Release Management

Daily Scrum

> The purpose of the Daily Scrum is to inspect and synchronize the team's progress towards the Sprint Goal, discuss if anything impedes the team and re-plan the team's work to achieve the Sprint Goal.
> https://www.scrum.org/resources/blog/daily-scrum-tips-tactics

Check out the website in the link. This site is one person's opinion, but it is associated with scrum.org so it's been vetted. There are points in it that I don't agree with but thought it worthwhile for you to consider.

The key idea is that you have the meeting daily (in a traditional M-F work environment) at the same time and in the same place. By being daily, you don't have to go into much detail because probably not much has happened since the last meeting. Also, you can identify and potentially deal with obstacles as soon as possible. Having it in the same place at the same time creates predictability and it is easier for the developers to block off the time and, most importantly, not create an interruption to their "flow" when working on tasks.

- 3 Basic Questions
  - What did I complete since the last daily scrum meeting
  - What obstacles do I have (if any)
  - What do I plan to complete between now and the next daily scrum meeting
- Duration
  - Daily Scrums should only take about 15 minutes max
  - If a team has 9 developers, then that gives each developer 1.5 minutes each
  - That is 30 seconds to answer each of the three questions

While not strictly a status meeting, it is a status meeting. You are communicating to the entire Scrum Team where you are with the item(s) you are working on. That's a status update. This helps everyone on the team understand where the sprint is relative to the original sprint goal.
There should NOT be a lot of discussion around the answers to these questions. They should be made as statements. The team (especially tempting for the Scrum Master and

PO) should not grill the person talking about their progress or implementations or how long things take to do, etc. If anyone has questions, they want answered, they should do it outside this meeting.

Stating what the obstacles a developer is facing is not an invitation to solve the problem right then. It is an invitation to anyone that can help solve it, to contact the developer after the meeting. If there is a discussion to be had it can occur right after the Daily Scrum and this is sometimes referred to as the parking lot or the 16th minute. The key idea here is that only the members of the Scrum Team that are needed for the discussion will remain after the Daily Scrum and it won't take up everyone's time. Some camps say the Scrum Master is responsible for removing obstacles. This is not my position. My position is that the Scrum Master is responsible for ensuring the obstacles are getting worked on and don't continue to be obstacles any longer than necessary. They facilitate the removal of obstacles.

Obstacle is another name for risk. We covered risk mitigation in Project Management and even talked about how it is done using the product backlog and grooming sessions. But risks come in all shapes and sizes. Risks exist within the sprints themselves. For example, several teams in SE I had situations where one or more developers were blocked from progress because of some task that wasn't completed by someone else. This is a risk that was realized in the form of work stoppage. The team should find a way to mitigate this risk. When tasks have a logical or functional dependency on each other, define that dependency during the sprint planning meeting and have a mitigation plan. If the task others are dependent on isn't finished or isn't finished in the estimated time, how can the dependent tasks continue? Well, you can look at the facade pattern or use of stubs or hard coding, etc. Often, the blockage is reflective of a realistic error that would need to be handled in the code anyway. For example, if I'm building a webpage that shows BucHunt players in a hunt but the person doing the database work isn't done, I can't "finish" my work. Or can I? Not being able to connect to the server or connect to the database is a real possibility in the operations so I must have code to handle it. If I do, then I can finish my task. To test the "happy path", I might need to fake some data during testing, but it wouldn't stop me from continuing to develop my code. If there is an API the other person is supposed to write that I must call, I might ask for the signature and just implement it with a no-op thus forcing an error condition that I handle in my code. If they can't give me the signature, I can define my own and send it to them letting them know that if they use a different one, they must update my code too.

Can you think of other examples of risks that can occur in a Sprint?

How about

1) A developer gets sick so their tasks don't get worked on
2) A significant defect is found in production
3) A change made during the sprint breaks everyone's code

What are the mitigations for each? For any examples you have experienced or came up with?

Since the team is self-organizing, any technical obstacles that can be solved by the team (e.g. I need someone to review my code before I can finish my pull request) should be proactively addressed by the team without the SM's involvement. The Scrum Master can step in if the team doesn't step up by asking "Who is going to do the code review?" Thus, the Scrum Master is facilitating the obstacle being removed but not removing it themselves. They do the same thing for organizational obstacles (e.g. "I don't have access to the deployment tools", the Scrum Master may not be able to grant access, but they can chase down whoever in the organization can and get the access granted so the developer doesn't have to spend their time doing that).

The Scrum Master is responsible for the process. So, if a developer is not completing a task they signed up for in a timely manner (e.g. the tasks was estimated at 4 hours and the developer has had it for two days without finishing it), they can ask the developer "what obstacles are in the way of completing the task". It is possible the developer is multi-tasking (which is a no-no in the scrum process) or needs to pair with someone else but is hesitant to ask for help.

All the developers should be taking tasks from the sprint board. The daily scrum is a good time to update the board as they talk. My teams normally used a wall as their sprint board. They would manually move a task from the column it was in over to the new column based on their progress with that task. They would pull a new task from the To-do column when they were ready to take on new work. If they thought of a new task that needed to be done but wasn't identified during sprint planning, they would add it to the board along with an estimate and an explanation of why it was needed. If everyone agreed, it stayed. If not, it was moved to the product backlog.

That link at the start of this chapter is very focused on using the Daily Scrum as a planning meeting or a re-planning meeting to keep the team on track to achieve the sprint goal set by the PO. There is some validity to this approach, but I feel that if the team is following the scrum process, this will take care of itself. I'm more interested in the scrum team achieving a sustainable pace of work that results in potentially shippable solutions each sprint. As the team moves through the Tuckman model, the dev team will become more productive, and the Product Owner will adjust their sprint goals to be SMART (with emphasis on the A-achievable). In the early sprints, I don't focus too hard on making every sprint perfect. I trust that the team will get there.

Peer Reviews

Software Engineering

According to Kinser

Peer Reviews (e.g., peer inspection of documentation, designs, code, and tests) provide constructive feedback in a collaborative, non-threatening manner. Remember to include compliments in addition to critiques. Reviews aren't just about code reviews but I'm going to focus on those in this context since we are talking about the daily scrum.

Code Reviews- this is my preference with some guidelines (for regular workplace situations)

At least 2 reviewers no more than 4
At least 1 reviewer has to be a senior member of the team
At least 1 reviewer has to be a junior member of the team
Each reviewer takes 3 passes through the code

Look for coding standards violations (contrary to the slide deck 09_Reviews_and_Inspections, Agile DOES like standards; Agile does NOT like oppressively complex standards)

Look for good coding practices above and beyond those identified in the standards.

Look for adherence to the design (does it solve the intended problem well and fits with related code/tasks)

Reviewers should not try to rewrite the code in their own image (NIH - Not Invented Here syndrome). This is one of the hardest things for junior developers to do. There are a nearly infinite number of ways to implement a programming task. The reviewer should evaluate the code to ensure it works as written. A side effect of this approach is that the reviewer may learn some new techniques. These are great opportunities to leave a compliment in the feedback.

Reviewers should complete and return their findings within 24 hours of being requested to do the review so that they don't become an obstacle to the author.

Now, do you include the tests when you submit your code for review? I'll leave this up to the teams to decide. I like to have my code reviewed before I build my test drivers but I know that Test Driven Development is popular which means you have them at the same time so you should have both reviewed.

I got the feeling in SE I that a LOT of you were NOT really testing your code. I got this idea because it didn't always work in the sprint review; but also, I can tell just from how you talked about testing in your status reports or during class time. One team even told me they didn't have testing as part of their definition of done. At least they were honest about not testing.

Some people like to do walkthroughs, this is where the developer presents their code to a group of people and “walks them through it”. This is used a lot in Government contracts. It is very heavy and time consuming, so people started just walking through key parts of the code (selected by the developer). Which means all the code isn’t reviewed. If they were devious (or just sensitive to criticism), the developer would only present the parts they thought were best instead of the parts they probably knew they needed feedback on. Usually, the group feedback would be dominated by one or two persons so it wasted everyone else’s time.

Code reviews are NOT a waste of time and don’t take much time. However, there are problems with code reviews mostly stemming from the fact that most developers don’t like to do them.
I very much enjoyed doing them and subsequently was solicited by most of my peers which meant I did a LOT of them over the years. I was solicited because I dropped what I was doing and did the review almost immediately and provided a quick turnaround (“Do unto others as you would have them do unto you”). I provided relevant and meaningful feedback. I didn’t just sign off on it without looking. Why? Because I learned as much by reading other people’s code as I taught them with my feedback. I see them as a Win-Win. You SHOULD DO THE SAME.
Not doing code reviews is a FAILS TO MEET EXPECTATIONS in this class because it is part of the scrum process and part of the course instructions, so it is expected to be done and to be done well.

Release Management

In SE I, I didn’t require teams to do an actual release to a production environment. It is required for SE II. There is a good tutorial here:
https://docs.github.com/en/repositories/releasing-projects-on-github/managing-releases-in-a-repository

In my career, I often had at least one full time person in charge of release management, sometimes more. They support the nightly builds, helping to reconcile merge conflicts and changes to the development, test, demo, and production environments. They often supported a half dozen or more teams of 20+ developers in each team. If they said a release was ready for production, I knew it was true.

The root of agility encourages you to merge frequently so this becomes the responsibility of everyone on the team. Merge conflicts are not anyone’s fault. When they happen, there is no one to blame. They are a natural occurrence when you have several people working on the same product because the pieces of that product that intersect so much

of what happens that when a few people are making different changes at the same time, it is inevitable they will stomp on each other's feet. The best solution to a merge conflict is to talk through the changes among the developers that made the changes that are in conflict (without pointing fingers or hurting anyone's feelings).

The first time you deploy a solution to a production environment it is a greenfield endeavor. That is to say, you get to deploy it without many constraints. The second and subsequent releases need to take into consideration the fact that the previous release is running in production and is getting used. Therefore, if your solution involves client data, sales records, or any kind of data that is changing with each interaction in the production environment, your release process needs to account for this fact. You can't just install the database, or you will wipe out the data collected in production. You must have scripts to port the existing production data to any new format or schema without losing or corrupting the data. Also, if the release fails, you have to be able to roll back those same porting scripts back to the previous release. This can be complicated. For example, in BucHunt players may have made progress on a hunt and then the team releases a new version of BucHunt. We can't lose the player's progress. The new version may include changes to the database schema and how we track progress or how we model players, or how we model hunts, etc. When the new version is released, all previous data (past hunts, current hunts, planned hunts, past players, current players, planned players, players progress, etc.) has to be ported to the new layout.

As discussed in SE I, how and when the release of a new version is done depends on the nature of the solution and how it is used. For example, GoldLink is primarily used during the regular workweek by the ETSU staff and faculty. Students access throughout the calendar week. The best time to do a release of updates would be a "dark release". A dark release is a release that typically happens late at night or early in the morning on a weekend with usage is very low and the impact to users would be minimized. However, this requires the development and operations staff involved to come in "off hours" and there is resistance. Because the main users are students, GoldLink is updated during the work week, usually late in the afternoon. Why? Because students don't have much of a voice and are not really viewed as customers. The staff and faculty start leaving in the afternoon (just watch the staff parking lots to see how full they are at 4pm).

Amazon does releases throughout the day, but they do it in a way that does not halt user interactions. They may slow the system down, but it won't be so noticeable as to cause complaints. How do they do this? They use microservices and separation of responsibility in their architecture which allows them to update individual portions of the solution in isolation. They never have to bring down the whole system in order to push out a new version.

Software Engineering

According to Kinser

In SE II, you will meet expectations if you are able to release to production in several of your sprints. You will exceed expectations if you release to a test environment first and perform automated testing prior to releasing to production. You will exceed expectations if you automate your release process using tools like Docker. You will exceed expectations if you release your main application independently of your microservice(s). You will exceed expectations if your release includes the execution of scripts that ports runtime data to your new schema. In the real world all of these things will be expected and will only get you a “meets expectations” on your annual review.

Chapter 23 - Retrospectives (things to look for)

Based on Agile Principle #12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

- What went well in the Sprint
- What could be improved
- What will we commit to improve in the next Sprint

Nice summary at https://www.scrum.org/resources/what-is-a-sprint-retrospective

As to how to conduct one, it varies by team. I recommend that you try the 3x5 card approach. Each developer (not the Scrum Master or PO) writes down three things that went well in the sprint that they would like to see continue happening. Write down three things that could be or should be improved in the next sprint. The Scrum Master facilitates the creation of these two lists and the discussions around them.
The team should vote on the top one or two items they would like to make sure continues to happen (using the 3x5 cards again). Then the team votes on which one or two items they want to improve on. Try to phrase the selected items in SMART terms. Finally, the dev team needs to define the steps they want to take to achieve all of the selected items going forward. This often results in changes to the definition of done and/or automation of steps in the sprint activities.

Remember to focus on the process, tools, etc. and not slide into a blame game. Scrum requires teamwork. Teamwork is not possible if a member is used as a scapegoat.

Examples of weak retrospective items:

- Get better at estimation
- Get more done
- Improve communication
- Improve teamwork

The above examples are positive things to say but they aren't phrased in SMART terms. They also don't state how the team plans to achieve these items.

Examples of strong retrospective items:

- Each developer needs to put their name next to any sprint task they are working on as soon as they start working on it so that everyone knows that task is taken.
- If someone says they need a code review during the daily standup, at least two other team members will agree to do it after the standup so that the first person is not stalled in their progress.
- Update the definition of done to include reviewing and updating the user guide so that it is always current with the latest changes

Fodder for the Retrospective:

i. If no grooming needed in a sprint
   Cause: The team didn't complete enough tasks in the last sprint so there are still 4 sprints worth of groomed items.
   OR The team didn't estimate/breakdown well enough so the previous story is taking multiple sprints to complete.

ii. If too many tasks are WIP (i.e., tasks aren't getting "done")
   Cause: Team members aren't working in small enough batches
   OR Team members are multi-tasking.

iii. If tasks are left NOT done at the end of the sprint
   Cause: Team members aren't estimating well
   OR Team members are not breaking the work down enough
   OR Team members are multi-tasking

iv. If all sprint tasks finished early and the team must pull more work in
   Cause: The team is overestimating the effort

v. If sprint planning takes more than an hour
   Cause: The team is not grooming well enough

vi. If sprint review takes more than 15 minutes
   Cause: The team is probably doing the retrospective as part of the review
   OR The team is talking about what didn't get done

vii. If daily scrum takes more than 15 minutes
   Cause: The team is problem solving during the daily scrum

The above list is just some examples of symptoms a team might observe/encounter and probable causes. The purpose of showing this list is to give teams some ideas of the kinds of things they should be looking for and how they might address them.

Software Engineering

According to Kinser

The retrospective is a good time to think about and discuss how the estimation process is working for the team and how they may need to adjust. It is one the hardest things to have working well for the team. Even seasoned professionals struggle with estimation. There is a lot of ambiguity involved. Each sprint is different enough from the last one to make improvements more challenging to achieve.

But it can be done with some diligence and honest evaluation. Try to use data points to fairly evaluate. For example, each developer should keep track of how much time they worked on tasks during the sprint and be honest about how this compared to how many hours they predicted they would work. How many hours each task took versus how many were estimated. This speaks to the team's overall velocity, both projected and actual velocity. If they are significantly different from each other, it affects the ability of the team to predictably pull in the right amount of work.

Once you get the projected and actual velocities to line up, only then can you really compare the velocity to the estimates from sprint planning. For example, if the team projects that they will work 80 hours collectively in a sprint (projected velocity) so they pull in enough tasks where the estimate equals 80 but then one or two people put in lots of extra hours such that the actual velocity was 100 hours and the team still didn't get all the items done. What was the issue? It could be any number of things which is why you must discuss it as a team. Maybe people worked on things that weren't on the sprint backlog. Maybe the estimates were really off. Maybe…

Why is it important to get this right? So that the team can confidently and consistently commit to the work that can be done in a sprint and consistently evaluate whether Product Backlog items are Sprintable. This gives the Product Owner the ability to improve their product roadmap (schedule when features will be released more accurately). The unknown makes people nervous. We like to have an idea of what is coming and when it will get here. We find comfort in a certain amount of predictability.

The retrospective isn't just about accurate estimation. The team can choose to focus on the quality of their code. For example, they may elect to run a static code analyzer against each feature branch before it is committed to the development branch. The team could decide what is okay and what isn't. They may elect to fix every error and every warning the analyzer produces even if it isn't in the code that was changed as part of the feature development (this would pay off some of the technical debt).

The team can choose to focus on automation of testing or automation of the review process or automation of the release process. It's really up to the team what they want to keep doing well and what they want to improve on.

Some dev teams include the Product Owner in the Retrospective as just an observer. Other dev teams may not be comfortable with them being present. Other teams may elect to include the Product Owner as an active participant so the Product Owner can put forward what they think went well and what they think could be improved on. It is up to the dev team to decide the level of the PO's participation and the Product Owner should respect that. The Scrum Master can help everyone work through it. Ultimately, in an ideal Agile environment, the dev team would be comfortable with anyone participating in the retrospective. However, I believe only the dev team should decide which items they will act on going forward and what they will change to support those changes.

Chapter 24 - Root of Agility

Food for thought

If you think Agile and Scrum are new and innovative, do some research on the Rational Unified Process (RUP). It has been around since the 1990s. It pushed for all the phases of the lifecycle to be executed concurrently. Test cases could be started as requirements were being defined. The same with designs. They also advocated for starting development on the known components of a solution even before all the requirements were identified. In many ways, RUP was a precursor to Agile and Scrum. It may have also been the precursor to Test Driven Development.

The Root of Agility

Change your mindset from robust set of features to robust solutions
Work in small batches
Merge frequently
Vertical Slices yield valuable software

To "be" an agile software engineer you need to do the above. There is a traditional thinking that we have to build the most robust set of features possible in the first release. If it doesn't have all the bells and whistles, people won't use it. History tells us otherwise. Facebook, MS Word, and Amazon did not start out with the feature rich offering you see today. They all started with very simplistic (in comparison) features. As demand increased, features were added, and others were enhanced. They evolved over time.

Agile engineers focus on solid engineering practices and processes so that their solutions are robust in terms of quality. They focus on doing a good job. They know the solution will evolve so they build it to be extended. They build it to be maintainable. They build it with the expectation that new requirements will be found, and existing

requirements will be changed.

As you gain more experience, you will find that one of the keys to sustainable progress is to have a vision and a plan for the larger solution but to implement it in small batches. In the past, the effort to test and release was significant and much less frequent than is possible today. Because of this, development took months if not years between releases. As a result, the changes between releases were complex which further complicated the release process.

With current technologies and tools, we can test and release almost immediately and with confidence. This allows us to design, build, and release individual units of work, whether it be a bug fix or a new feature or a simple enhancement. This is working in small batches. By doing this, we reduce the risk to the larger system because each change is small. And, if we are using good engineering practices, the full system is tested with each small release, so we have confidence in each release.

As a side-effect of working in small batches, team-based development must adapt by merging frequently. In the past, developers might work on code for weeks before merging their changes into the shared baseline. You can probably imagine challenges with merge conflicts if a team of 15-20 developers were merging at roughly the same time after weeks of coding in isolation. Because the trend is towards small batches of development, these merges are happening almost constantly. As a result, each developer needs to pull in the changes submitted by others, resolve conflicts (which will likely be few if any), and then commit their own changes to the shared baseline. By doing this daily, or even more frequently, large, big bang merges are avoided.

Working in small batches and merging frequently are further enhanced when the small batches are vertical slices of the system. By working with vertical slices (instead of horizontal slices), each release can add business value to the solution. The value will likely be small given the batches are small, but the value will be there and will accumulate with each subsequent release.

Using solid engineering techniques and processes, while working in small, vertical slices will result in a sustainable pace of releases that add business value consistently. This value includes being able to put new ideas out into user's hands and soliciting feedback from them quickly, and then adapting the solution based on that feedback. This is the root of agility.

Chapter 25 - Emergent Architecture

In the SDLC, we talked about doing the architectural design as the first step in the design phase. We talked about leveraging design patterns like Client-Server Architecture. The architectural design is very high level; defining systems, APIs, the

system data model, etc. We also talked about how this activity sometimes coincides with Requirements Analysis as the technical leads (or “architects”) ask clarifying questions to better understand what the solution needs to look like, who the targeted users are, the manner in which the system will be accessed, how the “-ilities” will be applied, etc.

In Waterfall projects, this is all defined upfront near the beginning of the project and then the whole thing is built in the implementation phase. In Scrum projects, I told you this would most likely be done in the Initial Grooming activity prior to starting the Scrum process. I did emphasize the need to only implement enough of the architecture so that you had a Potentially Shippable Product Release at the end of your first sprint. This required you to take a thin, vertical slice of the architecture, not implement all of it up front. This is Emergent Architecture.

Emergent Architecture is a fancy way of saying, “Let’s have an idea of what the big picture will look like but only build what we need as we need it.”

Agile Principles related to Emergent Architecture:

> #1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
> 2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
> 3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
> 9) Continuous attention to technical excellence and good design enhances agility.
> 10) Simplicity--the art of maximizing the amount of work not done--is essential.
> 11) The best architectures, requirements, and designs emerge from self-organizing teams.

As listed above, many of the Agile principles point to Emergent Architecture (#11 practically spells it out for you). The only way to deliver working software early and often is to start out with the MVP (Minimum Viable Product) and allow the architecture to evolve over time as the business needs it to. This does require the Dev Team to be okay with building software using a design and/or an architecture they KNOW will have to be replaced in the future (maybe the near future). Thus, welcoming changing requirements.

The knowledge that the architecture being implemented today might be replaced soon is not justification to cut corners on its implementation. Scrum teams may have an idea of what the future holds but it is not a certainty until it becomes the present. Meaning, you don’t know how long the architecture you are implementing is going to have to last in production so adhere to technical excellence and build it well. The idea that we are building an architecture that will likely get refactored or completely replaced in the future

seems to contradict Agile Principle #10 since we are doing work we “know” is not required long term. In other words, it seems like a wasted effort.

However, the idea of Emergent Architecture is fully supported by, and practically required by, Agile Principle #10. This is because it would be worse to spend a lot of time upfront building the final architecture that we “know” we will need eventually. It would be much more complex than what we need now, and the effort spent on it takes away from the immediate value we could deliver to our customers. For example, the long-term vision of ScavengeRUs (of which BucHunt is only a subset of), would require an architecture that can support potentially thousands of concurrent users with segregated hosting for each client with state of the art security since financial transactions would be integrated in it. To build this would take a significant effort and probably could not be accomplished in a single semester by one team. And that is all we would likely be able to build. Instead, we start with just BucHunt, a simple client-server architecture with no security to speak of and no ability to scale. Why? Because that is all that is needed for the immediate future.

The reality is this, in Scrum projects we welcome changing requirements. Not just new requirements; changing requirements. This means the team never really “knows” what the final architecture looks like. Because you don’t know with certainty what the final architecture will be, it is extremely important to always build your current architecture well (in case it has to last years) and flexible (in case it needs to morph in the future).

Some camps say that this means you don’t need to do the architectural design upfront like I recommend you do during Initial Grooming. They take the position that it is a waste of time given so many unknowns. I refer back to the Agile Manifesto; Respond to Change over Having a Plan. You can’t do this if you don’t have a plan. I believe the key is to have a plan but focus on the part of the plan that impacts the most immediate needs.

Some camps say it is more expensive to build things you know won’t fill your long-term needs because you end up refactoring everything over and over and the established implementations become an impediment to doing the right solution. This is a valid concern. However, if there was certainty in the requirements such that there was certainty in the long-term architecture, then Scrum is not the right model, and an Iterative approach would be more appropriate. Scrum is used when there are few certainties, and the road ahead has numerous forks.

Building a Flexible Architecture

Software Engineering

According to Kinser

One of the best ways to build a flexible architecture is by decoupling the components as much as possible so that each can evolve and morph independently of the rest of the solution. There are design patterns like Broker and Façade that can be leveraged for this. Clear segregation of responsibilities is another. The ones we will talk about in SE II are Microservices and Serverless architecture.

A microservice is a standalone executable that does a very specific job. It has no knowledge of the rest of the system. It is similar to a service in Service Oriented Architectures. It is similar to a library or a third-party product that you call through an API. In fact, microservices most often leverage an API Gateway. Most APIs have no business logic in them. They simply pass calls through to a backend. Sometimes it does a transformation on the data. An API Gateway is like this but can also contain business logic.

Let's take a super simple example. If your application has an "about" page that describes the purpose of your product and maybe a little history, then that content could be pulled in from a microservice. The page calls an API Gateway to read in the content. The API Gateway then calls the microservice. The microservice may have the content hard coded, or it may read it from a text file, or maybe it reads it from a database. It doesn't matter to the about page. Later on, your application adds localization and supports multiple languages. Now the microservice can be updated to take in a language parameter that defaults to the original native language. The microservice serves up the correct language content. If no language is specified, it serves up the default language. The parameter could be an enumerated type or fixed list so that no one can call for a language that isn't supported and additional languages can be added at anytime without impacting the interface. An alternative way is to have a separate microservice for every language supported. The API Gateway decides which one to call based on the language selected.

I think this article is helpful: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/. Jackson Coffey, SE II Spring '23, found this helpful video on Microservices; https://www.youtube.com/watch?v=lTAcCNbJ7KE

<add in a description of incremental development strategies like 1) hard coding content in HTML, 2) moving that to be pulled in from the main app (e.g. via a REST call) 3) moving it out of the main app and into a microservice so the main app does a direct call to the microservice using the same or similar REST call the browser still uses to get the content from the main app (basically the main app is now a pass through) 4) move the content to a text file that the microservice loads when the call comes in 5) insert an API gateway inbetween the main app and the microservice. Each step is a working solution (vertical slice). Each step adds values from an architecture perspective as it decouples

content from client facing code to the point that it can be modified by altering a text file (no code change).

Serverless architecture is the other topic we will cover in support of flexible architecture. If you know about or have used Amazon's AWS or Microsoft's Azure, then you are familiar with Serverless Architecture. The idea is that you don't have to worry about or manage your physical hardware. You rent it and you only pay for what you need. As traffic to your solution increases, the hosting service can automatically scale up your hardware allocations or add load balancing or… etc. These hosting services take a lot of the complexity of scaling and routing off the list of responsibilities teams have to deal with. They can also host your test and staging environments. In fact, they can bring these up and down on demand so that you don't have to pay for them when you aren't using them.

The sophistication and maturity of these hosted environments has steadily increased and the costs have dropped as well. Even large companies that can afford their own hardware have moved to hosted services. One of the drivers for this is upgrades. As the hardware capabilities advance, the hosting services automatically port solutions to the new equipment. The client doesn't have to deal with the accounting department for depreciation of tangible assets or do a study to identify the cost savings of buying new equipment. They can just focus on their business.

The downside to using a hosting service is that the client doesn't have any control over the quality of their operations. If the hosting service goes down, the client can't do anything about it except wait for them to fix the situation (https://www.cnn.com/2023/01/25/tech/microsoft-cloud-outage-worldwide-trnd).

In addition to this dependency, there is something called Vendor Lock In. Vendor Lock In predates hosting. For example, almost all relational databases support SQL as a standard way to interface to their databases. However, each database vendor extends SQL with customizations and convenience services. Some vendors have even been accused of purposefully making it harder for customers to just use the standards-based SQL. Why? If a customer can interface with a database vendor's solution using just SQL then they can port their application to any other database vendor with minimal effort. If the customer takes advantage of the vendor specific services, it is much harder (time consuming and labor intensive) to convert to a different vendor. Oracle is notorious for this tactic as is Microsoft.

The hosting services like AWS and Azure do the same thing. They provide customers with vendor specific tools and services that make it easier to use their hosting services which locks them in. Both hosting vendors may offer Ubuntu servers for a production

deployment but there will be differences in how this is done so that it isn't straight forward how you could switch to a different hosting vendor.

Chapter 26 - DevOps (History and current state)

DevOps – a short history

In SE I, we included Release, Maintenance, Upgrade/Update as SE phases. Most sources stop the SDLC at Testing. I included these last three because of DevOps. Operations teams are normally responsible for these three phases. Development can make them easier or harder for Operations. DevOps is shorthand for collaboration between the Development groups and the Operations groups in an organization.

When releases were years apart for the same product, the responsibility for the product shifted from the development group to the operations group at the point of release, often with inadequate communications (figuratively throwing it over the wall). Think about how good you are at understanding the federal tax code. You only deal with it once a year. Because of this, you probably don't really do things throughout the year to make it easier to file your taxes or to minimize your tax obligations. If you had to file paperwork every week or every two weeks, you would probably hate it but you would also be more informed and better at it just out of necessity.

When the SDLC was long-lived, taking years for each release, not only was there a chasm between operations and development. There was also a chasm between the testing group (aka QA team) and the support team. The Dev Team would build a product. The QA team would test it until time ran out. Then it was put into production for the operations team to deal with. Any issues they had that they could fix they did because waiting for the support team (aka maintenance team, which was often composed of new hires that never worked on the product) took way too long. There were lots of silos of responsibility and a lot of blame-games happening.

With Iterative and related methodologies, the releases began to occur closer together (maybe a year apart or just several months). This created not only a need for better handoffs to QA teams and operations, but the value of collaboration was much more. Reducing the complexity of releases had a noticeable, positive effect on all parties involved.

The first step in reducing the complexity of releases is to have a more robust testing process. When you only test the latest changes, it is easy for negative side-effects on existing features to slip into production. Regression testing started to be even more important. Automating these tests helped ensure they were done consistently and more quickly. This caused collaboration between testing teams and Dev Teams.

The second step in reducing the complexity of releases is to have a more robust release process. If you only release a solution once every two or three years too many things change between releases, and it is hard to have a standardized way to do the releases. As they get closer together, the steps can be standardized and automated which speeds up the release event and makes it more reliable.

The third step in reducing the complexity of releases is to support the operations team. Taking their needs and wants into consideration when building the product can make the release process more effective and efficient. It also improves the ongoing support of the product in the production environment. The more frequent releases also enable the development and maintenance teams to be more responsive to issues the operations teams face. The operations teams don't need to wait years for fixes anymore.

You see the pattern.

So, DevOps has always been around. The major difference now is that the interactions between the groups are almost constant. Thus, both groups benefit from working collaboratively with each other.

The separation of the groups as described here was also reflected in how they were organized. Many years ago, the VP of operations reported to the same boss as the VP of development and that was the only place any regular interaction occurred. Over time, the two groups were put under the same VP and eventually the same Director. In some current organizations there is very little if any distinction between the two because they are on the same teams, working side by side.

Another way to look at it is through the tools people used to get their work done. As software became more common, tools like IDEs and static code analyzers were created. Test tools initially existed separate from development. Operations tools were separate from both of these. Security tools were also isolated. As time went by, the IDEs began to merge with the test tools. We are currently seeing IDEs merging with Operation tools. Eventually they will merge with Security tools as well. The question will be, once the tools are merged will they be too complicated to use or will AI come to the rescue.

Think about MS Office. Originally, there was no MS Office. There was just MS Word. As it captured more market share, they bundled it with Excel and PowerPoint (even though they were completely different executables). They started integrating the solutions to be more compatible. Then they expanded the bundle to include Outlook and other apps. Now it is extremely complicated to use. (https://www.thewindowsclub.com/history-evolution-microsoft-office-software). Will DevOps tools follow a similar path? Probably.

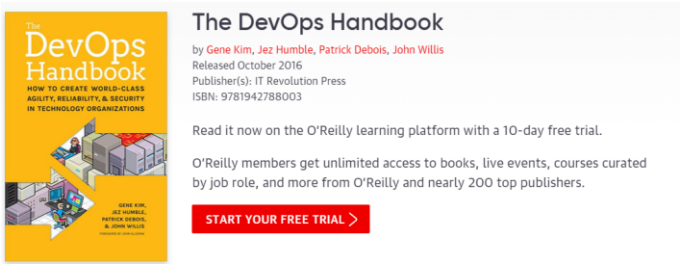
Who will win? Watch the market shares.

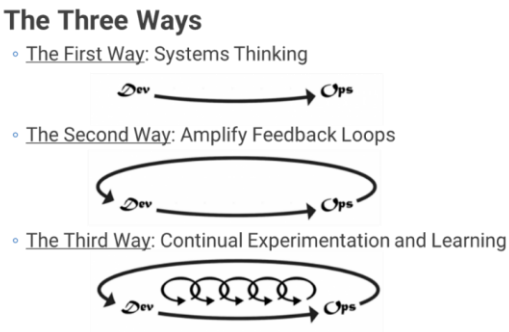
DevOps Now

By the book

Software Engineering

According to Kinser





From: The DevOps Handbook

This diagram is from 01_SoftwareProjectManagement.pptx and based on Kim, G., Humble, J., Debois, P., & Willis, J. (2016). The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations. Portland, OR: IT Revolution, Inc. ISBN: 978-1942788-00-3 (shown in the image just prior).

For the purposes of this course, you only really need to read sections 3, 4 and 5 of the book. The rest is either fluff or contains material aimed at the managers/directors of departments (not developers).

The First Way: Systems Thinking (From: The DevOps Handbook)

- Enables fast left-to-right flow of work from Dev to Ops to Customer
- Requires:
  - That we make work visible,
  - That we reduce batch sizes and work intervals,
  - That we build in quality by preventing defects from being passed along,
  - That we constantly optimize for global goals
- Practices:
  - Continuous Build, Integration, Test, and Deployment;
  - Creating environments on demand;
  - Limiting Work in Process (WIP);
  - Building systems that are safe to change

Software Engineering

According to Kinser

If you really dig into the First Way, it will start to sound like a restatement of the Agile Principles and the Root of Agility. So, in many ways you have already learned quite a bit about the First Way in SE I. We didn't really talk much about Continuous Integration and Continuous Deployment (CI/CD). However, these evolve naturally from working in small batches and merging frequently as part of Scrum sprints where you have a potentially shippable product every week or two.

How we break our stories down during Sprint Grooming and Sprint Planning will either enable or hinder our ability to develop in small batches (aka vertical slices). Part of the First Way of DevOps is to build the software in a way that supports Continuous Integration/Continuous Deployment (CI/CD) or micro-releases. Even if your solution doesn't need to be updated daily, building the software in this way gives the organization that option. As you get used to this approach, it can be just as efficient as traditional development because so many steps along the release pipeline will be automated. If you must do it manually, it might be better to do two-week sprints with releases at the end of each.

One might say that The First Way is really defining non-functional requirements that support more efficient operations. A non-functional requirement that defines a target scalability is directly related to operations in how the production environment must be configured and maintained. The First Way defines non-functional requirements on the process that dictates how software is built.

The Second Way: Amplify Feedback Loops (From: The DevOps Handbook)

- Enables the fast and constant flow of feedback from right to left at all stages of the value stream.
- Requires:
  - That we amplify feedback to prevent problems from happening again
  - That we enable faster detection and recovery
- Results:
  - Creating quality at the source,
  - Generating knowledge where it is needed,
  - Creating safer systems where problems are found and fixed long before a catastrophic event

It is very common for solutions to track key data points around customer interactions and report this back to the business. You probably get prompted during installation and setup of products all the time to allow usage data to be collected and sent back to the provider (MS does this, Google, a lot of phone apps, etc.) These data points help their marketing teams, and their product owners identify which features get used the most (have the most value) and which do not. It helps them see where customers have trouble. For example, Amazon, very early on in its existence, tracked how many carts were abandoned. This is where a customer, or a potential customer, selected items into their cart and then never went through with the purchase. This might be because they changed their mind or found a better deal on-line or were confused by the process of checking out. As a result, Amazon tried several different ways to improve checkouts, using these same data points to evaluate each idea. One solution that stuck but is

controversial is that Amazon makes third party sellers sign a contract that forbids them from selling their products anywhere else for less than they offer it on Amazon (https://www.engadget.com/california-amazon-antitrust-lawsuit-prices-195937087.html) Another is where they offer “coupons” that can only be redeemed at the end of the checkout process. Basically, they advertise the price they want to sell something at, then tell you it is only available with a coupon, which can only be redeemed at checkout. It’s all psychology based and it works. One last example is the One-Click checkout option. If you are ordering something for yourself to be delivered to the default address using the default credit card, you only need to click one button. Initially there were all kinds of security concerns about this simplification but the users like it so much those concerns have been pushed to the back burner.

Data collection like this can also benefit DevOps. If we know that a particular feature in a product isn’t used by enough customers to make it worthwhile, we can remove it from the product which reduces maintenance and testing and many other ongoing costs. If we know 95% of our customers only access our site via mobile phones, then we can customize our UI and access points to better support these types of connections (mobile phones are notorious for dropped connections so we would move to more stateless transactions).

Data could tell us where the slowest point in the release process is so that we can streamline it and increase productivity. Tracking the types and frequency of issues operations staff face, we can potentially alter the solution or our development processes to reduce their workload and improve efficiency. Without data you must guess what changes will matter most.

The First Way emphasizes automation from left to right in the value stream. Monitoring is a way to automate from right to left. What needs to be monitored? How often does it need to be measured? How often does it need to be reported? What conditions warrant special attention and by whom? All these questions and several others can result in arguments and stagnation. In many ways, it’s better to just start measuring something and see what it tells you.

There are several websites that will define different types of monitoring activities like Alerts, Trending, Telemetry, etc. https://www.atlassian.com/devops/devops-tools/devops-monitoring gives an overview of monitoring but doesn’t really say what you should monitor. Product reviews for third party monitoring tools can be found here https://logit.io/blog/post/production-monitoring-tools-software/ (including Splunk). This site gives different categories of data points and examples of what to monitor (plus a pretty good explanation of metrics vs monitoring vs alerts). https://www.digitalocean.com/community/tutorials/an-introduction-to-metrics-monitoring-and-alerting .

ETSU offers courses in data analytics, so I won’t try to steal their thunder. But here are some of the things I’ve leveraged over the years.

Software Engineering

According to Kinser

Look for things that will tell us when things have gone wrong (reactive monitoring)
- Server goes down
- Dropped transactions or connections

Look for indicators that will tell us when things might go wrong (proactive monitoring)
- Increase in failed login attempts (getting hacked? Or increase in usage)
- Memory usage steadily rising (memory leak? Or increase usage?)

Some things to consider tracking:
- Track login attempts, failed/successful, login device
- Track logouts (manual vs timeouts)
- Track response times
- Track informational, warning, errors, critical, crashes
- Track code metrics (enabled via IDE, optional external static analyzers)
- Track time it takes to go from Sprint Review to Production deployment

Be careful what you measure. If you only track/chart the count of open tickets, then the number of tickets can be skewed by stale tickets the longer the solution is in production. It's entirely possible that a ticket (that is still open from three years ago) is no longer a problem but because it is still "open" it is included in the count. If you try to count the number of tickets resolved each sprint as an indication of productivity you are ignoring the fact that not all tickets are equal in complexity and importance. Should you focus on fixing tickets or doing things to prevent the problems that cause tickets?

According to Kinser, never try to measure the productivity of the staff. This violates the Agile Principle of trust. It is better to set SMART goals for the team and focus on the progress towards these.

The Third Way: Continual Experimentation and Learning (From: The DevOps Handbook)
- Enables the creation of a high-trust culture that supports a dynamic, disciplined, and scientific approach to experimentation and risk-taking.
- Requires:
  - That we facilitate organizational learning from both successes and failures;
  - That we design our system to multiply the effects of new knowledge;
  - That we transform local discoveries into global improvements
- Results:
  - No fear of change or fear of risk
  - Faster innovations
  - Global knowledge

In past courses, you've probably discussed Technical Debt. This is when we cut corners to achieve a deadline or when we, for whatever reason, kick the can down the road by taking shortcuts now. https://en.wikipedia.org/wiki/Technical_debt

Software Engineering

According to Kinser

There is often an implication in many companies that refactoring or reworking the shortcut adds no value because there will be no new feature(s). The DevOps idea is that the fragility of shortcuts and technical debt reduces the overall quality of the process and the solution, resulting in constraints on or obstacles to the flow of work. To address this, DevOps promotes the idea there is value in improving things even if no new features are added. So, the value in paying off the technical debt is cost savings. Also, these improvements don't have to be organizational changing events but can occur at the individual or team level and can have a compounding effect.

For example, if a team always pushes their solution to production via manual steps, there is a high probability of human error and a lot of wasted time because human response times are slow in comparison to computers. So, if the team were to automate even part of the process via scripts, cron jobs, third party tools, etc., there would be less risk of error, improvement in release times, and more time to work on other items like new features. These improvements are compounded because you would reap the benefit with every release. If you improve release times by 10% and you release weekly, that is significant. It seems to make sense but oftentimes there is pressure to just get things done. The cost of automating is real. The steps have to be defined in detail, research has to be done as to which tools to use, some tools may need to be purchased and then have a recurring fee, training will be required, etc. The result is that the first few releases will more than likely be slower than the manual process. It's an investment.

I was once temporarily assigned to do maintenance on the F16 fighter plane while awaiting my clearance for another project. In the few months I was there, I figured out that there was a lot of technical debt in the data blocks and made changes (with approval) that reduced the storage of the system I was assigned to by 20%. Not a new feature. But with embedded systems, freeing up that much memory was significant. This is an example of the Third Way because cleaning up the data blocks was not even a defined task but something I did on top of my regular assignments. Being new to the team caused me to question things the rest took for granted. Fortunately, the team was open to my experimentation and ultimately supported my changes. (BTW, I was passed over for a raise that year because I hadn't been employed for the full review cycle. Yes, I'm still a little bitter.)

At another company where I was recently made the director of software development, one of the products we built was an embedded system to collect telemetry on power lines via something called an RTU (Remote Terminal Unit). The team responsible for the RTUs was only about 5 people. The team lead approached me soon after I became the director to ask if they could switch all their embedded solutions to C++. I asked him if everyone on the team knew C++ or if training would be required. The team wanted to learn C++ and saw this as an opportunity to gain experience with the language by refactoring a solution they were very familiar with. After discussing the pros and cons, pending commitments, etc., we came up with a 6-month plan to convert everything. This is an example of the third way. Did it result in a better solution? I don't remember. What I remember was that the morale of that team went through the roof. Apparently, they had felt ignored and were never allowed to innovate their products before then.

At a different company where I was just a manager, one of my team members who had been there for 14 years was very withdrawn and only put forth the minimum effort. He expressed no interest in his work, his career, or the company. After a couple of months, I was able to find out from various sources that he had grown salty because he had made multiple suggestions for improvement to his team's area of the product and to the processes we followed only to be ignored by the previous manager (who apparently didn't want to rock the boat or take any chances). I found one of his suggestions in the repository (a kind of product backlog) and assigned it to him. Initially he said he didn't have time to work on it. I got the team to support him by taking on some of his other assignments (and I made a few tasks just go away) so that he had time. After 4 weeks of working on his old idea, his energy turned positive, and he started to become more engaged. It took a few more months but eventually he got excited about what he did. He left the company a few months before I did but he was a much happier person overall. This is also an example of the third way. I took a risk, and it didn't really pay off for the company but it did pay off for the individual.

Conclusion

Software engineering is not the same as software development. Software engineering is about taking a disciplined, professional, structured, methodical, rational, and purposeful approach to building solutions. It takes more than technical know-how. It requires team dynamics, business acumen, a strong work ethic, and, on occasion, a touch of humility. This is why you are getting a degree and not just following YouTube How-To videos all day.

In SE I, I've tried to instill in you the fundamentals of the software development life cycle; Requirements, Design, Implementation, Test, Release, Maintenance, and Upgrades/Updates. Understanding this natural progression of thought is how we apply the fundamentals of problem solving in our chosen profession regardless of your degree concentration. Concurrently I exposed you to a variety of topics related to jobs; goal setting, elevator pitch, resumes, interviewing, a day-in-the-life, and how raises and promotions work. I've also brought in concepts like Tuckman's Team Formation; Forming, Storming, Norming, and Performing. We explored source control and versioning of software but also other artifacts like database scripts, resource files, and documentation. I tried to show that it's best to have a plan and that is why we have project management because complex solutions require coordination, planning, change management, and risk management. While the course was not a coding class, we did talk about coding practices, and most importantly that commenting your code is about the WHY you wrote it the way you did and WHAT it is doing. Many of these practices translate to the other concentrations as well. We ended the course with a focus on the Agile philosophy and the Scrum methodology.

Software Engineering

According to Kinser

Throughout SE I, I tried to focus you on understanding the materials provided by emphasizing putting the concepts into practice over the outcome of any particular exercise. I wanted you to engineer the solution. The more you take an engineering approach to problems the better your outcomes will be, but it takes time.

In SE II, I pushed you to do things more robustly. While a few new concepts were added, most of the semester was spent doing the same things we did in SE I. Why? Because without a solid understanding of the fundamentals, the more advanced concepts will never work. Each team evolved at different speeds because it was made up of different people. Each team focused more heavily on different parts of the engineering process; some on deployment, some on source control, some on automation, etc. I tried to create an environment of trust so that each person felt comfortable trying new things without fear of failure. I tried to explain the WHY behind a lot of what we learned and what other courses taught you. I tried to help you bring as much of your academic career forward as was practical so that you could see through practice how it all fits together.

I realize that the format and style of this course was not what you were probably accustomed to. It is fairly consistent with how I led my teams and departments. Because almost all of you are seniors and will be seeking to start your careers soon, I wanted to give you some insights and training on what to expect and how to succeed based on my many decades of experience. In most companies you will not have a detailed list of expectations that will earn you a raise or a promotion if you just do what you are told. The field of software engineering is about creativity. Even if you are asked to solve a well-known problem, there will be an expectation that you do it better than in the past; in a way that differentiates your company from others; that adds more value. It will be more common that you will have to create a solution to a problem that is so new that the problem itself is not fully defined. To be successful, you have to find a way to chart your own career's trajectory. You have to find your own ways to exceed expectations.

I wanted to present each of you with problems that you could not solve on your own because most problems of any real value require a team to solve them. Being able to work with a group of people with different skills than yours, with different goals than yours, with different experiences and values than yours, is something that will serve you well your entire career. I encourage you to be okay with not being the one with all the answers. I encourage you to see the greatness in others and help them celebrate their successes.

I hope I have served you well.

Software Engineering

According to Kinser

Appendix A: Mapping SE courses to Objectives and Outcomes

Below is a description of how SE I and SE II map to Department Objectives and Student Outcomes.

Department Objectives:

- Exhibit high standards of responsibility and ethics
    a. For ethics, SE I has a lecture and take-home exercise on ethics in computing, referencing the IEEE and ACM sites. They receive feedback on their exercise submissions.
    b. Both courses use a contract labor grade approach where students must perform against a job description. All course materials are available at the outset of the semester along with access to Topic Quizzes tied directly to each course's learning outcomes (they can take the quiz anytime they like). If a student does not do any of the items in the job description is a "fails to meet expectations" and must be compensated for if they do not want it to detract from their grade. Students are responsible for finding their own ways to "exceed expectations" in order to achieve a greater than C grade in the course.
- Communicate effectively with colleagues and other stakeholders
    a. SE I has in-class exercises almost every class session where students must form teams of varying sizes, collaborate, and put a topic into practice. There is also a 4 week long project they must work on as a team.
    b. SE II has a semester-long project they must work on as a team.
    c. SE II also has two presentations on technical topics related to SE II for each student, for which they receive feedback
    d. Both courses require each student to write two essays, for which they receive feedback
    e. Both courses require each student to write weekly status reports, for which they receive feedback
- Keep current in their fields of expertise
    a. In both courses, students leverage modern tools in their activities
    b. In SE II, the two presentations each student does are based on their research of modern topics or tools or techniques

    c. In both courses, students are allowed to leverage AI (e.g. ChatGPT) in many of their activities within the bounds of legal and ethical constraints
- Apply computing knowledge to real-world problems
    a. In both courses, the project they are assigned mirrors real-world applications
    b. In SE I, students are given an existing code base that they must evaluate and extend based on a set of requirements provided to them
    c. The semester long project in SE II requires students to build a solution starting with a simple vision statement (they must generate requirements and follow the SDLC to produce a working solution)

General Student Outcomes

- Analyze a complex computing problem and apply principles of computing and other relevant disciplines to identify solutions
    a. The in-class exercises and end of semester project in SE I require student to apply specific SE concepts in order to arrive at a solution (emphasis is placed on their ability to demonstrate understanding of the SE Topic)
    b. The semester long project in SE II requires students to build a solution starting with a simple vision statement (they must generate requirements and follow the SDLC to produce a working solution)
- Design, implement, and evaluate a computing-based solution to meet a given set of computing requirements in the context of the program's discipline
    a. The SE I end of semester project requires the students to extend an existing code base
    b. The semester long project in SE II requires students to build a solution starting with a simple vision statement (they must generate requirements and follow the SDLC to produce a working solution)
- Communicate effectively in a variety of professional contexts
    a. SE I has in-class exercises almost every class session where students must form teams of varying sizes, collaborate, and put a topic into practice. There is also a 4 week long project they must work on as a team.
    b. SE II has a semester-long project they must work on as a team.
    c. SE II also has two presentations on technical topics related to SE II for each student, for which they receive feedback

d. Both courses require each student to write two essays, for which they receive feedback
e. Both courses require each student to write weekly status reports, for which they receive feedback

- Recognize professional responsibilities and make informed judgments in computing practice based on legal and ethical principles
    a. For ethics, SE I has a lecture and take-home exercise on ethics in computing, referencing the IEEE and ACM sites. They receive feedback on their exercise submissions.
    b. In both courses, students are allowed to leverage AI (e.g. ChatGPT) in many of their activities within the bounds of legal and ethical constraints
    c. In both courses, students must ensure their solutions do not include 3rd party components for which they do not have a license and cannot infringe upon trademarks or copyrights in their creation of a solution.
- Function effectively as a member or leader of a team engaged in activities appropriate to the program's discipline
    a. SE I has in-class exercises almost every class session where students must form teams of varying sizes, collaborate, and put a topic into practice. There is also a 4 week long project they must work on as a team.
    b. SE II has a semester-long project they must work on as a team.
- (CSCS) Apply computer science theory and software development fundamentals to produce computing-based solutions.
    a. The SE I end of semester project requires the students to extend an existing code base
    b. The semester long project in SE II requires students to build a solution starting with a simple vision statement (they must generate requirements and follow the SDLC to produce a working solution