Software Development Notes

Lessons Learned

Ladislav Šulák <laco.sulak@gmail.com>

April 28, 2021

Contents

1 General

- *"Check your context. Only then, proceed"*, Udi Dahan.
- Refactoring, a good quote: *"When we really dive into the reasons why we can't let something go, there are really only two: An attachment to the past, or a fear for the future."*
- If we have learned anything over the last couple of decades, it is that programming is a craft more than it is a science. To write clean code, you must first write dirty code and then clean it.
- In most web-based development environments, the architecture can be broken down like this:
  - Local development and unit testing on the developer's machine
  - Development server where manual or automated integration testing is done
  - Staging server where the QA team and the users do acceptance testing
  - Production server
- Customers don't know exactly what they want and thus don't always tell the truth. They use their terms and their contexts. They leave out significant details. So, how can you possibly deliver a software project to someone who isn't telling you the whole truth about what they want? It's fairly simple. Just interact with them more. Challenge your customers early, and challenge them often.

How to Learn, The 10-step system

For example, learning technology - books are good, if you have a lot of time, but it is unnecessary and very time-demanding, and everything truly important is mixed with tons of little details! So basics of an alternative approach:

1. **How to get started**: What were the basic things I needed to know to get started using whatever I was learning?
2. **The breadth of the subject**: How big was the thing I was learning and what could I do with it? I didn't need to know every detail to start, but if I had a decent overview of what I could do and what was possible, I could always find more details later.

3. **The basics**: Beyond just getting started, what were the basic use cases and the most common things I'd need to know to use a particular technology? What was the 20% I could learn that would cover 80% of my daily usage?

It turns out that getting those three pieces of knowledge isn't as easy of a task as it might seem. Learning how to get started with a technology can be a challenge, and it's often difficult to find out what is the 20% you need to know to be 80% effective with a technology. So a solution to address these problem is the following:

1. **Get the big picture** - determine how big the topic is and what kind of subtopics exist within a few hours of research. To complete this step, do some basic research on the topic you want to learn about. You can probably accomplish most of this research using internet searches. If you happen to have a book on the subject, you might read an introductory chapter to skim through the material. Don't spend too much time on this step, though. Remember, the goal isn't to actually learn the topic here, but to just get a big picture of what it's about and how big it is.

2. **Determine scope** - now that you have at least somewhat of an idea of what your topic is and how big it is, it's time to narrow down your focus to determine what exactly you want to learn.

3. **Define success** - without knowing what success looks like, it's both difficult to aim and to know when you've actually hit the target. Before you try to learn anything, you should have a clear picture in your mind of what success will look like. When you know what your target is, you can more easily work backwards from the goal to determine the steps you need to take to get there. The goal of this step is to come up with a clear and concise statement that will define success for your learning endeavor.

4. **Find resources** - in this step you want to find as many resources as possible for learning about the topic you've selected. Don't worry about quality at this point. This is similar to a brainstorming step. Later on you'll filter your resources and select the best ones, but for now you want to get as many different resources as possible.

5. **Create a learning plan** - now that you have some resources, you can use those resources to get an idea of what you should learn and in what order you should learn it. For this step, you need to create your own learning path. Think of it as an outline for a book you'd write on the subject. In fact, your learning path will probably be very similar to the table of contents of a book when you're done. You basically want to end up with a series of modules you individually focus on learning until you reach your final goal. A good way to create your learning plan is to see how others are teaching the subject you want to learn about. When I'm working on this step, I'll often look at the table of contents of several of the books I've chosen as possible resources from step 4. If five different authors have chosen

to break up their content into the same sets of modules and the same ordering, chances are I should make my learning plan follow a similar approach.

6. **Filter resources** - at this point, you probably have plenty of books, blog posts, and other resources for learning about digital photography, but the problem is that you can't possibly utilize all of them. Much of the data is redundant and not all of it will fit your learning plan. It's not practical to try to read 10 books and 50 blog posts on a subject—and even if you did, a large portion of that information would be duplicated. It's important to narrow down your resources to a smaller list of the best ones to help you achieve your goals. For this step, go through all the resources you've gathered in step 4 and figure out which ones have content that will help you to best cover the content in your learning plan. Also take a look at reviews and try to determine which resources are of the highest quality. I usually will look at the Amazon reviews for the books I'm considering purchasing and narrow it down to the best one or two books that I think will provide me the best bang for my buck. Once you've completed this step, you're ready to move on to the first module of your learning plan. You'll repeat steps 7–10 for each learning plan module until you've made it to your destination.

7. **Learning enough to get started** - there are two common learning mistakes that most people make, myself included. First, there's the problem of jumping in without knowing enough—acting too soon. Second, there's the problem of preparing too much before jumping in—acting too late. You want to strike a balance between the two and learn just enough to get started, but not so much that you don't get to explore on your own—where you end up learning the best. For this step, the goal is to get just enough information about the topic you're learning about to be able to get started and to play around in the next step.

8. **Play around** - this step is both fun and scary. It's fun because you get to do exactly what the step says: play around. But it's scary because the step is completely unbounded. There are no rules. You can do whatever you want to do for this step. It's up to you to decide how to best execute this step. For this step, you want to take what you learned from step 7 and actually get started. Don't worry about outcomes. Just explore. Implement a smaller project if you want. Write down the questions that you have but don't have answers for. You'll have the opportunity to look for the answers to those questions in the next step

9. **Learn enough to do something useful** - take as much time as you need to thoroughly understand your subject matter by reading and experimenting, watching and doing Remember, though, you still don't have to completely consume every single resource you gathered. Only read or watch the parts that are relevant to what you're trying to learn right now. There are no golden stickers given out for reading a book cover to cover. Use the resources to help you teach yourself, driven primarily by the questions you've come up with by playing around. Finally, don't forget about your success criteria that you defined in step 3. Try to tie what you're

learning back to your ultimate goal. Each module you master should in some way move your forward toward your final destination.

10. **Teach** - if you want to learn a subject in depth, if you really want to gain understanding about a subject, you have to teach it. There's no other way. In reality, you only need to be one step ahead of someone to teach them. It's the only way to know for sure that you've learned something, and it's a great way to fill in the gaps in your own learning as you try to explain it to others. You can teach what you've learned in many different ways. You could write a blog post or create a YouTube video. You could even talk to your spouse about what you've learned and explain it to them. The important thing is that you actually take some time to take what you've learned out of your own mind and organize it in a way that someone else can understand. When you go through this process, you'll find that there are many things that you thought you understood that you didn't. You'll also begin to make connections that you didn't see before and simplify the information in your head as you try to condense it down and regurgitate it. Perhaps a good way that teaching is best approached is from a humble perspective, but with an authoritative tone. When you teach, you don't act like the knowledge you have makes you in some way better or smarter than your student, but you do teach with confidence, firmly believing what you're saying. No one wants to learn from someone who is unsure of what they're saying, and they also don't want to be made to feel stupid when they are being taught.

So to explain it, it all starts with getting a basic understanding of what you're trying to learn—enough to know what you don't know. Then take that information and use it to define the scope of what you want to learn, along with what success will look like. Armed with that knowledge, you can find resources—and not just books—to help you learn what you want to know. Finally, you can create your own learning plan to chart the course you're going to take to learn your subject and filter the materials down to just the best ones that will help you achieve your goal. **In the first 6 steps, you'll focus on doing enough research upfront to make sure that you know exactly what you're attempting to learn and how you'll know you're done. You do them just once. Steps 7 - 10 are repeated for each module you end up creating in your learning plan.**

Focus

- All you have to do is to prepare all possible conditions to avoid internal and external interruptions and resist for enough time - maybe 10 minutes.
- It is all about momentum. If you last for some time concentrating, it will be easier and easier. After some time, you will forget about food, sleep, tiredness, and so on. Welcome in the "zone".

Productivity and Planning

There exist many techniques for productivity. Some of them are Getting Things Done, Pomodoro Technique, or Don't break the chain (Seinfeld). The true secret to productivity: small things done repeatedly over a long time period. Routine (but don't be absolutely obsessed with your routine, there are unpredictable things that may happen). The following Kanban-related technique is very interesting (created by John Sonmez):

- You can use software like this: https://kanbanflow.com/.
- You may have columns for each day, and a special column "next week" (if you cannot do it in a given week).
- Tasks can be in states such as "not started", "in progress", and "done" (the smaller task the better - something between 30 and 120 minutes for instance).
- You should plan the whole quartal, month, and week. And ideally at their beginnings.
- You may use the Pomodoro Technique throughout the day to focus on a single task at a time and to work through the task list in Kanban board. **Pomodoro Technique**:
  - The basic idea is that you plan out the work you're going to do for a day. Then you set a timer for 25 minutes and work on the first task you've planned. You work only on a single task at a time and give it your complete focus for the full 25 minutes. If you're interrupted, there are various ways of handling the interruption, but generally you strive to not be interrupted at all. You never want to break focus.
  - At the end of the 25 minutes, you set a timer for 5 minutes and take a break. That's considered one pomodoro. After every four pomodori, you take a longer break, usually 15 minutes.
  - Technically, if you finish a task early, you're supposed to dedicate the remaining time to "overlearning." That is, you continue to work on the task by making small improvements or rereading material if you're trying to learn something. Some people tend to ignore this part and move on to the next task immediately.
  - Using the Pomodoro Technique, you can start thinking about your week in terms of a finite resource of pomodori.
  - Pomodoro technique can teach you a good lesson on prioritization. If you have just a certain amount of work (pomodori units) planned, you are extra careful how to split what you want to achieve to pomodori.
  - This technique has also psychological benefit - you can exactly control how much time you dedicate to some task during a day. When you have a goal of x pomodori for the day and you get that goal done - a goal you can actually

control - you know you did what you were supposed to do that day and you can give yourself permission to feel good about it - and more importantly - relax. And you can even more enjoy your free time. If you hit your pomodori goal - you are free to do whatever you want that day.

  - The Pomodoro Technique also forces you to focus, so when you do a full day's worth of work using the Pomodoro Technique, it ends up being a lot more work than you might normally be used to.

- For being even more efficient, you may use pomodoro timer for a single task: https://pomodoro-tracker.com/, or use can use a default one in Kanbanflow app mentioned above.

- You can accomplish for example 10 pomodori each day (which should be about 5 hours of hard, focused work). You have to track how many pomodori you done, and you have to set how many you want to achieve.

- Don't forget about breaks and free weeks! Using this technique from long-run can be unbearable for some people. Have vacation and have weeks without this technique - do just what you feel and like to do. Or have day off every once in a while.

- Create quotas! Create a repeatable task, define how many times it should be done, and commit to it! The whole system falls apart if your commitment is weak, so you have to choose attainable and maintainable quotas. Don't commit yourself to something you know you can't do; otherwise you're setting yourself up for failure. Start with small commitments and make them bolder as you become successful at reaching them. For example here are some quotas:
  - I will exercise 5x each week.
  - I will do cardio 3x each week.
  - I will create one blog post each week.
  - I will get 50 pomodori done each week.

Conway's Law

- Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure.

Jimmy's Law

- A broken, dysfunctional organization driven by meeting unhealthy goals and metrics will produce broken, dysfunctional systems.

LeBlanc's Law

- We've all looked at the mess we've just made and then have chosen to leave it for another day. We've all felt the relief of seeing our messy program work and deciding that a working mess is better than nothing. We've all said we'd go back and clean it up later. Of course, in those days we didn't know LeBlanc's law: *"Later equals never."*

Law of Demeter

- LoD is principle of least knowledge is a design guideline for developing software, particularly object-oriented programs.[1]
- Objects hide their data and expose operations. This means that an object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.
- The method should not invoke methods on objects that are returned by any of the allowed functions. In other words, talk to friends, not to strangers.
- More formally, the LoD for functions requires that a method m of an object O may only invoke the methods of the following kinds of objects:
  - O itself
  - m's parameters
  - any objects created or instantiated within m
  - O's direct component objects
  - a global variable, accessible by O, in the scope of m
- **Summary**
  - Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
  - Each unit should only talk to its friends; don't talk to strangers.
  - Only talk to your immediate friends.
- **Advantages**
  - The advantage of following the LoD is that the resulting software tends to be more maintainable and adaptable.
  - Since objects are less dependent on the internal structure of other objects, object containers can be changed without reworking their callers.
- **Disadvantages**

[1] https://en.wikipedia.org/wiki/Law_of_Demeter

  - Although the LoD increases the adaptiveness of a software system, it may result in having to write many wrapper methods (or using some *Facade* design pattern) to propagate calls to components; in some cases, this can add noticeable time and space overhead.
  - So it depends, but *Wrapper* / *Facade* is usually good enough to do. See video https://www.youtube.com/watch?v=FyJhALHmFXU. By using this you are also hiding information (unnecessary details).

Very important and always true things

- Everything begins and ends with a requirement. Business is the only thing that matters. We don't develop software for ourselves, but for clients to solve actual existing problems.
- Requirements are in constant flux. (The system is never done.)
- Estimates are always wrong.

Abstractions and requirements

- It is better to implement code that does not use a lot of abstraction, and to focus on a problem, when all requirements are specified fully and clearly. Otherwise, abstraction is a good and relatively cheap thing to do.
- Solving a specific problem is easier than general problems. General problem is much more difficult.
- **Solving a specific problem is easier if you know what specific problem is.** But in software, we usually don't fully know what a specific problem is. So from economic point of view, the best thing to do is to stay flexible.

Duck Typing

- This is about determining a suitability of an **object based on what it does rather than what it is.**
- It is like Turing Test. Just an analogy. If it behaves as a human, then the question whether it is actual human of not, does not matter.
- Practically, in dynamically typed languages, you are not specifying types. So we could say that Duck Typing is like polymorphism without any hierarchy. So, according to video https://www.youtube.com/watch?v=oaIxRQSAZXE, it is said, that we should not check types, but we should check the capability of a given object.

Common Anti-patterns

(during SW development[2])

- Create tons of functions
- Use one liner as much as possible
- Use recursion
- Extensive use of comments
- Adding code you may need but never will
- Lots of variables the more the better
- Start to refactor the code
- Using an interface just to forcing the creation of a method. This is a bad idea. Avoid interfaces that just force action.

Data Clamp

- It is a code smell (that something is maybe wrong, not that definitely is - that is anti-pattern).
- It is when more than 1 piece of data are found together. For example start date and end date. And you find these two in your application together very often. Maybe it is better to create an object from them, or use data range. But this totally depends on your application and data.
- To resolve these, we may have, for instance, less arguments to our functions, which is a good thing.

UML

- $\rightarrow$ is 'has-a' (composition) ... so it is basically "using"
- $\twoheadrightarrow$ is 'is-a' (inheritance)

Professional Programmer

- *"Do; or do not. There is no trying."* - Yoda
- *"Turning pro is a mindset. If we are struggling with fear, self-sabotage, procrastination, self-doubt, etc., the problem is, we're thinking like amateurs. Amateurs don't show up. Amateurs crap out. Amateurs let adversity defeat them. The pro thinks differently. He shows up, he does his work, he keeps on truckin', no matter what.", Steven Pressfield*

[2] https://www.youtube.com/watch?v=MTCYhbfSAuA&list=UU4xKdmAXFh4ACyhpiQ_3qBw&index=57

- *"How you do anything is how you do everything."*, T. Harv Eker. If you lower your standards in one area, you'll inadvertently find them dropping in other areas as well. Once you've crossed the line of compromise, it can be difficult to go back.
- Marketing is a multiplier for talent. The better marketing you have, the more it magnifies your talent.
- *"If you help enough people get what they want, you will get what you want.",* Zig Ziglar. This is the primary strategy that you should use in marketing yourself. It will be more effective than any other technique.
- Attitudes, disciplines, and actions. Taking responsibility. When a professional makes a mistake, he cleans up the mess. That feeling is the essence of professionalism. Because, you see, professionalism is all about taking responsibility.
- The first thing you must practice is apologizing. Apologies are necessary, but insufficient. You cannot simply keep making the same errors over and over. As you mature in your profession, your error rate should rapidly decrease towards the asymptote of zero. It won't ever get to zero, but it is your responsibility to get as close as possible to it.
- Every time QA, or worse a user, finds a problem, you should be surprised, chagrined, and determined to prevent it from happening again.
- Every single line of code that you write should be tested. Period. However, 100% is an asymptote. But isn't some code hard to test? Yes, but only because that code has been designed to be hard to test. The solution to that is to design your code to be easy to test. And the best way to do that is to write your tests first, before you write the code that passes them (TDD).
- The fundamental assumption underlying all software projects is that software is easy to change. If you violate this assumption by creating inflexible structures, then you undercut the economic model that the entire industry is based on.
- Why do most developers fear to make continuous changes to their code? They are afraid they'll break it! Why are they afraid they'll break it? Because they don't have tests. It all comes back to the tests. If you have an automated suite of tests that covers virtually 100% of the code, and if that suite of tests can be executed quickly on a whim, then you simply will not be afraid to change the code.
- Professionals spend time caring for their profession. You should plan on working 60 hours per week. The first 40 are for your employer. These 40 hours should be spent on your employer's problems, not on your problems. The remaining 20 are for you. During this remaining 20 hours you should be reading, practicing, learning, and otherwise enhancing your career.

- Perhaps you think this is a recipe for burnout. On the contrary, it is a recipe to avoid burnout. Presumably you became a software developer because you are passionate about software and your desire to be a professional is motivated by that passion. During that 20 hours you should be doing those things that reinforce that passion. Those 20 hours should be fun!
- Here is a minimal list of the things that every software professional should be conversant with: Design patterns. You ought to be able to describe all 24 patterns in the GOF book and have a working knowledge of many of the patterns in the POSA books.
- Design principles. You should know the SOLID principles and have a good understanding of the component principles.
- Methods. You should understand XP, Scrum, Lean, Kanban, Waterfall, Structured Analysis, and Structured Design.
- Disciplines. You should practice TDD, Object-Oriented design, Structured Programming, Continuous Integration, and Pair Programming.
- Artifacts: You should know how to use: UML, DFDs, Structure Charts, Petri Nets, State Transition Diagrams and Tables, flow charts, and decision tables.
- Hackerrank - small problems / challenges every day 10 minutes for example.
- Know your domain. It is the responsibility of every software professional to understand the domain of the solutions they are programming. When starting a project in a new domain, read a book or two on the topic. Interview your customer and users about the foundation and basics of the domain. Spend some time with the experts, and try to understand their principles and values. It is the worst kind of unprofessional behavior to simply code from a spec without understanding why that spec makes sense to the business.
- Your employer's problems are your problems. You need to understand what those problems are and work toward the best solutions. As you develop a system you need to put yourself in your employer's shoes and make sure that the features you are developing are really going to address your employer's needs.
- Professionals speak truth to power. Professionals have the courage to say no to their managers.
- When your manager tells you that the login page has to be ready by tomorrow, he is pursuing and defending one of his objectives. He's doing his job. If you know full well that getting the login page done by tomorrow is impossible, then you are not doing your job if you say "OK, I'll try." The only way to do your job, at that point, is to say "No, that's impossible." The best possible outcome is the goal that you and your manager share. The trick is to find that goal, and that usually takes negotiation.

- The "fact" that it will take longer is much more important then "why". Providing too much detail can be an invitation for micro-management.
- The most important time to say no is when the stakes are highest. The higher the stakes, the more valuable no becomes.
- By promising to try you are committing to succeed. This puts the burden on you. If your "trying" does not lead to the desired outcome, you will have failed.
- Professionals are often heroes, but not because they try to be. Professionals become heroes when they get a job done well, on time, and on budget. By trying to become the man of the hour, the savior of the day, such acting is not like a professional. The temptation to be a hero and "solve the problem" is huge. What we all have to realize is that saying yes to dropping our professional disciplines is not the way to solve problems. Dropping those disciplines is the way you create problems.
- Programming is so hard, in fact, that it is beyond the capability of one person to do it well. No matter how skilled you are, you will certainly benefit from another programmer's thoughts and ideas.
- It is a matter of professional ethics for senior programmers to spend time taking younger programmers under their wing and mentoring them.
- The fact that some programmers do wait for builds is tragic and indicative of carelessness. In today's world build times should be measured in seconds, not minutes, and certainly not hours.
- In one way or another, all professionals practice. They do this because they care about doing the best job they possibly can. What's more, they practice on their own time because they realize that it is their responsibility (and not their employer's) to keep their skills sharp. Practicing is what you do when you aren't getting paid. You do it so that you will be paid, and paid well.
- Despite the fact that your company may have a separate QA group to test the software, it should be the goal of the development group that QA find nothing wrong.
- When professionals make commitments, they provide hard numbers, and then they make those numbers. However, in most cases professionals do not make such commitments. Rather, they provide probabilistic estimates that describe the expected completion time and the likely variance.
- Professionals realize that "quick and dirty" is an oxymoron. Dirty always means slow.
- Choose disciplines that you feel comfortable following in a crisis. Then follow them all the time. Following these disciplines is the best way to avoid getting into

a crisis. If you follow the discipline of Test-Driven Development in non-crisis times but abandon it during a crisis, then you don't really trust that TDD is helpful. If you keep your code clean during normal times but make messes in a crisis, then you don't really believe that messes slow you down.

- Communicate, and avoid surprises. Nothing makes people more angry and less rational than surprises. Surprises multiply the pressure by ten.
- Most managers want good code, even when they are obsessing about the schedule. They may defend the schedule and requirements with passion; but that's their job. **It's your job to defend the code with equal passion.** So it is unprofessional for programmers to bend to the will of managers who don't understand the risks of making messes.
- **The Flow Zone**
  - This is a hyper-productive state, highly focused, tunnel-vision state of consciousness that programmers can get into while they write code. In this state they feel productive. In this state they feel infallible. And so they desire to attain that state, and often measure their self-worth by how much time they can spend there.
  - However, Robert C. Martin recommends to avoid the zone. According to him, the problem is that you lose some of the big picture while you are in the Zone, so you will likely make decisions that you will later have to go back and reverse. Code written in the Zone may come out faster, but you'll be going back to visit it more.
  - Also for him, he realized that he simply don't code well while listening to music. The music does not help him focus. He suspects, that what's really happening is that the music is helping programmers to enter the Zone.
- **Interruptions**
  - Pairing can be very helpful as a way to deal with interruptions. Your pair partner can hold the context of the problem at hand, while you deal with a whatever interrupted you. When you return to your pair partner, he quickly helps you reconstruct the mental context you had before the interruption.
  - TDD is another big help. If you have a failing test, that test holds the context of where you are. You can return to it after an interruption and continue to make that failing test pass.
  - Of course, there will be interruptions that distract you and cause you to lose time. When they happen, remember that next time you may be the one who needs to interrupt someone else. So the professional attitude is a polite willingness to be helpful.
- **Writer's Block**

  - Sometimes the code just doesn't come. I've had this happen to me and I've seen it happen to others. You sit at your workstation and nothing happens.
  - Often you will find other work to do. You'll read email. You'll read tweets. You'll look through books, or schedules, or documents. You'll call meetings. You'll start up conversations with others. You'll do anything so that you don't have to face that workstation and watch as the code refuses to appear.
  - The causes of this can be various. For example, not getting enough sleep - this is perhaps one of the biggest obstacles. Others are worry, fear, or depression.
  - Possible solution: Find a pair partner. There is a physiological change that takes place when you work with someone.
  - Creative output depends on creative input. Read science fiction for example. Or astronomy, physics, chemistry, or mathematics. While being actively stimulated by challenging and creative ideas, results in an almost irresistible pressure to create something myself. Not all forms of creative input work for me. Watching TV does not usually help me to create.
  - Software development is a marathon, not a sprint. A marathon runner takes care of her body both before and during the race. Professional programmers conserve their energy and creativity with the same care.
  - Can't go home till you solve this problem? Oh yes you can, and you probably should! Creativity and intelligence are fleeting states of mind. When you are tired, they go away. If you then pound your non-functioning brain for hour after late-night hour trying to solve a problem, you'll simply make yourself more tired and reduce the chance that the shower, or the car, will help you solve the problem.
  - When you are working on a problem, you sometimes get so close to it that you can't see all the options. You miss elegant solutions because the creative part of your mind is suppressed by the intensity of your focus. Sometimes the best way to solve a problem is to go home, eat dinner, watch TV, go to bed, and then wake up the next morning and take a shower.

- **Being Late**
  - You will be late. No matter how professional you are.
  - Regularly measure your progress against your goal, and come up with three fact-based end dates: best case, nominal case, and worst case. Be as honest as you can about all three dates. Do not incorporate hope into your estimates! Present all three numbers to your team and stakeholders. Update these numbers daily.

- **Hope**
  - Hope is the project killer. Hope destroys schedules and ruins reputations. Hope will get you into deep trouble. If the trade show is in ten days, and

your nominal estimate is 12, you are not going to make it. Make sure that the team and the stakeholders understand the situation, and don't let up until there is a fall-back plan. Don't let anyone else have hope.

- There is no way to rush. You can't make yourself code faster. You can't make yourself solve problems faster. If you try, you'll just slow yourself down and make a mess that slows everyone else down, too.

- **Overtime**
  - Overtime can work, and sometimes it is necessary. Sometimes you can make an otherwise impossible date by putting in some ten-hour days, and a Saturday or two. But this is very risky. You are not likely to get 20% more work done by working 20% more hours. What's more, overtime will certainly fail if it goes on for more than two or three weeks
  - You should not agree to work overtime unless (1) you can personally afford it, (2) it is short term, two weeks or less, and (3) your boss has a fall-back plan in case the overtime effort fails.

- **Meetings**
  - There are 2 truths about meetings. Often these two truths equally describe the same meeting.
    1. Meetings are necessary.
    2. Meetings are huge time wasters.
  - Professionals actively resist attending meetings that don't have an immediate and significant benefit. Even if you attend a meeting and it is boring, you should consider leaving. You can simply ask, at an opportune moment, if your presence is still necessary. You can explain that you can't afford a lot more time, and ask whether there is a way to expedite the discussion or shuffle the agenda. The important thing to realize is that remaining in a meeting that has become a waste of time for you, and to which you can no longer significantly contribute, is unprofessional. You have an obligation to wisely spend your employer's time and money, so it is not unprofessional to choose an appropriate moment to negotiate your exit.

- **Estimates**
  - The problem is that we view estimates in different ways. Business likes to view estimates as commitments. Developers like to view estimates as guesses. The difference is profound.
  - Professionals don't make commitments unless they know they can achieve them. It's really as simple as that. If you are asked to commit to something that you aren't certain you can do, then you are honor bound to decline. If you are asked to commit to a date that you know you can achieve, but would require long hours, weekends, and skipped family vacations, then the choice

is yours; but you'd better be willing to do what it takes. Commitment is about certainty. Other people are going to accept your commitments and make plans based upon them.

- An estimate is a guess. No commitment is implied. No promise is made. Missing an estimate is not in any way dishonorable. The reason we make estimates is because we don't know how long something will take.
- An estimate is not a number. An estimate is a distribution.

PERT (Program Evaluation and Review Technique)

- It was created in 1957, to support the U.S. Navy's Polaris submarine project. One of the elements of PERT is the way that estimates are calculated. The scheme provides a very simple, but very effective way to convert estimates into probability distributions suitable for managers.
- When you estimate a task, you provide three numbers. This is called trivariate analysis:
  - *Optimistic Estimate.* This number is wildly optimistic. You could only get the task done this quickly if absolutely everything went right. Indeed, in order for the math to work this number should have much less than a 1% chance of occurrence.
  - *Nominal Estimate.* This is the estimate with the greatest chance of success.
  - *Pessimistic Estimate.* Once again this is wildly pessimistic. It should include everything except hurricanes, nuclear war, stray black holes, and other catastrophes. Again, the math only works if this number has much less than a 1% chance of success.
- Given these 3 estimates, we can describe the probability distribution as follows: $\mu = \frac{O+4N+P}{6}$, where μ is the expected duration of the task. For most tasks this will be a somewhat pessimistic number because the right-hand tail of the distribution is longer than the left-hand tail.
- sigma is the standard deviation: $s = \frac{P-O}{6}$, and it is a measure of how uncertain the task is.
- You often do not have just 1 task, but a sequence of tasks. You simply sum all μ and sigmas.
- If you are a programmer of more than a few years' experience, you've likely seen projects that were estimated optimistically, and that took three to five times longer than hoped. The simple PERT scheme just shown is one reasonable way to help prevent setting optimistic expectations.

Wideband Delphi

- Created in 1970s and it is an estimation technique. There are many variations, but all have the same goal - consensus.

- The strategy is simple. A team of people assemble, discuss a task, estimate the task, and iterate the discussion and estimation until they reach agreement.
- There are more approaches - Flying Fingers, Planning Poker, or Affinity Estimation.

1.1 Programming Paradigms

- In 1938, Alan Turing laid the foundations of what was to become computer programming.
- By 1945, Turing was writing real programs on real computers in binary language.
- Assembly language came in 1940s, Fortran in 1953, and then a lot of others.
- Each of the paradigms removes capabilities from the programmer. None of them adds new capabilities. Each imposes some kind of extra discipline that is negative in its intent. The paradigms tell us what not to do, more than they tell us what to do. Another way to look at this issue is to recognize that each paradigm takes something away from us.
- We use polymorphism as the mechanism to cross architectural boundaries; we use functional programming to impose discipline on the location of and access to data; and we use structured programming as the algorithmic foundation of our modules. Notice how well those three align with the 3 big concerns of architecture: function, separation of components, and data management.

Functional Programming

- In many ways, the concepts of functional programming predate programming itself. It started with Alonzo Church, who in 1936 invented l-calculus while pursuing the same mathematical problem that was motivating Alan Turing at the same time. His l-calculus is the foundation of the LISP language, invented in 1958 by John McCarthy. Functional programming imposes discipline upon assignment.
- **Variables** in functional languages do not vary **(all are immutable)**. All race conditions, deadlock conditions, and concurrent update problems are due to mutable variables. You cannot have a race condition or a concurrent update problem if no variable is ever updated. You cannot have deadlocks without mutable locks.
- In other words, all the problems that we face in concurrent applications - all the problems we face in applications that require multiple threads, and multiple processors - cannot happen if there are no mutable variables.
- The question you must be asking yourself, then, is whether immutability is practicable. The answer to that question is affirmative, if you have infinite storage and infinite processor speed. Lacking those infinite resources, the answer is a bit more nuanced. Yes, immutability can be practicable, if certain compromises are made.
- The limits of storage and processing power have been rapidly receding from view. Nowadays it is common for processors to execute billions of instructions per second and to have billions of bytes of RAM. The more memory we have, and the faster our machines are, the less we need mutable state.

Structured Programming

- Discovered by Edsger Wybe Dijkstra in 1968. Structured programming imposes discipline on direct transfer of control.
- Böhm and Jacopini proved, that all programs can be constructed from just three structures: sequence, selection, and iteration. This discovery was remarkable: The very control structures that made a module provable were the same minimum set of control structures from which all programs can be built. Thus structured programming was born.
- Science is fundamentally different from mathematics, in that scientific theories and laws cannot be proven correct. I cannot prove to you that Newton's second law of motion, F = ma, or law of gravity, are correct. I can demonstrate these laws to you, and I can make measurements that show them correct to many decimal places, but I cannot prove them in the sense of a mathematical proof. No matter how many experiments I conduct or how much empirical evidence I gather, there is always the chance that some experiment will show that those laws of motion and gravity are incorrect. That is the nature of scientific theories and laws: They are falsifiable but not provable. Science does not work by proving statements true, but rather by proving statements false. Those statements that we cannot prove false, after much effort, we deem to be true enough for our purposes. Ultimately, we can say that mathematics is the discipline of proving provable statements true. Science, in contrast, is the discipline of proving provable statements false.
- Dijkstra once said, "Testing shows the presence, not the absence, of bugs." In other words, a program can be proven incorrect by a test, but it cannot be proven correct. All that tests can do, after sufficient testing effort, is allow us to deem a program to be correct enough for our purposes. The implications of this fact are stunning. Software development is not a mathematical endeavor, even though it seems to manipulate mathematical constructs. Rather, software is like a science. We show correctness by failing to prove incorrectness, despite our best efforts.
- Structured programming forces us to recursively decompose a program into a set of small provable functions. We can then use tests to try to prove those small provable functions incorrect. If such tests fail to prove incorrectness, then we deem the functions to be correct enough for our purposes.
- It is this ability to create falsifiable units of programming that makes structured programming valuable today. This is the reason that modern languages do not typically support unrestrained goto statements. Moreover, at the architectural level, this is why we still consider functional decomposition to be one of our best practices.
- At every level, from the smallest function to the largest component, software is like a science and, therefore, is driven by falsifiability. Software architects strive

to define modules, components, and services that are easily falsifiable (testable). To do so, they employ restrictive disciplines similar to structured programming, albeit at a much higher level.

- Structured programming allows modules to be recursively decomposed into provable units, which in turn means that modules can be functionally decomposed. That is, you can take a large-scale problem statement and decompose it into high-level functions. Each of those functions can then be decomposed into lower-level functions, ad infinitum. Moreover, each of those decomposed functions can be represented using the restricted control structures of structured programming.

Object-Oriented Programming

- Discovered in 1966, by Ole Johan Dahl and Kristen Nygaard. They moved the function call stack frame to the heap and invented OO. Object-oriented programming imposes discipline on indirect transfer of control.
- What is OO? The combination of data and function. A way to model the real world. The nature of OO is **encapsulation, inheritance,** and **polymorphism**.
- OO is the ability, through the use of polymorphism, to gain absolute control over every source code dependency in the system. It allows the architect to create a plugin architecture, in which modules that contain high-level policies are independent of modules that contain low-level details. The low-level details are relegated to plugin modules that can be deployed and developed independently from the modules that contain high-level policies.
- OO languages provide easy and effective **encapsulation** of data and function. As a result, a line can be drawn around a cohesive set of data and functions. Outside of that line, the data is hidden and only some of the functions are known. We see this concept in action as the private data members and the public member functions of a class.
- **Inheritance** is simply the re-declaration of a group of variables and functions within an enclosing scope. This is something C programmers were able to do manually long before there was an OO language (it's a simple trick, that is +- used in single inheritance in C++).
- Did we have **polymorphic behavior** before OO languages? Of course we did. The bottom line is that polymorphism is an application of pointers to functions. But this is dangerous, you have to remember the conventions (such as initialization of a pointer and so on). Using an OO language makes polymorphism trivial and eliminates these dangers.
- **Dependency Inversion**

  - Source code dependency (the inheritance relationship) between module M and the interface I points in the opposite direction compared to the flow of control.
  - The fact that OO languages provide safe and convenient polymorphism means that any source code dependency, no matter where it is, can be inverted (thanks to interfaces!).
  - With this approach, software architects working in systems written in OO languages have absolute control over the direction of all source code dependencies in the system. They are not constrained to align those dependencies with the flow of control. No matter which module does the calling and which module is called, the software architect can point the source code dependency in either direction.
  - **That is power! That is the power that OO provides. That's what OO is really all about—at least from the architect's point of view.**
  - As an example, you can rearrange the source code dependencies of your system so that the database and the user interface (UI) depend on the business rules, rather than the other way around. This means that the UI and the database can be plugins to the business rules. It means that the source code of the business rules never mentions the UI or the database.
  - In short, when the source code in a component changes, only that component needs to be redeployed. This is independent deployability. If the modules in your system can be deployed independently, then they can be developed independently by different teams. That's independent developability.
  - **So, OO is the ability, through the use of polymorphism, to gain absolute control over every source code dependency in the system.**

- See SOLID principles for writing clean OOP code that make it easy for a programmer to develop software that is easy to maintain and extend.

1.2 Software Development Paradigms and Methodologies

- Software development is the process of conceiving, specifying, designing, programming, documenting, testing, and bug fixing involved in creating and maintaining applications, frameworks, or other software components.[3]
- In bug report, there should be at least:
  - how to reproduce the bug
  - what should have happened
  - what actually happened

Paradigms and models

Waterfall Model

- Before agile, but it is still used in some applications.
- *Requirements -> Design -> Development -> Testing -> Deployment* (and big outcome at the end).
- In Waterfall, the next phase typically cannot be started until the previous one has been completed. The goal is to gather and analyze all the detailed requirements early in the process so that a complete solution can be architect-ed and build with highly predictable results.[4]
- Waterfall development can work well for complex or mission-critical systems or for and for organizations that require the highest levels of fault tolerance (such as the military or aerospace). However, projects using Waterfall processes take too long, in many cases months or years, to produce results that can be verified by the user and often and lacks the flexibility for today's environment.

Agile

- Like Waterfall model, but iterative - with cumulative outcomes.
- Agile is all about working collaboratively with people who have different skills and mindsets to achieve a common goal.[5]
- **Manifesto**[6]:
  - **Individuals and interactions** over processes and tools.
  - **Working software** over comprehensive documentation.

[3] https://en.wikipedia.org/wiki/Software_development
[4] https://theagileblueprint.wordpress.com/2011/03/02/comparing-waterfall-and-rational-unified-process/
[5] https://www.testingexcellence.com/there-is-no-qa-team-in-agile/
[6] https://agilemanifesto.org/

  - **Customer collaboration** over contract negotiation.
  - **Responding to change** over following a plan.

  That is, while there is value in the items on the right (not bold text), we value the items on the left more (bold text).

- The most used practices in agile development are Scrum, Kanban, XP, and a lot of others, see the next subsections below.

Methodologies and Frameworks

DevOps

- It is a set of practices that combines software development (Dev) and information-technology operations (Ops) which aims to shorten the systems development life cycle and provide continuous delivery with high software quality.

Lean

- Lean development can be summarized by 7 principles, very close in concept to lean manufacturing principles:
  - Eliminate waste
  - Amplify learning
  - Decide as late as possible
  - Deliver as fast as possible
  - Empower the team
  - Build integrity in
  - Optimize the whole

Kanban

- It is a lean method to manage and improve work across human systems.
- Work items are visualized to give participants a view of progress and process, from start to finish—usually via a Kanban board.
- Kanban is commonly used in software development in combination with other methods and frameworks such as Scrum.

Rational Unified Process (RUP)

- It is an iterative software development process framework. It is use-case driven, architecture-centric, and incremental and iterative.
- The RUP has determined a project life-cycle consisting of four phases:[7]

[7] https://techterms.com/definition/rup

  - *Inception phase.* The idea for the project is stated. The development team determines if the project is worth pursuing and what resources will be needed.
  - *Elaboration phase.* The project's architecture and required resources are further evaluated. Developers consider possible applications of the software and costs associated with the development.
  - *Construction phase.* The project is developed and completed. The software is designed, written, and tested.
  - *Transition phase.* The software is released to the public. Final adjustments or updates are made based on feedback from end users.
- Iterations occur in each phase. Activities in iterations are focused on one of the four activities: gathering requirements, analyzing, designing, implementing, and testing. Each of these activities place a more or less important role as the project moves from phase to phase.

Scrum

- **One of the goals if to create a self-organizing unit (team) so that the team is able and willing to undertake responsibility for its work.**
- Scrum master is not a team assistant. He is not responsible for product delivery or results. He is not leader of a team, does not estimate priorities, does not initiate discussions. Team does not communicate with product owner through scrum master.
- Scrum master supports self-organization of a team. He helps team with obstacles, supports to estimate common goal(s) and being team more effective. He organizes meetings, help a company to fulfill long-term goals and strategies. In case that there is a need for changing a process, he is an initiator of such change. He is constantly educating himself and the others.
- Scrum has a sprint (usually 2 weeks), and during that, there are multiple meetings:
  - *Inbox meeting* - meeting about potential US/BUGs from INBOX (or even IDEASBOX), but only urgent/high priorities. No technical discussions. USs should have short descriptions and if everything is clear, after this meeting, they will be put to BACKLOG. Questions:
    * Is everything clear to developer?
    * Do a developer knows what to do (not how - that belongs to Backlog meeting)?
  - *Backlog meeting* - meeting about prepared US/BUGs from INBOX, status PLANNABLE. Technical discussions about how to do it. Pre-defined tasks of US:
    * Development

    * Discussion
    * Testing
    * Documentation
    * Code+documentation review
    * QA testing
  - *Retrospective meeting* - feelings and thoughts about the last sprint.
  - *Daily stand-up* - daily status of everyone in team.

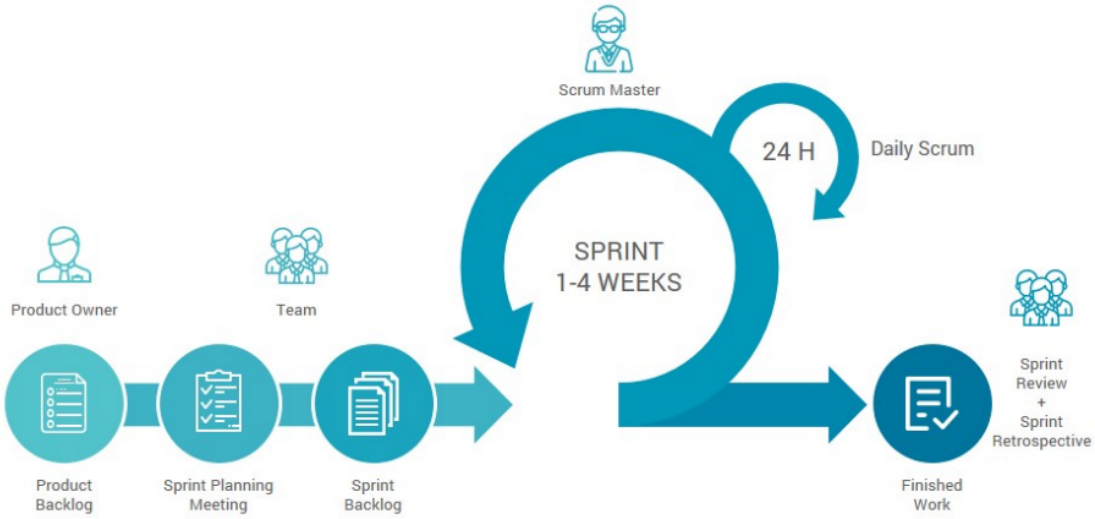

Figure 1.1: Scrum

Extreme Programming (XP)

- It is intended to improve software quality and responsiveness to changing customer requirements. XP is a way to improve your development process.
- As a type of agile software development, it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.
- It is a philosophy of software development based on the values of communication, feedback, simplicity, courage, and respect. Good relationships lead to good business. If members of a team don't' care about each other and what they are doing, XP won't work.

- Do your best and then deal with the consequences. That's extreme – you leave yourself exposed.
- **XP is lightweight – you only do what you need to do to create value for the customer.**
- XP can work with teams of any size. The practices need to be augmented and altered when many people are involved.
- XP adapts to vague or rapidly changing requirements. XP shines in this area compared to other techniques.
- Other elements of extreme programming include:
  - programming in pairs or doing extensive code review,
  - unit testing of all code,
  - avoiding programming of features until they are actually needed,
  - a flat management structure,
  - code simplicity and clarity,
  - expecting changes in the customer's requirements as time passes and the problem is better understood,
  - frequent communication with the customer and among programmers,
  - XP always keeps the system in a deployable condition, problems are not allowed to accumulate,
  - XP tests from the perspective of programmers writing tests function by function, and feature by feature,
  - values are universal - my values at work are exactly the same as my values in the rest of my life,
  - incremental design - invest in the design of the system every day. Your design improves as your understanding of the project improves. The most effective time to design is in the light of experience. Refactoring a design becomes less problematic and stressful as you continue to apply it.
  - shared code - collective responsibility, plus anyone on the team can improve any part of the system at any time,
  - daily deployment is a goal - rapid Deployment is a step in the right direction,
  - energized work - overly long work hours lead to reduced efficiency and can even remove value from a project.
- The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels.
- As an example, code reviews are considered a beneficial practice; taken to the extreme, code can be reviewed continuously, i.e. the practice of pair programming.

OKR

- This is especially great for smaller teams.

Practices

CI, CD

- See the next section.

Pair Programming (PP)

- Two programmers work together at one workstation. One, the driver, writes code while the other, the observer or navigator, reviews each line of code as it is typed in. The two programmers switch roles frequently.

Test-Driven Development (TDD)

- See chapter about testing.

Behavior-Driven development (BDD)

- It is an Agile software development process that encourages collaboration among developers, QA and non-technical or business participants in a software project.
- It encourages teams to use conversation and concrete examples to formalize a shared understanding of how the application should behave.
- It emerged from test-driven development (TDD). Behavior-driven development combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design to provide software development and management teams with shared tools and a shared process to collaborate on software development.
- Behavior-driven development specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit (requirements set by the business).
- It encourages collaboration between developers, QA and non-technical or business participants in a software project.

1.3 Continuous Integration/Delivery/Deployment

- **Continuous Integration** - merge high-quality code ASAP. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.
- **Continuous Delivery** - producing software in short cycles, ensuring that the software can be reliably released at any time and, when releasing the software, doing so manually. It aims at building, testing, and releasing software with greater speed and frequency.
- **Continuous Deployment** - it goes one step further than continuous delivery. With this practice, every change that passes all stages of your production pipeline is released on production. There's no human intervention, and only a failed test will prevent a new change to be deployed to production.
- **Why CI/CD?**
  - Prevent regressions (covered by tests).
  - Find issues ASAP (early, low-cost feedback to devs).
  - Save precious time of devs, automate testing and let machines do the jobs.
  - Avoiding human mistakes (people may forgot to test something).
  - Avoid merging broken code (everything merged must be tested and deployable).
  - It forces devs to write tests.

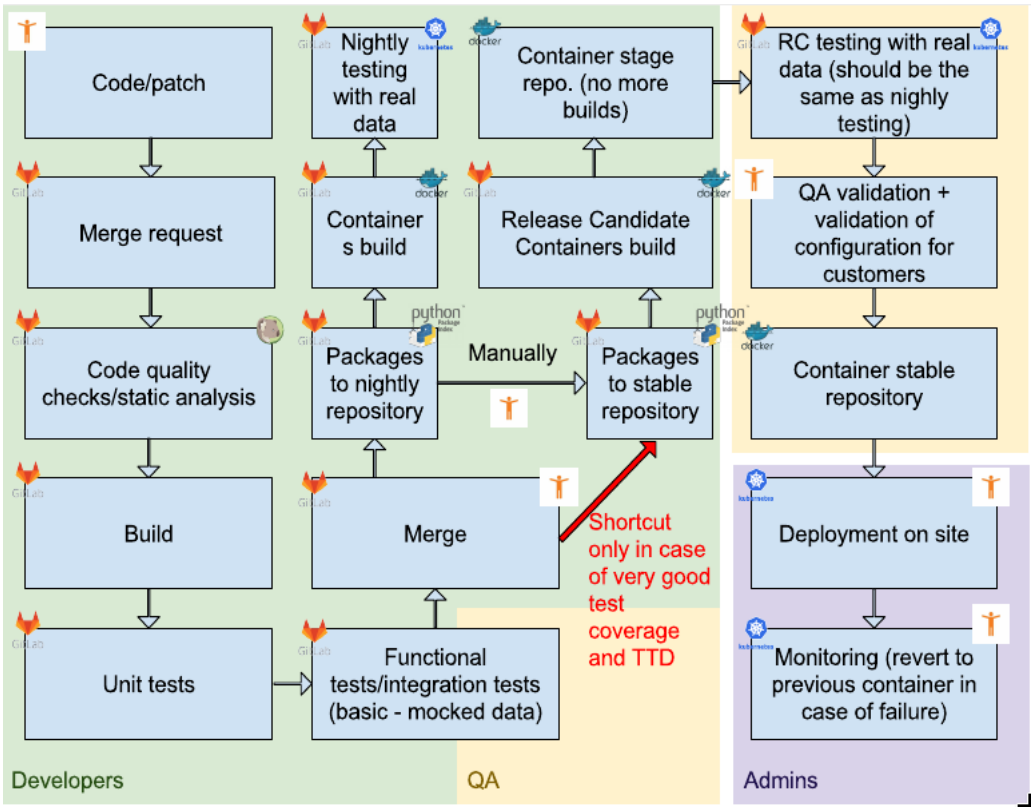

Figure 1.2: Ideal world how CI/CD should work. Many companies are migrating to this model. (TTD means test-driven development, typo.)

- **CI/CD differences**
  - Martin Fowler, who first wrote about Continuous Integration together with Kent Beck, describes CI as follows: *"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly."*
  - CI stands for continuous integration, whereas CD is often used interchangeably to signify "continuous delivery" and "continuous deployment." Are they both the same thing? No. Do they have a common goal? Yes.[8]
  - CI involves a series of steps that are automatically performed to integrate code from multiple sources, create a build and test. Each time a build or a set of

[8] https://blog.codeship.com/whats-the-difference-between-continuous-delivery-vs-continuous-deployment/

code passes the tests, it's automatically deployed out to a staging environment where further testing such as load testing and manual exploratory testing is conducted. This process can be repeated for days depending upon the project delivery requirements.

- Continuous delivery helps you build a refined version of the software by continuously implementing fixes and feedback until finally, you decide to push it out to production. In other words, continuous delivery involves human decision-making around what to release to the customers, and when.
- Continuous delivery is a series of practices designed to ensure that code can be rapidly and safely deployed to production by delivering every change to a production-like environment and ensuring business applications and services function as expected through rigorous automated testing. It doesn't mean every change is deployed to production ASAP.
  - *Unit test -> Platform Test -> Deliver to Staging -> Application Acceptance Tests -> Deploy to Production -> Post deploy Tests*
  - All previous steps are performed automatically in Continuous Deployment; in Continuous Delivery, *Application Acceptance Tests -> Deploy to Production* step is done manually.
- Continuous deployment - every change goes through an automated pipeline and a working version of the application is automatically pushed to production. It usually involves a production-like staging area with a mandatory time lag in the final release. This lag involves reviewing and manually accepting the changes in the code before releasing it to production. In contrast, continuous deployment does not require a staging area for code changes to be manually reviewed and verified. This is because automated testing is integrated early in the development process and continues throughout all the phases of the release.
- There are some exceptions where the concepts of delivery and deployment aren'=;t as relevant as they are elsewhere; for instance, if you've contributed to a library or created an artifact, you are unlikely to deploy it on a running system. In other words, there is no deployment phase. You simply push your code into a repository for other applications to consume.
- Jenkins and Ansible are popular automation tools for CI/CD in the market. To automate deployments, it is needed to manage:
  - Application packaging
  - Release versioning
  - Database updates
  - Server configuration management
  - Calendaring

    - Roll-forward and rollback
    - Security access
    - Auditing
  - Continuous delivery, when rightly coupled with continuous deployment, strengthens the foundation of a DevOps pipeline and is core to agile DevOps initiatives.

- **Container is a running image. There can be several containers from 1 image.**
- **Possible ideal scenario?**
  1. Dev creates a patch.
  2. He creates a new MR.
  3. Gitlab CI pipeline fires up.
     a) Static code analysis.
     b) Unit tests.
     c) Integration tests.
     d) Docker image is built.
     e) System tests with other services.
     f) Docker image pushed to internal registry.
  4. When the previous step is successful, dev/reviewer manually plays with container in dev environment. If everything went well, MR is approved and merged.
     - Container rebuild/mark container image as RC version (this depends on merging strategy).
     - Kubernetes deploy a new RC image to staging environment (triggered by devs or automatically).
  5. Container is monitored in staging environment.
     - Container or a feature is tested and monitored in staging environment by devs/QEs.
     - If bug is found, must be fixed and new image is deployed to stage environment.
     - If critical malfunction happen, container can be even automatically reverted to previous version by Kubernetes.
     - If image works as expected, it is marked as stable.
  6. (Docker) Image is ready for production.
     - Image is labeled by devs as production version.

  - Admins get notification from developers/QEs that a particular image is ready to be deployed.
  - Admins press a "magic button" in docker management system on the top of Kubernetes.
    * If critical malfunction is detected by monitoring, image is automatically rollbacked to the latest working version.
    * Deployed image is monitored by monitoring system, if suspicious activity is happening (or revert is requested from devs), admins can simply revert to older version of image (magic revert button, no supercomplex downgrades of packages).

- **How to enable CI/CD?**
  - **GitLab** - selfhosted, specified in file **.gitlab-ci.yml.** Test environment is isolated, containers are destroyed after test; a new testing environment is always created. This CI pipeline may contain several stages - from fastest to slowest and more expensive tests (executed if previous stages passed).
  - **GitHub + Travis CI** - this CI may need an extra AWS instance for more complicated tests.
  - **Jenkins** - not the best one, many things must be implemented manually.
  - **Phabricator** - this CI is not that mature as in GitLab case.
- CD may require to containerize services first and this requires a big test coverage.
- Build system - for building "**build artifacts**" (.deb, pypi) - so that the most things are in packages, which will make deployment easier and less error prone. Alternative = **containers**, but they are not for everything.
- **Why containers?**
  - Containers are easy for deployment, they make CD easier.
  - Containers are Linux processes with:
    * constrained resources - cgroups
    * isolations - namespaces
    * security - Seccomp, Capabilities, SELinux

    Steps:
    1. build - *buildah* (run builds in an isolated container, run without root)
    2. run & develop locally - *podman*
    3. store/share - *skopeo* (run without root, move images between environments, inspect remote images)
    4. run in a production cluster - *CRI-O* (read-only container filesystem, user namespaces - soon in Kubernetes as well, enable fewer capabilities)

  - Provide isolation against outside influences like:
    * incompatible update of packages on OS
    * incompatible versions of packages, incompatible configurations
    * allows to mix various package versions per service
    * gives abstraction on service (no need of knowledge what is happening inside for deployment)
  - Provide consistency - the same image will be tested and deployed (no last moment surprises during deployment).
  - Containers are versioned, if critical issue is found on production, stop container and put back an older one.
  - Containers can be updated separately - lowering downtime, asynchronous releases.
  - Containers are technology of virtualization.
  - Popular technologies are Docker, Kubernetes, OpenShift, ...

Buildah + Ansible

Combo for building container images (+ansible plugins).

- *ansible-bender* - wraps the functionality around *Buildah + ansible_playbook*. If I want to see logs from previous builds for example. It has configurable layering and caching.
- Perhaps see blog.tomecek.net

Docker

- Docker utilizes Linux containers.
  - Linux containers, commonly referred to as LXC, originated in 2008, and they rely on the Linux kernel *cgroups*[9] functionality that originated in Linux kernel version 2.6.24.
  - Linux containers themselves are an operating system virtualization method that you can utilize to run multiple isolated Linux systems on a single host. They all utilize the kernel version that is running on the host on which the containers are running.
- Docker relies on using the host OS's Linux kernel for the OS it was built on. For this reason, you can have almost any Linux OS as your host operating system and be able to layer other OSes on top of the host. Another benefit of Docker is the

[9] cgroups is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

size of images when they are born. They do not contain the largest piece: the kernel or the operating system. This makes them incredibly small, compact, and easy to ship.

- Provides isolation of applications and their dependencies from host machine.
- It is not a VM, it shares kernel of host machine, but processes are very isolated from host.
- Images are small, minimal "Linux alpine" docker image has 5MB (the smallest VM image has much more).
- Once docker image is build, the same image will be tested, deployed to staging and then to production. Developers are responsible for proper dependencies installation and image build.
- CI tests in GitLab can be powered by Docker.
- Docker is likely to support Rootless mode in the future.
- When people say “Docker” they typically mean **Docker Engine**, the client-server application made up of the Docker daemon, a REST API that specifies interfaces for interacting with the daemon, and a command line interface (CLI) client that talks to the daemon (through the REST API wrapper). Docker Engine accepts docker commands from the CLI, such as *docker run <image>*.
- *Dockerd*
  - Docker build, it uses *dockerd* REST API to transfer build context as a tarball; image is built completely remotely by the engine, parses Dockerfile.
  - *Imagebuilder* - parses Dockerfile to determier images) using the Docker commands. The command that we will be looking at is docker search. With the docker search command build steps Source-to-image - uses one of many builder images. Uses docker client library to start builder image using a remote image and to commit the builder container using also remote engine. Library and CLI engine.
- Except Docker CLI, there are the following features:
  - **Docker registries** - there are 3 options to store docker images: Docker Hub, Docker Trusted Registry, and Docker Registry.
  - **Docker Machine**[10] - tool that you can utilize to set up and manage your Docker hosts. You can install and run Docker on Mac or Windows, provision and manage multiple remote Docker hosts, or provision Swarm clusters. If you want an efficient way to provision multiple Docker hosts on a network, in the cloud or even locally, you need Docker Machine.

[10] https://docs.docker.com/machine/overview/

- **Docker Compose** - another tool in the Docker ecosystem that can be used to create multiple containers with a single command. This allows you to spin up application stacks that may include some web servers, a database server, and/or file servers as well. Docker Compose utilizes a docker-compose.yml file to start up and configure all the containers that you have specified.
- **Docker Swarm** - it allows you to create and manage clustered Docker servers. Swarm can be used to disperse containers across multiple hosts. It also has the ability to scale containers as well. The installation for Docker Swarm actually launches a container that is used as the Swarm Manager master to communicate to all the nodes in a Swarm cluster.
- **Docker UCP** (Universal Control Plane) - is a solution for Docker that enables you to control various aspects of your Docker environment through a web interface.

Podman

- CLI tool for replacing Docker daemon in most use cases.
- Rootless storage under the user home dir. No CLI diff between root and user.
- Each container runs in its own user namespace.
- Podman pods - similar concept to Kubernetes pods. Group of containers that share resources and deploy as a single unit.

Kubernetes

- Created by Google. Orchestration tool for running docker containers. GitLab supports Kubernetes integration.
- Dynamically load-balance resources, migrates containers to less utilized servers.
- Makes deployment and management of containers easier.
- Configure, which containers should communicate between themselves at 1 place.
- Deploy everything on 1 place on multiple virtual machines.
- Scalability - distribute containers over multiple Vms, increase performance, deploy more containers.
- For a breaf vocabulary, there is a nice video[11]
- Parts:
  - *Master*: contains 1+ *Replication Controllers*. They communicate with *Pods*.

[11] https://www.youtube.com/watch?v=1xo-0gCVhTU

  - *Node*: *Kubelet* (it is an instance of a computer, it is application that is running); It runs Pods and communicates with Master.
  - *Pod*: Runs 1+ containers and exists on a *Node*.
  - *Service*: Handles requests and it is usually a load balancer.
  - *Deployment*: Defines desired state - Kubernetes handles the rest.

- *CRI-O - Container Runtime Interface* - client-server Kubernetes runtime for running containers in a cluster. It aims at replacing Docker and it's set to be the default runtime when running OpenShift. It looks heavily tested! For more information, see https://medium.com/cri-o.

- *Kubeflow* - ML + Kubernetes. Kuberflow aggregates all the tools like TF, Pytorch etc. Anywhere you run Kubernetes, you should be able to run Kuberflow.

- *Kubernetes Operators*
  - *Operator* is a controller service that actively manages the full lifecycle of an application on Kubernetes. A mature Operator can deploy, upgrade, backup, repair, scale, and reconfigure an application that it manages.
  - Automated software managers for Kubernetes clusters - application-specific controllers that extends Kubernetes API to create, configure, manage instances of complex stateful applications.
  - Operators are Kubernetes agents that know how to deploy, scale, manage, backup, and even upgrade complex, stateful applications.
  - They are good for: databases, file, block and object storage, applications with their own notion of a "cluster", apps for distribution on Kubernetes.
  - Nice tutorials: https://learn.openshift.com/operatorframework
  - There exist a lot of operators in the wild: https://github.com/operator-framework/awesome-operators

- Kubernetes vulnerabilities are of 3 types:
  - Cluster vulnerabilities - misconfiguration, general security primitives (bad network policies, admin role misuse, ...), pod spec files, data dir access and ownership rights.
  - Workload vulnerabilities - privileged containers/applications, vulnerable application code, missing seccomp profiles, unnecessary syscall capabilities.
  - Wrong tools for the job - 3rd party products / solutions not made for for containers. Unfortunately you have to rely on vendors.
  - See slides on https://devconfcz2019.sched.com/event/JcoQ/boost-your-security-and-resiliency-in-kubernetes

OpenShift

- Openshift is a family of containerization software developed by Red Hat.[12]
- It's based on top of Docker containers and the Kubernetes container orchestrator for enterprise application development and deployment.
- Openshift should have certificates. See *openshift-acme*[13]: letting it provision free certificates from Let's Encrypt (Openshift and Kubernetes cluster)
- *ci-operator*
  - This automates and simplifies the process of building and testing OpenShift component images.
  - Mainly intended to be run inside a *Pod* in a cluster, triggered by the *Prow CI* infrastructure, but it is also possible to run it as a CLI tool on a developer laptop.

Serverless + KNative

- Serverless
  - It can simplify the process of deploying code into production.
  - *Event -> Function -> Result*
  - History: AWS Lambda (2014)-> OpenWhisk (Fn project, Serverless Framework) -> Googld Cloud Functions (Azure Functions, Iron Functions, OpenFaaS) -> Oracle Fn (Funktion, Firebase Cloud Functions, Riff) -> Openshift Cloud Functions (KNative, Google loud Functions GA)
- KNative
  - It extends Kubernetes to provide a set of middleware components for building modern source-centric, container based apps that can run anywhere - premises, cloud, 3rd party data-center.[14]

[12] https://openshift.io/
[13] https://github.com/tnozicka/openshift-acme
[14] https://medium.com/@pkerrison/pizza-as-a-service-2-0-5085cd4c365e and https://blog.openshift.com/knative-serving-your-serverless-services/

1.4 Concurrency

- Concurrency can sometimes improve performance, but only when there is a lot of wait time that can be shared between multiple threads or multiple processors. Neither situation is trivial.
- The design of a concurrent algorithm can be remarkably different from the design of a single-threaded system. The decoupling of what from when usually has a huge effect on the structure of the system. Also, concurrency often requires a fundamental change in design strategy.
- Concurrency incurs some overhead, both in performance as well as writing additional code. Correct concurrency is complex, even for simple problems.
- Concurrency bugs aren't usually repeatable, so they are often ignored as one-offs instead of the true defects they are.
- Keep your concurrency-related code separate from other code. Attempt to partition data into independent subsets than can be operated on by independent threads, possibly in different processors.
- Get our non-threaded code working first. Do not try to chase down non-threading bugs and threading bugs at the same time. Make sure your code works outside of threads.
- Make your thread-based code especially pluggable and tunable, so that you can run it in various configurations.
- Run with more threads than processors. Things happen when the system switches between tasks. To encourage task swapping, run with more threads than processors or cores. The more frequently your tasks swap, the more likely you'll encounter code that is missing a critical section or causes deadlock.
- Run your threaded code on all target platforms early and often. Multi-threaded code behaves differently in different environments. You should run your tests in every potential deployment environment.
- Learn how to find regions of code that must be locked and lock them. Do not lock regions of code that do not need to be locked.
- Keep the amount of shared objects and the scope of the sharing as narrow as possible.

Basic definitions

Process

- A process can be thought of as an instance of a program in execution.

- A process is an independent entity to which system resources (e.g., CPU time and memory) are allocated.
- Each process is executed in a separate address space, and one process cannot access the variables and data structures of another process. If a process wishes to access another process's resources, inter-process communications have to be used. These include pipes, files, sockets, and other forms.

Thread

- A thread exists within a process and shares the process's resources (including its heap space).
- Multiple threads within the same process will share the same heap space. This is very different from processes, which cannot directly access the memory of another process. Each thread still has its own registers and its own stack, but other threads can read and write the heap memory.
- A thread is a particular execution path of a process. When one thread modifies a process resource, the change is immediately visible to sibling threads.

Reentrant lock

- A lock that can be acquired in one method and released in another.

Semaphore

- An implementation of the classic synchronization mechanism, a lock with a count.

Bound resources

- Resources of a fixed size or number used in a concurrent environment. Examples include database connections and fixed-size read/write buffers.

Starvation

- One thread or a group of threads is prohibited from proceeding for an excessively long time or forever. For example, always letting fast-running threads through first could starve out longer running threads if there is no end to the fast-running threads.

Livelock

- Threads in lockstep, each trying to do work but finding another "in the way." Due to resonance, threads continue trying to make progress but are unable to for an excessively long time - or forever.

Deadlock

- Two or more threads waiting for each other to finish. Each thread has a resource that the other thread requires and neither can finish until it gets the other resource. There are 4 conditions required for deadlock to occur. All 4 of these conditions must hold for deadlock to be possible. Break any one of these conditions and deadlock is not possible:
- **Mutual Exclusion**
  - Mutual exclusion occurs when multiple threads need to use the same resources and those resources a) cannot be used by multiple threads at the same time, and b) are limited in number. A common example of such a resource is a database connection, a file open for write, a record lock, or a semaphore. **So mutual exclusion means that only 1 process or thread can access a resource at a given time.** (Or, more accurately, there is limited access to a resource. A deadlock could also occur if a resource has limited quantity.)
  - Breaking Mutual Exclusion:
    - Using resources that allow simultaneous use, for example, AtomicInteger.
    - Increasing the number of resources such that it equals or exceeds the number of competing threads.
    - Checking that all your resources are free before seizing any.

    Unfortunately, most resources are limited in number and don't allow simultaneous use. And it's not uncommon for the identity of the second resource to be predicated on the results of operating on the first.
- **Lock & Wait**
  - Once a thread acquires a resource, it will not release the resource until it has acquired all of the other resources it requires and has completed its work. So this is a situation where processes that already hold a resource can request additional resources, without relinquishing their current resources.
  - Breaking Lock & Wait:
    - Check each resource before you seize it, and release all resources and start over if you run into one that's busy.

    This approach introduces several potential problems:
    - Starvation. One thread keeps being unable to acquire the resources it needs (maybe it has a unique combination of resources that seldom all become available). This leads to low CPU utilization.
    - Livelock. Several threads might get into lockstep and all acquire 1 resource and then release 1 resource, over and over again. This is especially likely with simplistic CPU scheduling algorithms (think embedded devices or simplistic hand-written thread balancing algorithms). This leads to high and useless CPU utilization.

- **No preemption**
  - One thread cannot take resources away from another thread. Once a thread holds a resource, the only way for another thread to get it is for the holding thread to release it.
  - Breaking Preemption:
    * Allow threads to take resources away from other threads. This is usually done through a simple request mechanism.
    * When a thread discovers that a resource is busy, it asks the owner to release it. If the owner is also waiting for some other resource, it releases them all and starts over. Managing all those requests can be tricky.
- **Circular Wait**
  - Imagine 2 threads, T1 and T2, and 2 resources, R1 and R2. T1 has R1, T2 has R2. T1 also requires R2, and T2 also requires R1.
  - Breaking Circular Wait:
    * **This is the most common approach to preventing deadlock.** For most systems it requires no more than a simple convention agreed to by all parties.
    * From the previous example, by simply forcing both Thread 1 and Thread 2 to **allocate resources in the same order makes circular wait impossible.**
    * More generally, if all threads can agree on a **global ordering of resources** and if they all allocate resources in that order, then deadlock is impossible.

Execution models used in concurrent programming

Most concurrent problems you will likely encounter will be some variation of these 3 problems:

- **Producer-Consumer**
  - One or more producer threads create some work and place it in a buffer or queue. One or more consumer threads acquire that work from the queue and complete it.
  - The queue between the producers and consumers is a bound resource. This means producers must wait for free space in the queue before writing and consumers must wait until there is something in the queue to consume.
  - Coordination between the producers and consumers via the queue involves producers and consumers signaling each other. The producers write to the queue and signal that the queue is no longer empty. Consumers read from

the queue and signal that the queue is no longer full. Both potentially wait to be notified when they can continue.

- **Readers-Writers**
  - When you have a shared resource that primarily serves as a source of information for readers, but which is occasionally updated by writers, throughput is an issue.
  - Emphasizing throughput can cause starvation and the accumulation of stale information. Allowing updates can impact throughput. Coordinating readers so they do not read something a writer is updating and vice versa is a tough balancing act. Writers tend to block many readers for a long period of time, thus causing throughput issues.
  - The challenge is to balance the needs of both readers and writers to satisfy correct operation, provide reasonable throughput and avoiding starvation.
- **Dining Philosophers**
  - Imagine a number of philosophers sitting around a circular table. A fork is placed to the left of each philosopher. There is a big bowl of spaghetti in the center of the table. The philosophers spend their time thinking unless they get hungry. Once hungry, they pick up the forks on either side of them and eat. A philosopher cannot eat unless he is holding two forks. If the philosopher to his right or left is already using one of the forks he needs, he must wait until that philosopher finishes eating and puts the forks back down. Once a philosopher eats, he puts both his forks back down on the table and waits until he is hungry again.
  - Replace philosophers with threads and forks with resources and this problem is similar to many enterprise applications in which processes compete for resources. Unless carefully designed, systems that compete in this way can experience deadlock, livelock, throughput, and efficiency degradation.

2 Testing

- **Tests are as important to the health of a project as the production code is.** Perhaps they are even more important, because tests preserve and enhance the flexibility, maintainability, and reusability of the production code. So keep your tests constantly clean. Work to make them expressive and succinct. Invent testing APIs that act as domain-specific language that helps you write the tests. If you let the tests rot, then your code will rot too. Keep your tests clean.
- What makes a clean test? Three things. Readability, readability, and readability. Readability is perhaps even more important in unit tests than it is in production code. What makes tests readable? The same thing that makes all code readable: clarity, simplicity, and density of expression. In a test you want to say a lot with as few expressions as possible.
- There are multiple types of testing and for each one there must be clean separation between them. Each category also requires different setup and process how to tests.
- Besides the following 3 types, there should be also **continuous code inspection** - static code analyzers like pylint (**or even better - coala**, which has multiple inspection tools integrated and usable as docker image - 90+ analyzers including python, bash, json, yaml, ...); each commit is analyzed automatically with that. These are cheap and fast checks.
- More tests devs produce, the better will CI works, the easier CD.
- "It is important to note that testing is not a phase, it's an activity and that testing starts from the very beginning of the development process, right from when the user stories are written."[1]
- Tests should be easily executable, writable, readable, automation friendly, isolated, and consistent across a project (the same structure and framework). They should be as close to developers as possible – devs let to run any test they need in short time as well as they must get feedback as soon as possible.
- The *build-operate-check* pattern defines the structure of the tests. Each of the tests is clearly split into 3 parts. The first part builds up the test data, the second part operates on that test data, and the third part checks that the operation yielded the expected results.

[1] https://www.testingexcellence.com/how-to-setup-a-qa-function-from-scratch-for-agile-projects/

- Golden Rule of API Design fits in: It's not enough to write tests for an API you develop; you have to write unit tests for code that uses your API.
- Tests are also independently deployable. In fact, most of the time they are deployed in test systems, rather than in production systems. Tests are the most isolated system component.
- QA - Quality Assurance, QE - Quality Engineer(ing).
  - Developers should be responsible for testing their own code by writing unittests and easier functional tests.
  - QEs must know about features, devs must know which parts of code are QEs suffering.
  - QEs must tightly collaborate with developers and vice versa and should be rather mixed together in 1 room).
  - **QE responsibilities**
    - Thinking about project from customer POW (high level design review).
    - Testing product from customer POW – QA doesn't care about implementation details, but functionality.
    - Maintain internal test frameworks (with developers).
    - Maintain CI automation, tools (with developers).
    - RC testing, testing deployment, upgrades.
    - Tests suites: system tests, complex integration tests. Also (if needed) additional testing: performance, security, ...
    - Sharing knowledge about proper testing with developers.
    - **BUT NOT**: unit testing, designing or coding features, bug-fixing product, code review, simple integration tests, knowledge of implementation details.

Test-Driven-Development

- There are 3 laws of TDD:
  - First Law - You may not write production code until you have written a failing unit test.
  - Second Law - You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
  - Third Law - You may not write more production code than is sufficient to pass the currently failing test. This law is also known as "fake it 'til you make it". It is a seemingly idiotic practice with massive benefits. See https://www.youtube.com/watch?v=PhiXo5CWjYU.

    * It helps you achieve and maintain flow. TDD helps you to achieve initial momentum and not getting stack thinking about the details of perfect implementation.
    * It helps you increase test coverage.
    * You might incrementally move toward better algorithms.

- These 3 laws lock you into a cycle that is perhaps 30 seconds long. The tests and the production code are written together, with the tests just a few seconds ahead of the production code. If we work this way, we will write dozens of tests every day, hundreds of tests every month, and thousands of tests every year. If we work this way, those tests will cover virtually all of our production code.
- TDD cycle has 3 parts - red, green, refactor. The first thing to do is to write the simplest possible implementation (and not just one silly use case).
- You begin by writing a small portion of a unit test. But within a few seconds you must mention the name of some class or function you have not written yet, thereby causing the unit test to fail to compile. So you must write production code that makes the test compile. But you can't write any more than that, so you start writing more unit test code.
- One of the most powerful benefits of TDD is that when you have a suite of tests that you trust, then you lose all fear of making changes. When you see bad code, you simply clean it on the spot.
- The tests you write after the fact are defense. The tests you write first are offense. After-the-fact tests are written by someone who is already vested in the code and already knows how the problem was solved. There's just no way those tests can be anywhere near as incisive as tests written first.
- TDD is a discipline that enhances certainty, courage, defect reduction, documentation, and design.
- There are times when following the 3 laws is simply impractical or inappropriate. These situations are rare, but they exist. **No professional developer should ever follow a discipline when that discipline does more harm than good.**

F.I.R.S.T.

Clean tests follow these 5 rules:

- **Fast**. Tests should be fast. They should run quickly. When tests run slow, you won't want to run them frequently. If you don't run them frequently, you won't find problems early enough to fix them easily. You won't feel as free to clean up the code. Eventually the code will begin to rot.

- **Independent**. Tests should not depend on each other. One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like.
- **Repeatable**. If your tests aren't repeatable in any environment, then you'll always have an excuse for why they fail. You'll also find yourself unable to run the tests when the environment isn't available
- **Self-validating**. The tests should have a boolean output. Either they pass or fail. You should not have to read through a log file to tell whether the tests pass. You should not have to manually compare two different text files to see whether the tests pass. If the tests aren't self-validating, then failure can become subjective and running the tests can require a long manual evaluation.
- **Timely**. The tests need to be written in a timely fashion. Unit tests should be written just before the production code that makes them pass. If you write tests after the production code, then you may find the production code to be hard to test. You may decide that some production code is too hard to test. You may not design the production code to be testable.

Test Hierarchy

More details about each type of tests is written in the following sections.

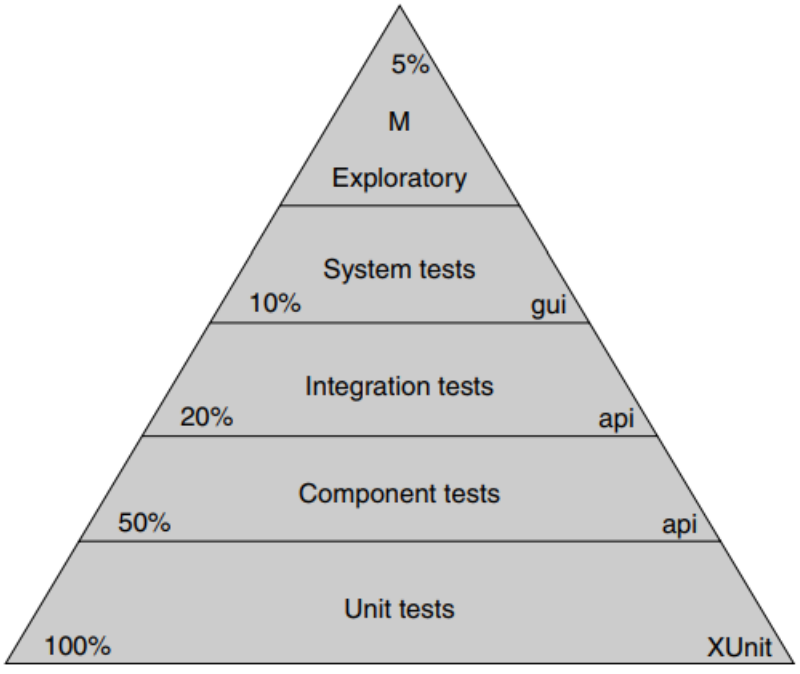

Figure 2.1: Test hierarchy including a coverage.

2.1 Unit Tests

- Service is not running, parts of code are tested (using mocking data, no database, I/O, or network connections).
- Testing of functions, methods, objects, classes => units.
- Should be the fastest test suite.
- Run by developers during development of code.
- Should have the biggest coverage.
- It is unit tests that keep our code flexible, maintainable, and reusable. The reason is simple. If you have tests, you do not fear making changes to the code! Without tests every change is a possible bug. No matter how flexible your architecture is, no matter how nicely partitioned your design, without tests you will be reluctant to make changes because of the fear that you will introduce undetected bugs. Tests enable change.
- At the bottom of the test pyramid (figure above) are the unit tests. These tests are written by programmers, for programmers, in the programming language of the system. The intent of these tests is to specify the system at the lowest level. Developers write these tests before writing production code as a way to specify what they are about to write.
- Unit tests provide as close to 100% coverage as is practical. Generally this number should be somewhere in the 90s.

2.2 Component Tests

- These are some of the acceptance tests. Generally they are written against individual components of the system. The components of the system encapsulate the business rules, so the tests for those components are the acceptance tests for those business rules.
- A component test wraps a component. It passes input data into the component and gathers output data from it. It tests that the output matches the input. Any other system components are decoupled from the test using appropriate mocking and test-doubling techniques.
- Component tests are written by QA with assistance from development.
- Component tests cover roughly half the system. They are directed more towards happy-path situations and very obvious corner, boundary, and alternate-path cases. The vast majority of unhappy-path cases are covered by unit tests and are meaningless at the level of component tests.

2.3 Integration Tests

- The whole set of services is running and being tested how cooperates. Additional services may be required, such as database or network.
- Should be run after successful unit testing. Usually needs more resources than unit testing.
- Docker may improve setup of integration tests:
  - Isolated test, multiple isolated tests can be run in parallel.
  - Pre-configured docker images with services/DB.
  - Docker-compose: allows running multiple containers in one test setup.
  - Just provide test data and mount container to your git repo.
  - The same containers can be used for CI/QA/developers (no duplication).
  - With all mentioned above: it should be possible to run integration tests just by 1 command, no extra effort from devel side.
- These tests only have meaning for larger systems that have many components. These tests assemble groups of components and test how well they communicate with each other. The other components of the system are decoupled as usual with appropriate mocks and test-doubles.
- They do not test business rules. Rather, they test how well the assembly of components dances together. They are plumbing tests that make sure that the components are properly connected and can clearly communicate with each other.
- Integration tests are typically written by the system architects, or lead designers, of the system. The tests ensure that the architectural structure of the system is sound. It is at this level that we might see performance and throughput tests.
- Integration tests are typically written in the same language and environment as component tests. They are typically not executed as part of the CI suite, because they often have longer run-times. Instead, these tests are run periodically (nightly, weekly, etc.) as deemed necessary by their authors.
- Once this stage is done, usually *Smoke Testing* is performed. These are very short tests (either unit of functional tests), that will determine whether a given system (or a component) is ready for the next stage of testing. If all parts of the application or component are implemented and runnable. They focus only on the main functionality, that should be pretty much stable. They are mostly automatized. The next phase once this is successful, is *System Testing*.
- Difference between unit and integration tests[2]:

[2] https://www.youtube.com/watch?v=uCxL7NGEohI

- Integrating with anything non-deterministic means your test must be considered as integration test.
- Integrating with deterministic parts of your language (such as standard library) does not necessarily make your test an integration test.
- Integrating with any 3rd party code makes your test an integration test. (!)
- Integrating with any other class/module that you own makes your test an integration test.

If you want to turn your integration test into unit test, maybe a solution is to use dependency injection.

2.4 System Tests

- Service is running and is tested as 1 piece. These tests should test only 1 service at once.
- Testing the product itself, testing how all subsystems cooperate (like production deployment).
- This is the final phase of testing, the most resources are spent here.
- These are automated tests that execute against the entire integrated system. They are the ultimate integration tests. They do not test business rules directly. Rather, they test that the system has been wired together correctly and its parts inter-operate according to plan. We would expect to see throughput and performance tests in this suite.
- These tests are written by the system architects and technical leads. Typically they are written in the same language and environment as integration tests for the UI. They are executed relatively infrequently depending on their duration, but the more frequently the better.
- System tests cover perhaps 10% of the system. This is because their intent is not to ensure correct system behavior, but correct system construction. The correct behavior of the underlying code and components have already been tested in the lower layers of the pyramid.

2.5 Exploratory Tests

- This is where humans put their hands on the keyboards and their eyes on the screens. These tests are not automated, nor are they scripted. The intent of these tests is to explore the system for unexpected behaviors while confirming expected behaviors.
- The goal is not coverage. We are not going to prove out every business rule and every execution pathway with these tests. Rather, the goal is to ensure that the system behaves well under human operation and to creatively find as many "peculiarities" as possible.
- It is possible to invest day or two of "bug hunting" in which as many people as possible, including managers, secretaries, programmers, testers, and tech writers, "bang" on the system to see if they can make it break.
- Exploratory testing is often thought of as a black box testing technique. It may cover also a quick smoke testing.

2.6 Specialized Tests

These tests are not always implemented. Here belong acceptance tests, functional tests, and non-functional tests.

Acceptance Tests

These tests are written by a collaboration of the stakeholders and the programmers in order to define when a requirement is done.

- The word "done" means all code written, all tests pass, QA and the stakeholders have accepted. Done. But how can you get this level of done-ness and still make quick progress from iteration to iteration? You create a set of automated tests that, when they pass, meet all of the above criteria! When the acceptance tests for your feature pass, you are done.
- The purpose of acceptance tests is communication, clarity, and precision. By agreeing to them, the developers, stakeholders, and testers all understand what the plan for the system behavior is. Achieving this kind of clarity is the responsibility of all parties.
- Acceptance tests must always be automated (because of cost). Don't look at these tests as extra work. Look at them as massive time and money savers. These tests will prevent you from implementing the wrong system and will allow you to know when you are done.
- In an ideal world, the stakeholders and QA would collaborate to write these tests, and developers would review them for consistency. In the real world, stakeholders seldom have the time or inclination to dive into the required level of detail. So they often delegate the responsibility to business analysts, QA, or even developers. If it turns out that developers must write these tests, then take care that the developer who writes the test is not the same as the developer who implements the tested feature.
- Unit tests dig into the guts of the system making calls to methods in particular classes. Acceptance tests invoke the system much farther out, at the API or sometimes even UI level. So the execution pathways that these tests take are very different.
- Keep the GUI tests to a minimum. They are fragile, because the GUI is volatile. The more GUI tests you have the less likely you are to keep them.
- The only way Robert C. Martin knows of to effectively eliminate communication errors between programmers and stakeholders is to write automated acceptance tests.

- It should be QA's role to work with business to create the automated acceptance tests that become the true specification and requirements document for the system. Iteration by iteration they gather the requirements from business and translate them into tests that describe to developers how the system should behave.

Functional Tests

- These are a type of black-box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered (unlike white-box testing).
- Functional testing usually describes what the system does. Functional testing does not imply that you are testing a function (method) of your module or class. Functional testing tests a slice of functionality of the whole system.
- Functional testing differs from system testing in that functional testing "verifies a program by checking it against ... design document(s) or specification(s)", while system testing "validate[s] a program by checking it against the published user or system requirements".
- Functional testing has many types:
  - Smoke testing (see above).
  - Sanity testing - a sanity test or sanity check is a basic test to quickly evaluate whether a claim or the result of a calculation can possibly be true. It is a simple check to see if the produced material is rational. The point of a sanity test is to rule out certain classes of obviously false results, not to catch every possible error.
  - Regression testing - this is re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change.
  - Usability testing - this is a technique used in user-centered interaction design to evaluate a product by testing it on users.

Non-Functional Tests

This is the testing of a software application or system for its non-functional requirements: the way a system operates, rather than specific behaviors of that system. It can be for example:

- Documentation testing
- Load testing
- Performance testing

- Recovery testing
- Security testing
- Scalability testing
- Stress testing

3 Clean Code

- The only way to go fast, is to go well. (productivity / clean code / clean architecture)
- The only way to make the deadline and the only way to go fast is to keep the code as clean as possible at all times.
- Generalizing your code in this way to make it adaptable for something which will never happen costs complexity and inflexibility in other areas plus time and thought, which is wasted.
- Any time you bring in a new software engineering technique, you have to consider what do I get, and what does it cost. Everything has a cost, but its proponents won't tell you so much about it. At the very least, it will make your code less straightforward, and someone who doesn't know the trick you're pulling will have a hard time following what's going on. And quite likely, it will reduce the performance of your code to some extent.
- A programmer with "code-sense" will look at a messy module and see **options and variations**.
- Bad code tries to do too much, it has muddled intent and ambiguity of purpose. Clean code is focused. Each function, each class, each module exposes a single-minded attitude that remains entirely undistracted, and unpolluted, by the surrounding details.
- Code, without tests, is not clean. It should have unit and acceptance tests.
- Duplication. When the same thing is done over and over, it's a sign that there is an idea in our mind that is not well represented in the code. I try to figure out what it is. Then I try to express that idea more clearly.
- Reduced duplication, high expressiveness, and early building of simple abstractions. That's a clean code.
- **The ratio of time spent reading vs. writing is well over 10:1**. We are constantly reading old code as part of the effort to write new code. Because this ratio is so high, we want the reading of code to be easy, even if it makes the writing harder. Of course there's no way to write code without reading it, so making it easy to read actually makes it easier to write.

- Nothing has a more profound and long-term degrading effect upon a development project than a bad code. Bad schedules can be redone, bad requirements can be redefined. Bad team dynamics can be repaired. But bad code rots and ferments, becoming an inexorable weight that drags the team down.
- Each variable should have the smallest possible scope. For example, a local object can be declared right before its first usage.
- Programming is often an exploration. You think you know the right algorithm for something, but then you wind up fiddling with it, prodding and poking at it, until you get it to "work." Before you consider yourself to be done with a function, make sure you understand how it works. It is not good enough that it passes all the tests. You must know that the solution is correct. Often the best way to gain this knowledge and understanding is to refactor the function into something that is so clean and expressive that it is obvious how it works.
- **Distinguish business exceptions from technical.** It is a potential source of confusion to represent them both using the same exception hierarchy, not to mention the same exception class. It would be a mistake to attempt to resolve these situations you caused yourself. Instead, we let the exception bubble up to the highest architectural level and let some general exception-handling mechanism do what it can to ensure that the system is in a safe state, such as rolling back a transaction, logging and alerting administration, and reporting back (politely) to the user. Mixing technical exceptions and business exceptions in the same hierarchy blurs the distinction and confuses the caller about what the method contract is, what conditions it is required to ensure before calling, and what situations it is supposed to handle.
- **Replace magic numbers with named constants.** However, some constants are so easy to recognize that they don't always need a named constant to hide behind so long as they are used in conjunction with very self-explanatory code.
- **Encapsulate conditionals.** Extract functions that explain the intent of the conditional. For example:

  if (shouldBeDeleted(timer))

  is preferable to

  if (timer.hasExpired() && !timer.isRecurrent())
- **Avoid negative conditionals.** Negatives are just a bit harder to understand than positives.
- **Encapsulate boundary conditions.** Boundary conditions are hard to keep track of. Put the processing for them in one place. Don't let them leak all over the code. For example:

if(level + 1 < tags.length) { parts = new Parse(body, tags, level + 1, offset + endTag); body = null; }

and better is:

int nextLevel = level + 1; if(nextLevel < tags.length) { parts = new Parse(body, tags, nextLevel, offset + endTag); body = null; }

- **Names should describe side-effects.** Names should describe everything that a function, variable, or class is or does. Don't hide side effects with a name. For example, *getSomeObj()* is not a proper name, if this function also creates such object *'SomeObj'*. Better name would be *createOrReturnSomeObj()*.

- **Don't use unambiguous names**, for example *doRename()* and *renamePage()*. What do the names tell you about the difference between the two functions? Nothing. A better name for that function is *renamePageAndOptionallyAllReferences()*. This may seem long, and it is, but it's only called from one place in the module, so it's explanatory value outweighs the length.

- **Frameworks** can be very powerful and very useful. Framework authors often believe very deeply in their frameworks. The examples they write for how to use their frameworks are told from the point of view of a true believer. Other authors who write about the framework also tend to be disciples of the true belief. They show you the way to use the framework. Often they assume an all-encompassing, all-pervading, let-the-framework-do-everything position. **This is not the position you want to take. Look at each framework skeptically. Yes, it might help, but at what cost?** Ask yourself how you should use it, and how you should protect yourself from it.

- **Prefer domains-specific types to primitive types.**

- You've been focused for hours on some gnarly problem, and there's no solution in sight. **The trick is that while you're coding, the logical part of your brain is active and the creative side is shut out.** It can't present anything to you until the logical side takes a break.

3.1 Comments

"Don't comment bad code—rewrite it." — Brian W. Kernighan and P. J. Plaugher

- Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation.
- The proper use of comments is to compensate for our failure to express ourselves in code. Note that I used the word failure. I meant it. Comments are always failures. We must have them because we cannot always figure out how to express ourselves without them, but their use is not a cause for celebration.
- So when you find yourself in a position where you need to write a comment, think it through and see whether there isn't some way to turn the tables and express yourself in code. Every time you express yourself in code, you should pat yourself on the back. Every time you write a comment, you should grimace and feel the failure of your ability of expression.
- Why am I so down on comments? Because they lie. Not always, and not intentionally, but too often. The older a comment is, and the farther away it is from the code it describes, the more likely it is to be just plain wrong. The reason is simple. Programmers can't realistically maintain them.
- Inaccurate comments are far worse than no comments at all. They delude and mislead. They set expectations that will never be fulfilled.
- Truth can only be found in one place: the code. Only the code can truly tell you what it does. It is the only source of truly accurate information.
- **Good comments**
  - Legal comments (licenses, authors and so on).
  - Informative comments (useful basic information with a comment). A comment like this can sometimes be useful, but it is better to use the name of the function to convey the information where possible.
  - Explanation of intent.
  - Clarification (sometimes it is just helpful to translate the meaning of some obscure argument or return value into something that's readable). There is a substantial risk, of course, that a clarifying comment is incorrect.
  - Warning of consequences.
  - TODO comments - TODOs are jobs that the programmer thinks should be done, but for some reason can't do at the moment. However, a lot of people think that TODO comments are not good any more, because we have issue tracking systems for features, bugs, and ideas.

  - Amplification of something very important.

- **Bad comments**
  - Redundant, misleading, mandated, or noise comments.
  - Closing brace comments - if you find yourself wanting to mark your closing braces, try to shorten your functions instead.
  - Commented-out code - there was a time, back in the sixties, when commenting-out code might have been useful. But we've had good source code control systems for a very long time now. Those systems will remember the code for us. We don't have to comment it out any more. Just delete the code. We won't lose it. Promise.
  - Nonlocal information
  - Too much information
  - Inobvious connection - the connection between a comment and the code it describes should be obvious. The purpose of a comment is to explain code that does not explain itself. It is a pity when a comment needs its own explanation.

3.2 Formatting

- First of all, let's be clear. Code formatting is important. It is too important to ignore and it is too important to treat religiously. Code formatting is about communication, and communication is the professional developer's first order of business.
- The functionality that you create today has a good chance of changing in the next release, but the readability of your code will have a profound effect on all the changes that will ever be made.
- **The newspaper metaphor.** Think of a well-written newspaper article. You read it vertically. We would like a source file to be like a newspaper article. The name should be simple but explanatory. The name, by itself, should be sufficient to tell us whether we are in the right module or not. The topmost parts of the source file should provide the high-level concepts and algorithms. Detail should increase as we move downward, until at the end we find the lowest level functions and details in the source file.
- **Vertical openness between concepts**. Nearly all code is read left to right and top to bottom. Each line represents an expression or a clause, and each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines. Each blank line is a visual cue that identifies a new and separate concept.
- **Vertical density.** If openness separates concepts, then vertical density implies close association. So lines of code that are tightly related should appear vertically dense.
- **Vertical distance.** Concepts that are closely related should be kept vertically close to each other. Clearly this rule doesn't work for concepts that belong in separate files. But then closely related concepts should not be separated into different files unless you have a very good reason. Indeed, this is one of the reasons that protected variables should be avoided.
- **Variable declarations**
  - Variables should be declared as close to their usage as possible. Because our functions are very short, local variables should appear a the top of each function. Control variables for loops should usually be declared within the loop statement.
  - In rare cases a variable might be declared at the top of a block or just before a loop in a long-ish function.

3.3 Error Handling

- Error handling is important, but if it obscures logic, it's wrong.
- Use exceptions rather than return codes (more details in the next section).
- Provide context with exceptions.
- Define exception classes in terms of a caller's needs.
- It is important to end up with a good amount of separation between your business logic and your error handling.
- **Don't return *Null***
  - When we return null, we are essentially creating work for ourselves and foisting problems upon our callers. All it takes is one missing null check to send an application spinning out of control. We invite errors with returning null.
  - If you are tempted to return null from a method, consider throwing an exception or returning a SPECIAL CASE object instead. If you are calling a null-returning method from a third-party API, consider wrapping that method with a method that either throws an exception or returns a special case object.
- **Don't pass *Null***
  - Returning null from methods is bad, but passing null into methods is worse. Unless you are working with an API which expects you to pass null, you should avoid passing null in your code whenever possible.
  - You can use assertions, but that does not solve the problem. It is better to avoid passing nulls where they don't belong.

3.4 Functions and Methods

- **Blocks and indenting**
  - The blocks within if statements, else statements, while statements, and so on should be one line long. Probably that line should be a function call.
  - Not only does this keep the enclosing function small, but it also adds documentary value because the function called within the block can have a nicely descriptive name.
- **Dependent functions**
  - If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible. This gives the program a natural flow. If the convention is followed reliably, readers will be able to trust that function definitions will follow shortly after their use.
- **Vertical ordering**
  - In general we want function call dependencies to point in the downward direction.
  - That is, a function that is called should be below a function that does the calling. This creates a nice flow down the source code module from high level to low level.
  - This is the exact opposite of languages like Pascal, C, and C++ that enforce functions to be defined, or at least declared, before they are used.
- **Do one thing**
  - A way to know that a function is doing more than "one thing" is if you can extract another function from it with a name that is not merely a restatement of its implementation.
- **One level of abstraction per function**
  - In order to make sure our functions are doing "one thing," we need to make sure that the statements within our function are all at the same level of abstraction.
  - Mixing levels of abstraction within a function is always confusing. Readers may not be able to tell whether a particular expression is an essential concept or a detail.
  - The statements within a function should all be written at the same level of abstraction, which should be one level below the operation described by the name of the function.
- **Reading code from top to bottom**
  - The Stepdown Rule: We want the code to read like a top-down narrative.

  - We want every function to be followed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions.

- **Switch or if/else statements**
  - It's hard to make a small switch or if/else statement. Even a switch statement with only two cases is larger than I'd like a single block or function to be. It's also hard to make a switch statement that does one thing. By their nature, switch statements always do *N* things. Unfortunately we can't always avoid switch statements, but we can make sure that each switch statement is buried in a low-level class and is never repeated. We do this, of course, with polymorphism. For example:

```
public Money calculatePay(Employee e)
    throws InvalidEmployeeType {
        switch (e.type) {
            case COMMISSIONED:
                return calculateCommissionedPay(e);
            case HOURLY:
                return calculateHourlyPay(e);
            case SALARIED:
                return calculateSalariedPay(e);
            default:
                throw new InvalidEmployeeType(e.type);
        }
}
```

src/switch_problem.java

  - There are a lot of problems with this function. First, it's large, and when new employee types are added, it will grow. Second, it very clearly does more than one thing. Third, it violates SRP because there is more than one reason for it to change. Fourth, it violates the OCP because it must change whenever new types are added. But possibly the worst problem with this function is that there are an unlimited number of other functions that will have the same structure.
  - The solution here is to bury the switch statement in the basement of an abstract factory design pattern, and never let anyone see it. The factory will use the switch statement to create appropriate instances of the derivatives of *Employee*, and the various functions, such as *calculatePay*, *isPayday*, and *deliverPay*, will be dispatched polymorphically through the Employee interface.
  - One general rule for switch statements is that they can be tolerated if they appear only once, are used to create polymorphic objects, and are hidden behind an inheritance relationship so that the rest of the system can't see them. Of course every circumstance is unique, and there are times when I violate one or more parts of that rule. Solution to above problem might me:

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
```

```
4     public abstract void deliverPay(Money pay);
5   }
6
7
8   public interface EmployeeFactory {
9     public Employee makeEmployee(EmployeeRecord r) throws
        InvalidEmployeeType;
10  }
11
12
13  public class EmployeeFactoryImpl implements EmployeeFactory {
14    public Employee makeEmployee(EmployeeRecord r) throws
        InvalidEmployeeType {
15      switch (r.type) {
16        case COMMISSIONED:
17          return new CommissionedEmployee(r) ;
18        case HOURLY:
19          return new HourlyEmployee(r);
20        case SALARIED:
21          return new SalariedEmploye(r);
22        default:
23          throw new InvalidEmployeeType(r.type);
24      }
25    }
26  }
```

src/switch_solution.java

- **Use descriptive names**
  - The smaller and more focused a function is (hardly 20 LoC), the easier it is to choose a descriptive name.
  - Don't be afraid to make a name long. A long descriptive name is better than a short enigmatic name. A long descriptive name is better than a long descriptive comment. Use a naming convention that allows multiple words to be easily read in the function names, and then make use of those multiple words to give the function a name that says what it does.
  - Choosing descriptive names will clarify the design of the module in your mind and help you to improve it. It is not at all uncommon that hunting for a good name results in a favorable restructuring of the code. Be consistent in your names. Use the same phrases, nouns, and verbs in the function names you choose for your modules.
  - Consider, for example, the names *includeSetupAndTeardownPages*, *includeSetupPages*, *includeSuiteSetupPage*, and *includeSetupPage*. The similar phraseology in those names allows the sequence to tell a story.
- **Function arguments**
  - The ideal number of arguments for a function is 0 (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification—and then shouldn't be used anyway.

  - Arguments are even harder from a testing point of view. Imagine the difficulty of writing all the test cases to ensure that all the various combinations of arguments work properly. If there are no arguments, this is trivial.
  - Flag arguments are ugly. Passing a boolean into a function is a truly terrible practice. It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing. It does one thing if the flag is true and another if the flag is false!
  - When a function seems to need more than 2 or 3 arguments, it is likely that some of those arguments ought to be wrapped into a class of their own. Reducing the number of arguments by creating objects out of them may seem like cheating, but it's not. When groups of variables are passed together, the way x and y are in the example above, they are likely part of a concept that deserves a name of its own. Consider, for example, the difference between the following declarations:

    Circle makeCircle(double x, double y, double radius);

    Circle makeCircle(Point center, double radius);

- **Have no side effects.** Side effects are lies. Your function promises to do one thing, but it also does other hidden things.

- Output (that are input) arguments.
  - For example, *"report"* is an output argument:

    public void appendFooter(StringBuffer report)
  - In the days before object oriented programming it was sometimes necessary to have output arguments. However, much of the need for output arguments disappears in OO languages because this is intended to act as an output argument. In other words, it would be better for *appendFooter* to be invoked as*: report.appendFooter();*
  - In general output arguments should be avoided. If your function must change the state of something, have it change the state of its owning object.

- **Command Query Separation**
  - Functions should either do something or answer something, but not both. Either your function should change the state of an object, or it should return some information about that object. Doing both often leads to confusion.
  - For example, function: *public boolean set(String attribute, String value);*
    * It sets the value of a named attribute and returns true if it is successful and false if no such attribute exists.
    * Imagine this from the point of view of the reader. What does it mean? Is it asking whether the "username" attribute was previously set to "uncle-bob"? Or is it asking whether the "username" attribute was successfully

set to “unclebob”? It’s hard to infer the meaning from the call because it’s not clear whether the word “set” is a verb or an adjective.

* The real solution is to separate the command from the query so that the ambiguity cannot occur:

```
if (attributeExists("username")) {
  setAttribute("username", "unclebob");
  ...
}
```

src/command_query_separation.java

- **Prefer exceptions to returning error codes**
  - Returning error codes from command functions is a subtle violation of command query separation. It promotes commands being used as expressions in the predicates of if statements: *if (deletePage(page) == E_OK)*
  - On the other hand, if you use exceptions instead of returned error codes, then the error processing code can be separated from the happy path code and can be simplified:

```
try {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
  logger.log(e.getMessage());
}
```

src/exceptions_instead_or_return_codes.java

- **Extract try/catch blocks**
  - Try/catch blocks are ugly in their own right. They confuse the structure of the code and mix error processing with normal processing. So it is better to extract the bodies of the try and catch blocks out into functions of their own.

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
```

```
18 }
```

src/extracted_exception_blocks.java

  - In the above, the *delete* function is all about error processing. It is easy to understand and then ignore. The *deletePageAndAllReferences* function is all about the processes of fully deleting a page. Error handling can be ignored. This provides a nice separation that makes the code easier to understand and modify.

- **Error handling is one thing.** Functions should do one thing. Error handing is one thing. Thus, a function that handles errors should do nothing else. This implies (as in the example above) that if the keyword try exists in a function, it should be the very first word in the function and that there should be nothing after the catch/finally blocks.

- **How to write clean functions?**
  - Writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down first, then you massage it until it reads well. The first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read.
  - When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code.
  - So then I massage and refine that code, splitting out functions, changing names, eliminating duplication. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing.
  - In the end, I wind up with functions that follow the rules I've laid down in this chapter. I don't write them that way to start. I don't think anyone could.

3.5 Classes

- **In general, base classes should know nothing about their derivatives. There are exceptions to this rule, of course.** Sometimes the number of derivatives is strictly fixed, and the base class has code that selects between the derivatives. However, in that case the derivatives and base class are strongly coupled and always deploy together in the same (for instance) jar file. In the general case we want to be able to deploy derivatives and bases in different jar files.
- Make logical dependencies physical.
- **Well-defined modules have very small interfaces that allow you to do a lot with a little.** Poorly defined modules have wide and deep interfaces that force you to use many different gestures to get simple things done. A well-defined interface does not offer very many functions to depend upon, so coupling is low. A poorly defined interface provides lots of functions that you must call, so coupling is high. Good software developers learn to limit what they expose at the interfaces of their classes and modules. The fewer methods a class has, the better. The fewer variables a function knows about, the better. The fewer instance variables a class has, the better. Hide your data. Hide your utility functions. Hide your constants and your temporaries. Don't create classes with lots of methods or lots of instance variables. Don't create lots of protected variables and functions for your subclasses. **Concentrate on keeping interfaces very tight and very small.**
- Interfaces occur at the highest level of abstraction (user interfaces), at the lowest (function interfaces), and at levels in between (class interfaces, library interfaces, etc.). Good interfaces are easy to use correctly, and hard to use incorrectly.
- **Encapsulation**
  - We like to keep our variables and utility functions private, but we're not fanatic about it.
  - Sometimes we need to make a variable or utility function protected so that it can be accessed by a test. For us, tests rule. If a test in the same package needs to call a function or access a variable, we'll make it protected or package scope.
- **Classes should be small**
  - The first rule of classes is that they should be small. The second rule of classes is that they should be smaller than that. With functions we measured size by counting physical lines. With classes we use a different measure. **We count responsibilities.**

  - The name of a class should describe what responsibilities it fulfills. In fact, naming is probably the first way of helping determine class size. If we cannot derive a concise name for a class, then it's likely too large. The more ambiguous the class name, the more likely it has too many responsibilities.

SOLID

- Good software systems begin with clean code. On the one hand, if the bricks aren't well made, the architecture of the building doesn't matter much. On the other hand, you can make a substantial mess with well-made bricks. This is where the SOLID principles come in. They were grouped and stabilized in early 2000s.
- The SOLID principles tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected. The use of the word "class" does not imply that these principles are applicable only to object-oriented software. A class is simply a coupled grouping of functions and data. Every software system has such groupings, whether they are called classes or not. The SOLID principles apply to those groupings.
- The goal of the principles is the creation of mid-level (module level) software structures that:
  - tolerate change,
  - are easy to understand, and
  - are the basis of components that can be used in many software systems.

SRP (The Single Responsibility Principle)

- It states that a class or module (or a function) should have one, and only one, reason to change (responsibility). This principle gives us both a definition of responsibility, and a guidelines for class size. Classes should have one responsibility—one reason to change. Each small class encapsulates a single responsibility, has a single reason to change, and collaborates with a few others to achieve the desired system behaviors.
- **Gather together those things that change for the same reason, and separate those things that change for different reasons.**
- It has an inappropriate name, it is very misunderstood by programmers and does not mean that every module should do just one thing. Make no mistake, there is a principle like that. A function should do one, and only one, thing. We use that principle when we are refactoring large functions into smaller functions; we use it at the lowest levels. But it is not one of the SOLID principles - it is not the SRP.
- Software systems are changed to satisfy users and stakeholders; those users and stakeholders are the "reason to change" that the principle is talking about. Indeed,

we can rephrase the principle to say this: A module should be responsible to one, and only one, user or stakeholder. Unfortunately, the words "user" and "stakeholder" aren't really the right words to use here. There will likely be more than one user or stakeholder who wants the system changed in the same way. Instead, we're really referring to a group—one or more people who require that change. We'll refer to that group as an actor. And by module we mean source file for example. Thus the final version of the SRP is: A module should be responsible to one, and only one, actor.

OCP (The Open-Closed Principle)

- Bertrand Meyer made this principle famous in the 1988. The gist is that for software systems to be easy to change, they must be designed to allow the behavior of those systems to be changed by adding new code, rather than changing existing code: **A software artifact should be open for extension but closed for modification.**
- We want to structure our systems so that we muck with as little as possible when we update them with new or changed features. In an ideal system, we incorporate new features by extending the system, not by making modifications to existing code.
- Most students of software design recognize the OCP as a principle that guides them in the design of classes and modules. But the principle takes on even greater significance when we consider the level of architectural components. If component A should be protected from changes in component B, then component B should depend on component A. Why should A hold such a privileged position? Because it contains the business rules. Higher-level components in a hierarchy are protected from the changes made to lower-level components.
- **The OCP is one of the driving forces behind the architecture of systems.** The goal is to make the system easy to extend without incurring a high impact of change. This goal is accomplished by partitioning the system into components, and arranging those components into a **dependency hierarchy that protects higher-level components from changes in lower-level components.**

LSP (The Liskov Substitution Principle)

- Barbara Liskov's famous definition of subtypes, from 1988.
- In short, this principle says that to build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another.
- So, if you have a base type, and a subtype, that subclass should be substitutable for the base class (at any point in a program).

- What this practically means is that a **method in a subclass must receive everything that base class is expecting to be able to receive, but it may also expect a bit more** (superset). What a **method in a subclass can return, must be the same as in base class, or just its subset**. Also, a **set of possible states of subclass must either be the same as in base class, or their subset. So basically, everything base class can do, must do also derived class (plus, of course, something more).** See videos [5:30][1] and [10:30][2]

  Some people (also see the videos above) recommend that composition si better than inheritance.

- So basically, you are liberal what to receive, but conservative what to send (if people were like this, world would be fantastic). No unexpected surprises, so that you may 'count' on subclass. In reality, we mostly do it wrong - we are subclassing 'too much'. You are maybe overusing inheritance in places where inheritance shouldn't actually be used.

- **Derived classes must be usable through the base class interface, without the need for the user to know the difference.**[3] For example: *Client* $\rightarrow$ *AbstractServer* $\leftarrow$ *ConcreteServer*, and we should be able to substitute that *AbstractServer* with *ConcreteServer* and the *Client* shouldn't know the difference. This is a simple polymorphism.

- **A simple violation of substitutability, can cause a system's architecture to be polluted with a significant amount of extra mechanisms.**

- This is probably the most technical of the 5 SOLID principles and the one I would guess fewest people consider much. However, it does have some important consequences for designing object-oriented software.

- If we can rely on the LSP, it allows us to use polymorphism reliably in our code. Polymorphism is a key way to avoid repetition in code as it allows you to maximize the generality of the code you are writing.

ISP (The Interface Segregation Principle)

- **This principle advises software designers to avoid depending on things that they don't use.** The lesson here is that depending on something that carries baggage that you don't need can cause you troubles that you didn't expect.

- **It says that it is better to have smaller interfaces, rather than a few very large interfaces. But be careful, this depends on a given project**

[1] https://www.youtube.com/watch?v=bVwZquRH1Vk&list=PLrhzvIcii6GMQceffIgKCRK98yTm0oolm&index=3
[2] https://www.youtube.com/watch?v=0bHQHszbIcE&list=PLrhzvIcii6GMsfGSgRL1xmS3Bvo4TPliQ
[3] https://www.youtube.com/watch?v=TMuno5RZNeE

and your requirements! Small can mean different things across the projects. In this way, we are favoring composition over inheritance, and decoupling over coupling. If you are thinking about microservices, it is basically the same thing (we are creating objects with small responsibilities and then we are composing them together).[4]

- In general, it is harmful to depend on modules that contain more than you need. This is obviously true for source code dependencies that can force unnecessary recompilation and redeployment—but it is also true at a much higher, architectural level.

- For example, consider that an architect is working on a system, S. He wants to include a certain framework, F, into the system. Now suppose that the authors of F have bound it to a particular database, D. So S depends on F. which depends on D. Now suppose that D contains features that F does not use and, therefore, that S does not care about. Changes to those features within D may well force the redeployment of F and, therefore, the redeployment of S. Even worse, a failure of one of the features within D may cause failures in F and S.

DIP (The Dependency Inversion Principle)

- In essence, the DIP says that our classes should depend upon abstractions, not on concrete details. An example, instead of depend on an implementation details of some concrete class, it would be better to depend on an interface instead.
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend on details. Details should depend on abstractions.
  - The "inversion" concept does not mean that lower-level layers depend on higher-level layers. Both layers should depend on abstractions that draw the behavior needed by higher-level layers.
  - The code that implements high-level policy should not depend on the code that implements low-level details. Rather, details should depend on policies.

- In a statically typed language, like Java, this means that the use, import, and include statements should refer only to source modules containing interfaces, abstract classes, or some other kind of abstract declaration. Nothing concrete should be depended on. The same rule applies for dynamically typed languages, like Ruby and Python. Source code dependencies should not refer to concrete modules. However, in these languages it is a bit harder to define what a concrete module is. In particular, it is any module in which the functions being called are implemented.

[4] https://www.youtube.com/watch?v=xahwVmf8itI&list=PLrhzvIcii6GMQceffIgKCRK98yTmOoolm&index=4

Clearly, treating this idea as a rule is unrealistic, because software systems must depend on many concrete facilities. For example, the String class in Java is concrete, and it would be unrealistic to try to force it to be abstract. The source code dependency on the concrete java.lang.string cannot, and should not, be avoided. By comparison, the String class is very stable. Changes to that class are very rare and tightly controlled. Programmers and architects do not have to worry about frequent and capricious changes to String. We tolerate such concrete dependencies because we know we can rely on them not to change.

- Good software designers and architects work hard to reduce the volatility of interfaces. They try to find ways to add functionality to implementations without making changes to the interfaces.
- DIP violations cannot be entirely removed, but they can be gathered into a small number of concrete components and kept separate from the rest of the system.

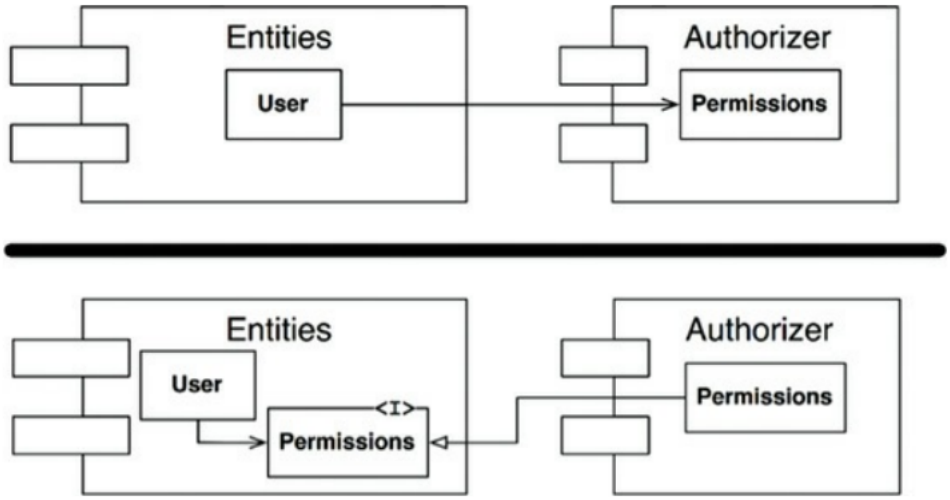

Figure 3.1: An example of DIP application.

3.6 System Level

- "*Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test.*" — Ray Ozzie, CTO, Microsoft Corporation
- Cities grow from towns, which grow from settlements. At first the roads are narrow and practically nonexistent, then they are paved, then widened over time. How many times have you driven, bumper to bumper through a road "improvement" project and asked yourself, "Why didn't they build it wide enough the first time!?" But it couldn't have happened any other way. Who can justify the expense of a six-lane highway through the middle of a small town that anticipates growth? It is a myth that we can get systems "right the first time." Instead, we should implement only today's stories, then refactor and expand the system to implement new stories tomorrow. This is the essence of iterative and incremental agility. Test-driven development, refactoring, and the clean code they produce make this work at the code level.
- **"Construction" is a very different process from "use".** Software systems should separate the startup process, when the application objects are constructed and the dependencies are "wired" together, from the runtime logic that takes over after startup.
- **Dependency Injection**
  - This is the application of *Inversion of Control (IoC)* to dependency management. Inversion of Control moves secondary responsibilities from an object to other objects that are dedicated to the purpose, thereby supporting the Single Responsibility Principle. In the context of dependency management, an object should not take responsibility for instantiating dependencies itself. Instead, it should pass this responsibility to another "authoritative" mechanism, thereby inverting the control.
  - True Dependency Injection goes one step further. The class takes no direct steps to resolve its dependencies; it is completely passive. Instead, it provides setter methods or constructor arguments (or both) that are used to inject the dependencies. During the construction process, the DI container instantiates the required objects (usually on demand) and uses the constructor arguments or setter methods provided to wire together the dependencies. Which dependent objects are actually used is specified through a configuration file or programmatically in a special-purpose construction module.
  - So it is used when you want to change behavior of your class in runtime, and not in compile time. It can be helpful for achieving Dependency Inversion (with interfaces and so on). So dependency injection increases flexibility of our code. Also, unit testing is easier because of isolation.
- **Big Design Up Front**

  - This is a practice of designing everything up front before implementing anything at all.
  - In fact, BDUF is even harmful because it inhibits adapting to change, due to the psychological resistance to discarding prior effort and because of the way architecture choices influence subsequent thinking about the design.
  - Building architects have to do BDUF because it is not feasible to make radical architectural changes to a large physical structure once construction is well underway.
  - We can start a software project with a "naively simple" but nicely decoupled architecture, delivering working user stories quickly, then adding more infrastructure as we scale up.
  - Of course, this does not mean that we go into a project "rudderless." We have some expectations of the general scope, goals, and schedule for the project, as well as the general structure of the resulting system. However, we must maintain the ability to change course in response to evolving circumstances.

- **Optimize decision making**
  - We often forget that it is also best to postpone decisions until the last possible moment. This isn't lazy or irresponsible; it lets us make informed choices with the best possible information. A premature decision is a decision made with sub-optimal knowledge.

- **Simple design**

  A design is "simple" if it follows these rules (given in order of importance):
  - **Tests**. Writing tests leads to better designs. Fortunately, making our systems testable pushes us toward a design where our classes are small and single purpose. It's just easier to test classes that conform to the SRP. The more tests we write, the more we'll continue to push toward things that are simpler to test. So making sure our system is fully testable helps us create better designs.
  - **No duplication**
    * Duplication is the primary enemy of a well-designed system. It represents additional work, additional risk, and additional unnecessary complexity.
    * The most obvious form of duplication is when you have clumps of identical code. These should be replaced with simple functions/methods.
    * Switch/case or if/else chain that appears again and again in various modules, always testing for the same set of conditions, is also duplication. These should be replaced by polymorphism.
    * In databases, there are *Codd Normal Forms* for eliminating duplication.

* Modules that have similar algorithms, but don't share similar lines of code. It is duplication and should be addressed by *template method* design pattern, which is a common technique for removing higher-level duplication. For example:

```
public class VacationPolicy {
  public void accrueUSDivisionVacation() {
    // code to calculate vacation based on hours worked to date
    // ...
    // code to ensure vacation meets US minimums
    // ...
    // code to apply vaction to payroll record
    // ...
   }
   public void accrueEUDivisionVacation() {
    // code to calculate vacation based on hours worked to date
    // ...
    // code to ensure vacation meets EU minimums
    // ...
    // code to apply vaction to payroll record
    // ...
   }
}
```

src/duplication_problem.java

can be refactored to:

```
abstract public class VacationPolicy {
  public void accrueVacation() {
    calculateBaseVacationHours();
    alterForLegalMinimums();
    applyToPayroll();
  }
  private void calculateBaseVacationHours() { /* ... */ };
  abstract protected void alterForLegalMinimums();
  private void applyToPayroll() { /* ... */ };
}

public class USVacationPolicy extends VacationPolicy {
  @Override protected void alterForLegalMinimums() {
    // US specific logic
  }
}

public class EUVacationPolicy extends VacationPolicy {
  @Override protected void alterForLegalMinimums() {
    // EU specific logic
}
```

src/duplication_solution.java

– **Expresses the intent of the programmer**

* The majority of the cost of a software project is in long-term maintenance. In order to minimize the potential for defects as we introduce change, it's critical for us to be able to understand what a system does. As systems become more complex, they take more and more time for a developer to understand, and there is an ever greater opportunity for a

misunderstanding. Therefore, code should clearly express the intent of its author.

  * Good names. Small functions and classes. Using standard nomenclature, for example design patterns (and use standard names for classes that implements these patterns).
  * Well-written unit tests - documentation by example. Someone reading our tests should be able to get a quick understanding of what a class is all about.

- **Minimizes the number of classes and methods**
  * In an effort to make our classes and methods small, we might create too many tiny classes and methods. So this rule suggests that we also keep our function and class counts low.
  * Our goal is to keep our overall system small while we are also keeping our functions and classes small. Remember, however, that this rule is the lowest priority of the four rules of simple design. So, although it's important to keep class and function count low, it's more important to have tests, eliminate duplication, and express yourself.

4 Design Patterns

They are a general repeatable solution to a commonly occurring problem in software design.

- A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.
- Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation.
- Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.
- Originally there were 23 design patterns basic, language-independent design patterns. Now there are about 40.
- They are also about more abstract understanding of software. They also help to create well structured software in less time. They also offer a "shared language" to communicate.
- Actually a lot of design patterns are implemented in programming language. So don't over-complicate things and don't use something that is built-in! For example, *Iterator* pattern.

4.1 Creational Patterns

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Singleton

- The experience shows that most singletons really do more harm than good. A lot of people never uses singleton, it is even considered as a code smell. It is "global", anyone can modify that without even knowing it - you lost control. Also, having just 1 instance in growing application can be misleading. Maybe in the future, there will be more instances. Also, in multi-threaded application, singleton can be dangerous.
- The single-instance requirement is often imagined. In many cases, it's pure speculation that no additional instances will be needed in the future.
- Singletons cause implicit dependencies between conceptually independent units of code. This is problematic both because they are hidden and because they introduce unnecessary coupling between units.
- Singletons also carry implicit persistent state, which hinders unit testing. Unit testing depends on tests being independent of one another, so the tests can be run in any order and the program can be set to a known state before the execution of every unit test. Once you have introduced singletons with mutable state, this may be hard to achieve. In addition, such globally accessible persistent state makes it harder to reason about the code, especially in a multi-threaded environment.
- Multi-threading introduces further pitfalls to the singleton pattern. As straightforward locking on access is not very efficient, the so-called double-checked locking pattern (DCLP) has gained in popularity. Unfortunately, this may be a further form of fatal attraction. It turns out that in many languages, DCLP is not thread-safe and, even where it is, there are still opportunities to get it subtly wrong.
- The cleanup of singletons may present also a challenge.
- Some of these shortcomings can be overcome by introducing additional mechanisms. However, this comes at the cost of additional complexity in code that could have been avoided by choosing an alternative design. Therefore, restrict your use of the Singleton pattern to the classes that truly must never be instantiated more than once. Don't use a singleton's global access point from arbitrary code.

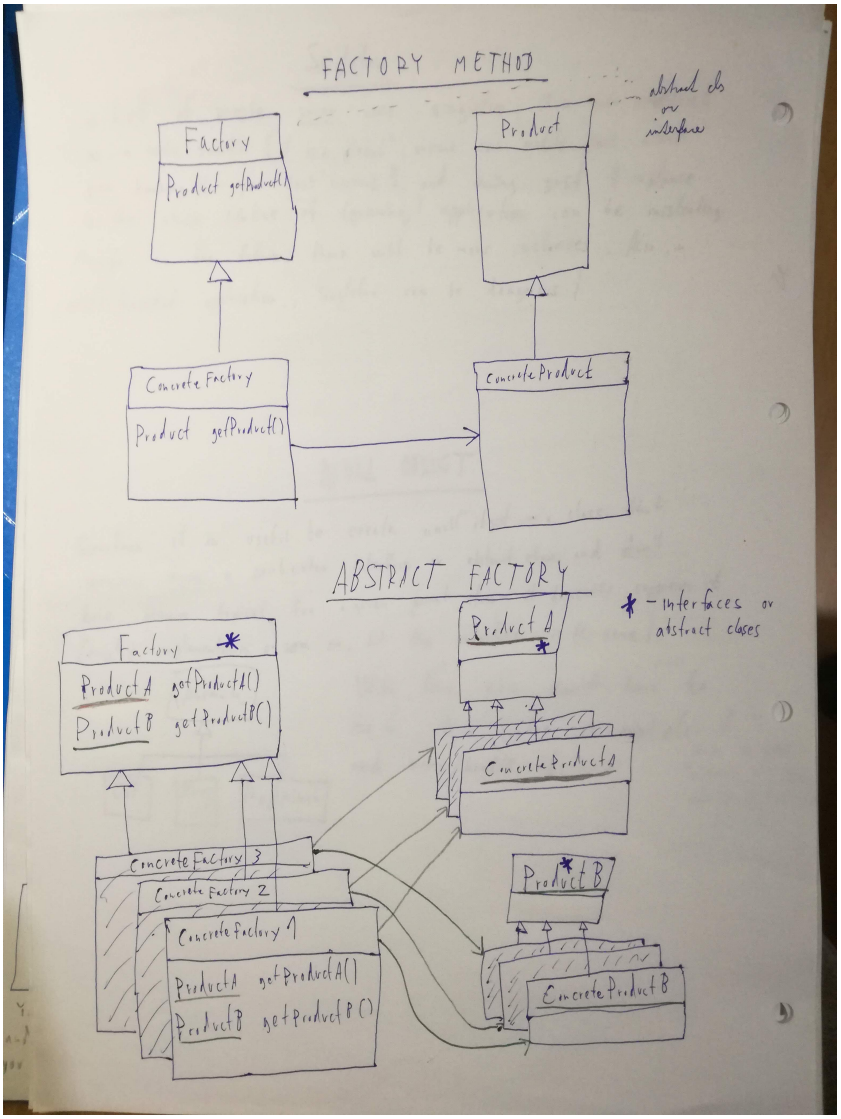

Figure 4.1: Factory method and Abstract Factory design patterns.

4.2 Structural Patterns

These design patterns are all about class and object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

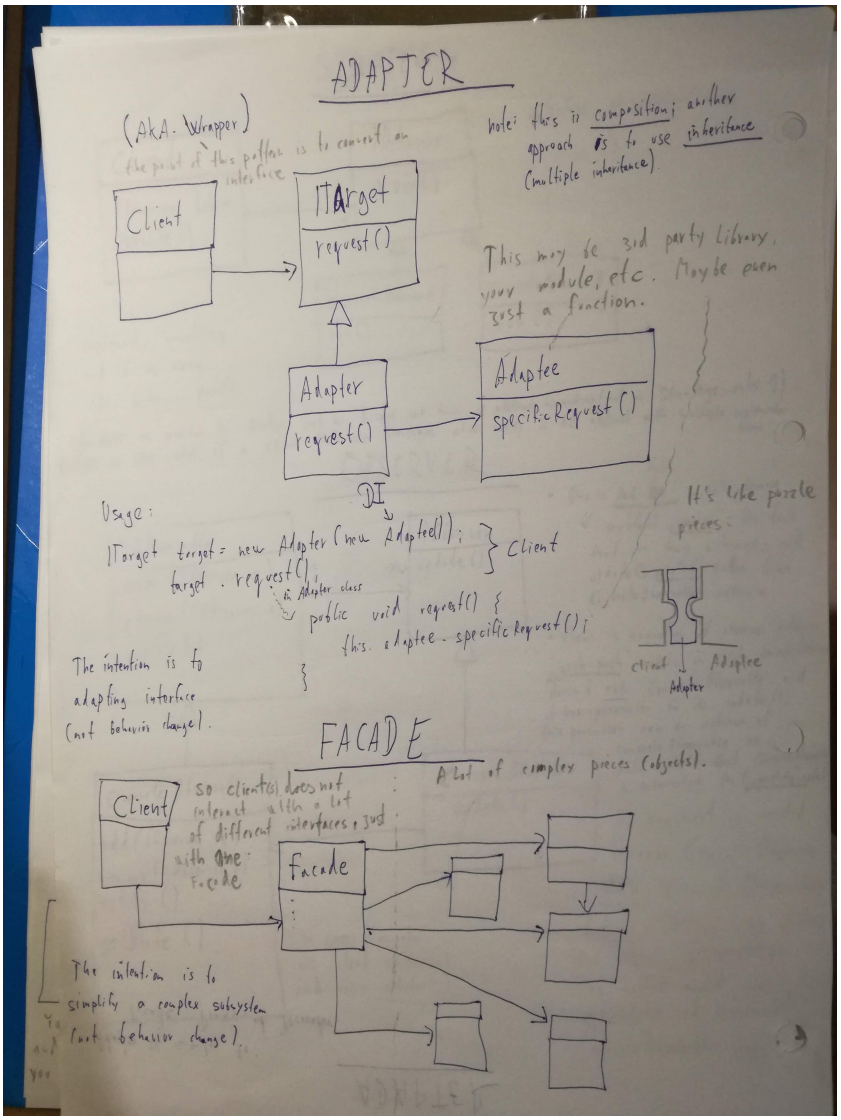

Figure 4.2: Adapter and Facade design patterns.

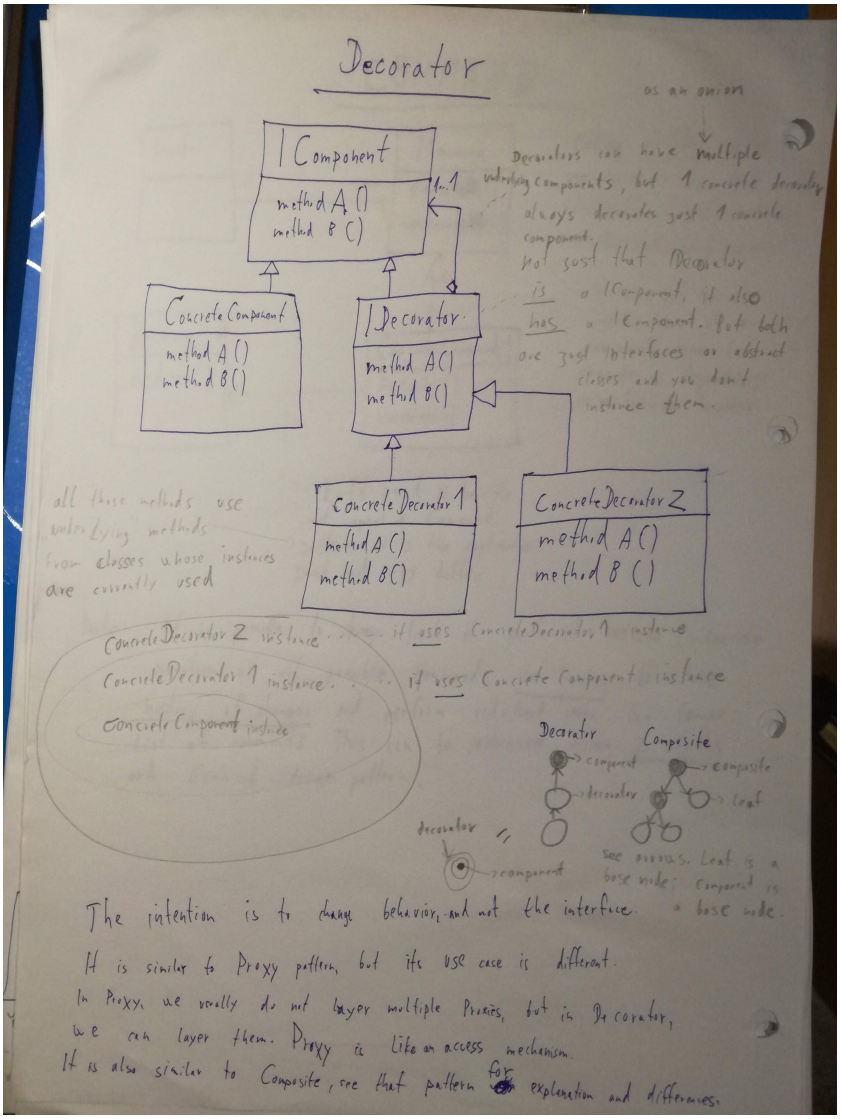

Figure 4.3: Decorator design pattern.

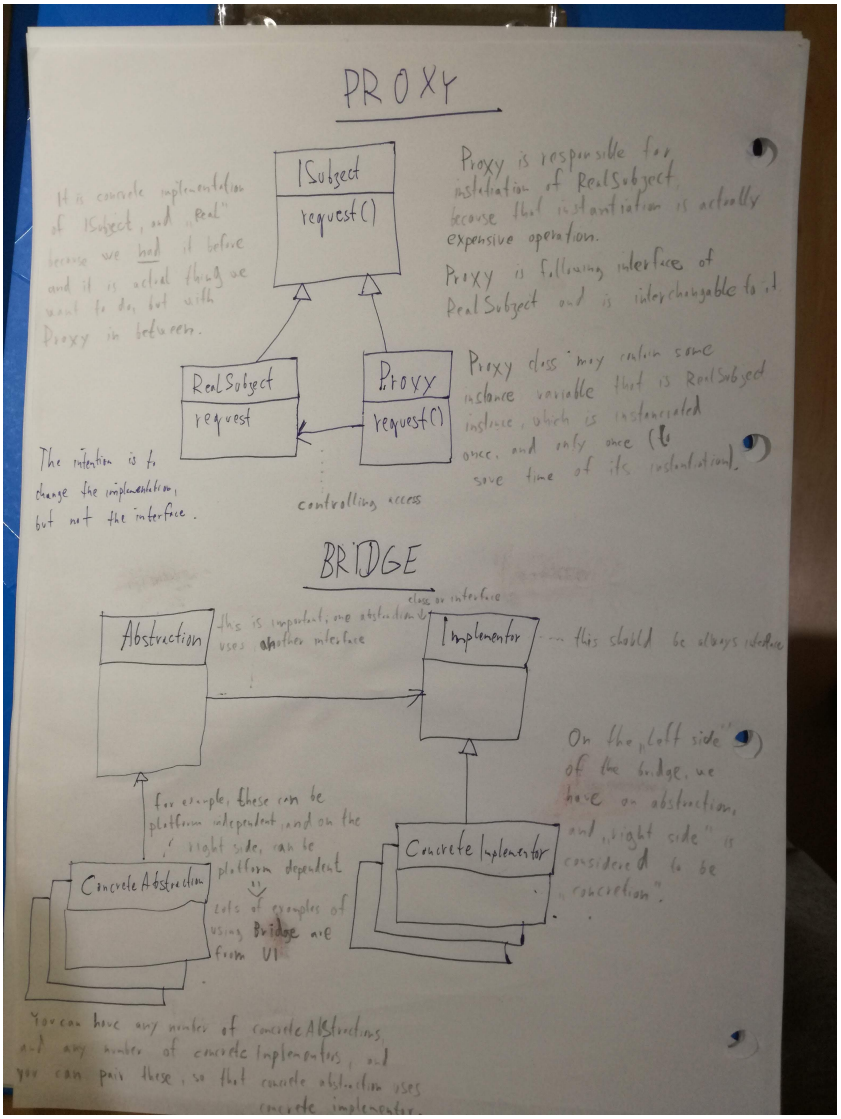

Figure 4.4: Proxy and Bridge design patterns.

4.3 Behavioral Patterns

These design patterns are all about class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

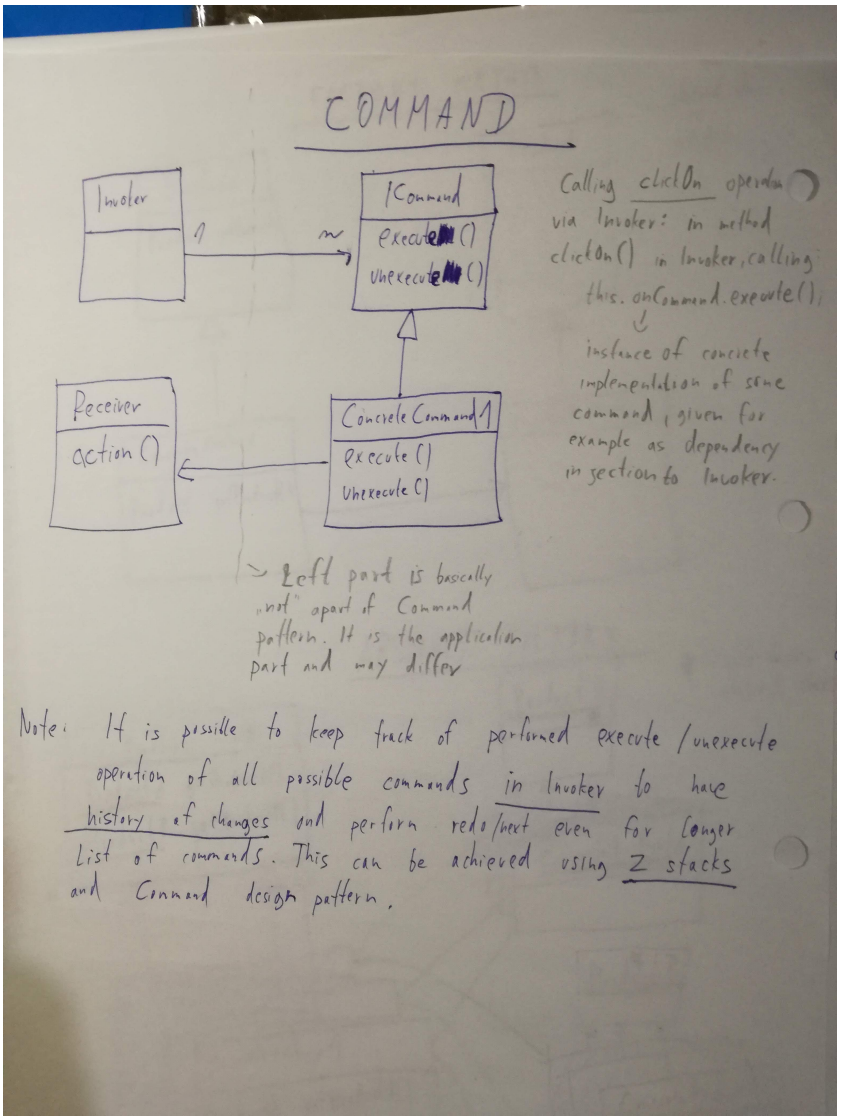

Figure 4.5: Command design pattern.

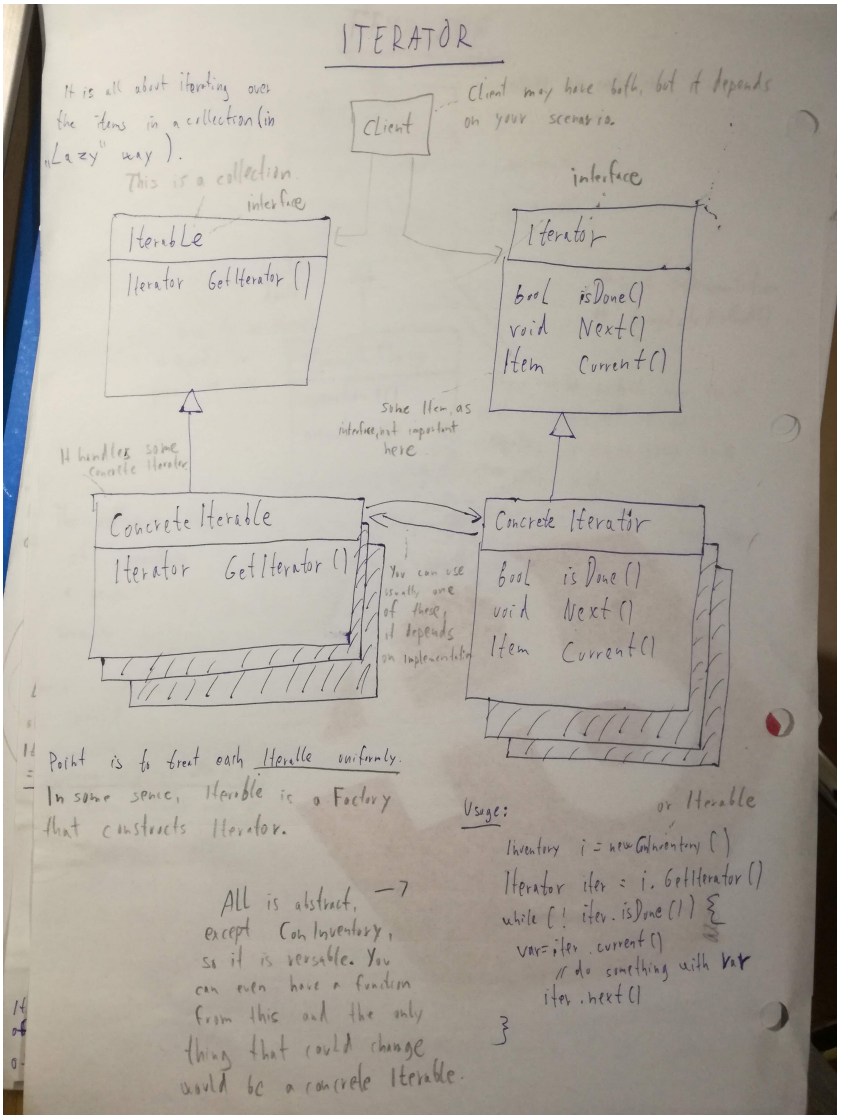

Figure 4.6: Iterator design pattern.

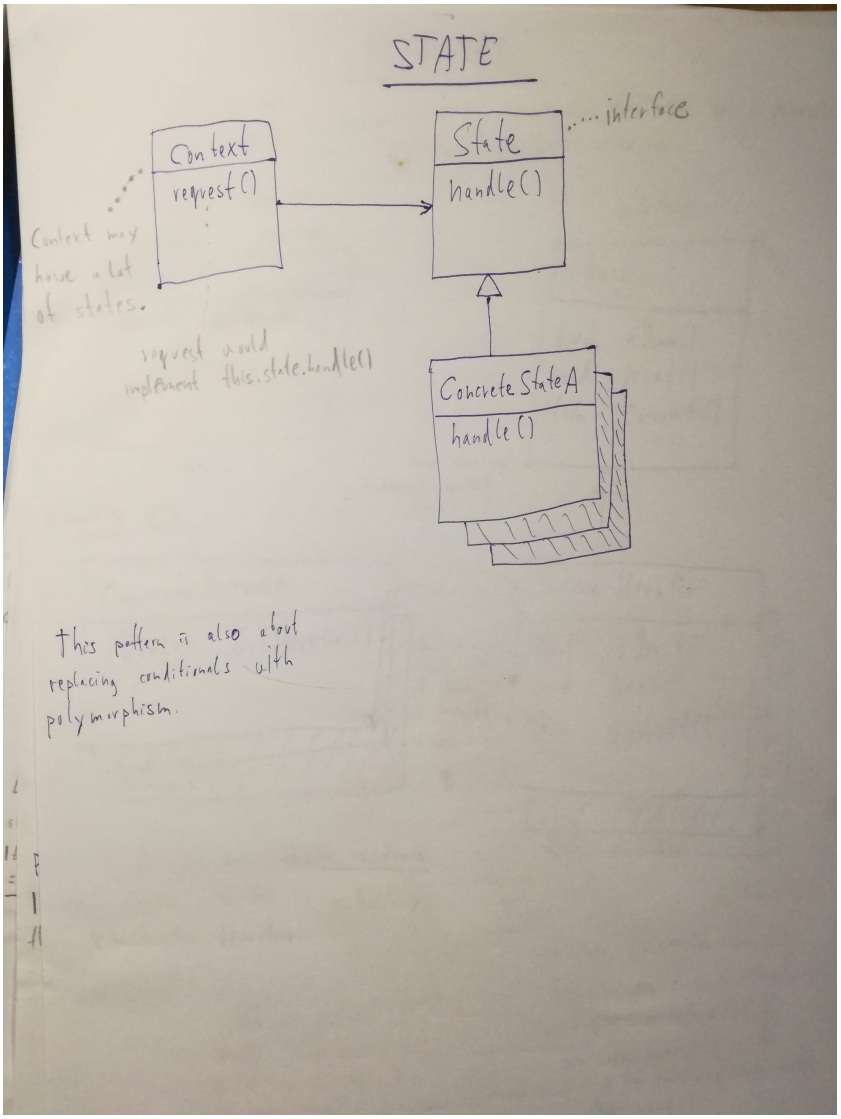

Figure 4.7: State design pattern.

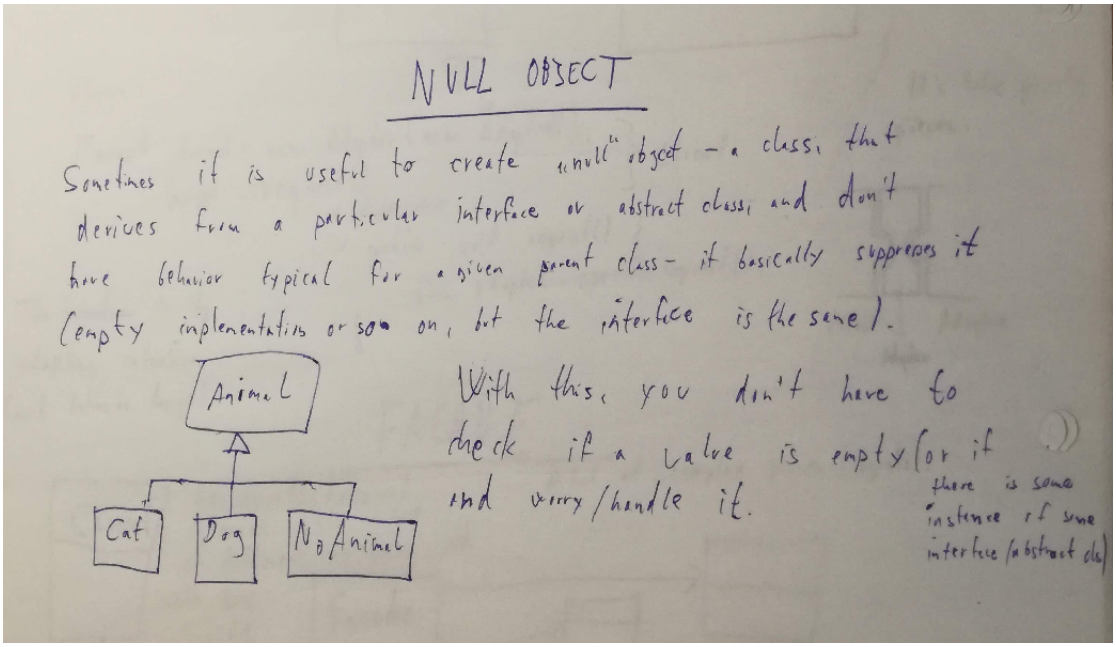

Figure 4.8: Null Object design pattern.

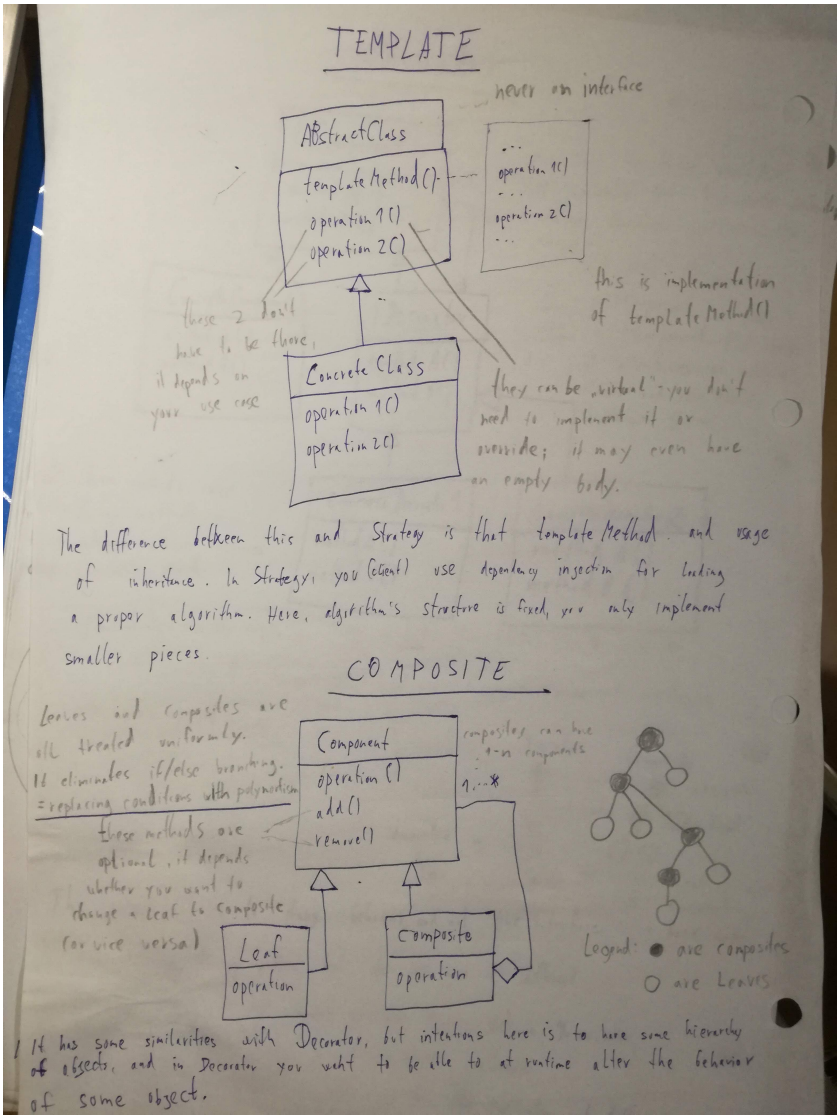

Figure 4.9: Template Method design pattern.

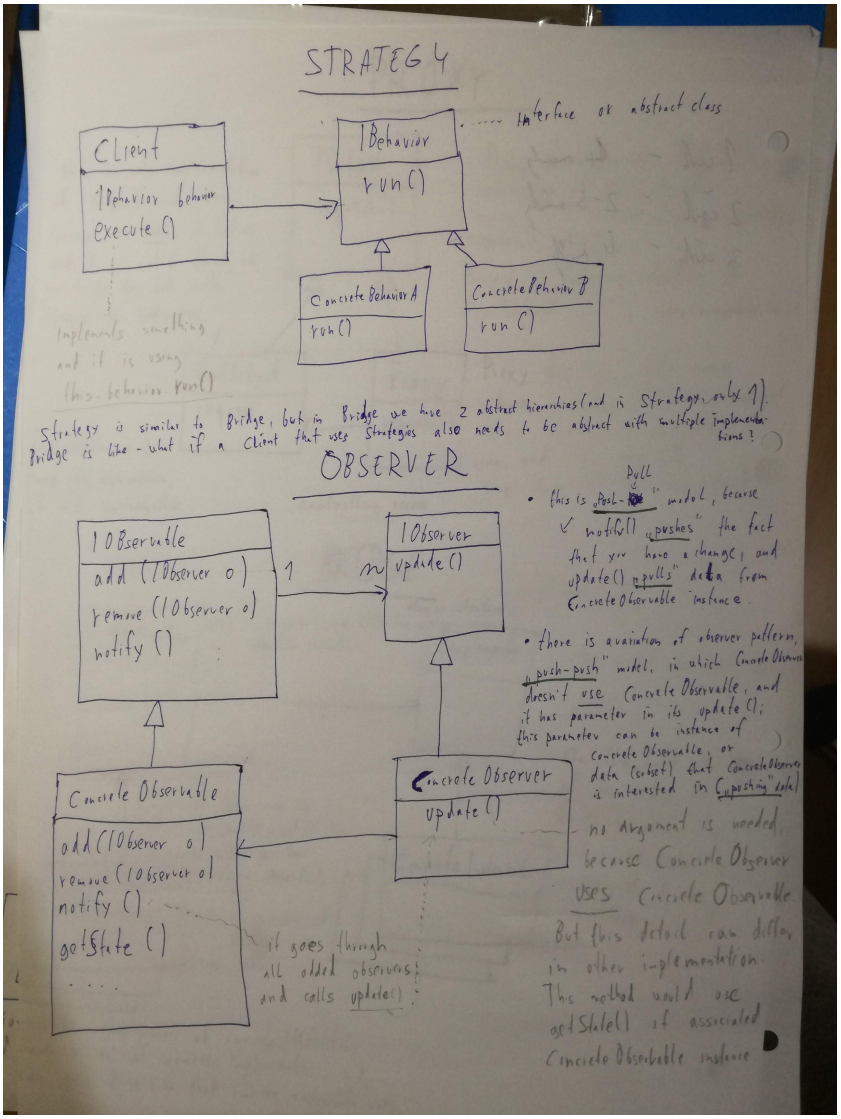

Figure 4.10: Strategy and Observer design pattern.

5 Software Architecture Patterns

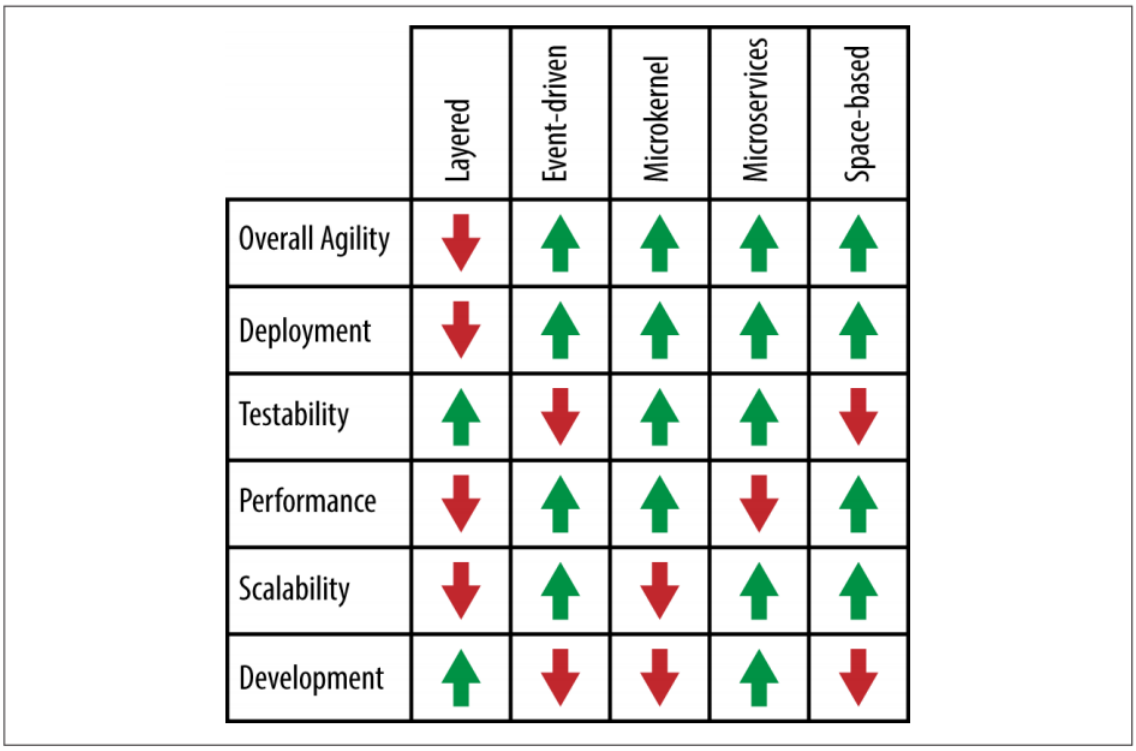

Figure 5.1: Summary of some software architecrutal patterns that are detailed in this chapter.

- The word "architecture" is often used in the context of something at a high level that is divorced from the lower-level details, whereas "design" more often seems to imply structures and decisions at a lower level. But this usage is nonsensical when you look at what a real architect does. But there is no difference between them. None at all. The low-level details and the high-level structure are all part of the same whole.
- The goal of software architecture is to minimize the human resources required to build and maintain the required system.
- The measure of design quality is simply the measure of the effort required to meet the needs of the customer. If that effort is low, and stays low throughout the

lifetime of the system, the design is good. If that effort grows with each new release, the design is bad. It's as simple as that.

- A software architect is a programmer; and continues to be a programmer. Software architects are the best programmers, and they continue to take programming tasks, while they also guide the rest of the team toward a design that maximizes productivity. Software architects may not write as much code as other programmers do, but they continue to engage in programming tasks.
- **Which kinds of decisions are premature?** Decisions that have nothing to do with the business requirements (the use cases) of the system. These include decisions about frameworks, databases, web servers, utility libraries, dependency injection, and the like. A good system architecture is one in which decisions like these are rendered ancillary and deferrable. A good system architecture does not depend on those decisions. **A good system architecture allows those decisions to be made at the latest possible moment, without significant impact.**
- Your architecture should tell readers about the system, not about the frameworks you used in your system. If you are building a health care system, then when new programmers look at the source repository, their first impression should be, "Oh, this is a heath care system."
- GUI is a detail. Web is GUI. Mobile apps are also GUI. You have to separate business logic from details! Make plugins! Database system is also just a detail. All these are just technologies (the same comes with frameworks), and technologies change! Also, you want to test business rules separately, not with GUI or database. These things can change! It is not good to have all mixed together from long term perspective. And, as an architect, you want to put details like that behind boundaries that keep them separate from your core business logic.
- **The first step in determining the initial architecture of the system is to identify the actors and use cases.** Imagine that you identified some actors in use case diagram (for example). According to the SRP, these N actors will be the N primary sources of change for the system. Every time some new feature is added, or some existing feature is changed, that step will be taken to serve one of these actors. Therefore we want to partition the system such that a change to one actor does not affect any of the other actors.
- **Eisenhower Matrix**
  - I have 2 kinds of problems, the urgent and the important. The urgent are not important, and the important are never urgent.
  - The first value of software - **behavior - is urgent but not always particularly important**. The second value of software - **architecture - is**

important but never particularly urgent. Of course, some things are both urgent and important. Other things are not urgent and not important.

- We can arrange them into priorities:
  1. Urgent and important
  2. Not urgent and important
  3. Urgent and not important
  4. Not urgent and not important
- Sometimes managers fail to separate those features that are urgent but not important from those features that truly are urgent and important. This failure then leads to ignoring the important architecture of the system in favor of the unimportant features of the system. It is the responsibility of the software development team to assert the importance of architecture over the urgency of features. (!)

- **A good architecture must support:**
  - **The development of the system**
    - The reason why so many systems lack good architecture: They were begun with none, because the team was small and did not want the impediment of a superstructure.
    - On the other hand, a system being developed by five different teams, each of which includes seven developers, cannot make progress unless the system is divided into well-defined components with reliably stable interfaces.
  - **The deployment of the system**
    - To be effective, a software system must be deployable. The higher the cost of deployment, the less useful the system is. A goal of a software architecture, then, should be to make a system that can be easily deployed with a single action.
    - Unfortunately, deployment strategy is seldom considered during initial development. This leads to architectures that may make the system easy to develop, but leave it very difficult to deploy. For example, in the early development of a system, the developers may decide to use a "micro-service architecture." They may find that this approach makes the system very easy to develop since the component boundaries are very firm and the interfaces relatively stable. However, when it comes time to deploy the system, they may discover that the number of micro-services has become daunting; configuring the connections between them, and the timing of their initiation, may also turn out to be a huge source of errors.

  * A good architecture does not rely on dozens of little configuration scripts and property file tweaks. It does not require manual creation of directories or files that must be arranged just so. A good architecture helps the system to be immediately deployable after build. This is achieved through the proper partitioning and isolation of the components of the system, including those master components that tie the whole system together and ensure that each component is properly started, integrated, and supervised.

- **The operation of the system**
  * The impact of architecture on system operation tends to be less dramatic than the impact of architecture on development, deployment, and maintenance.
  * Almost any operational difficulty can be resolved by throwing more hardware at the system without drastically impacting the software architecture.
- **The maintenance of the system**
  * Of all the aspects of a software system, maintenance is the most costly.
  * The never-ending parade of new features and the inevitable trail of defects and corrections consume vast amounts of human resources.
- **Keeping options open**
  * Software has two types of value: the value of its behavior and the value of its structure. The second of these is the greater of the two because it is this value that makes software soft. Software was invented because we needed a way to quickly and easily change the behavior of machines. But that flexibility depends critically on the shape of the system, the arrangement of its components, and the way those components are interconnected.
  * The way you keep software soft is to leave as many options open as possible, for as long as possible. (!)
  * If you can develop the high-level policy without committing to the details that surround it, you can delay and defer decisions about those details for a long time. And the longer you wait to make those decisions, the more information you have with which to make them properly.
  * This also leaves you the option to try different experiments. If you have a portion of the high-level policy working, and it is agnostic about the database, you could try connecting it to several different databases to check applicability and performance. The same is true with web systems, web frameworks, or even the web itself.
  * What if the decisions have already been made by someone else? What if your company has made a commitment to a certain database, or a

certain web server, or a certain framework? A good architect pretends that the decision has not been made, and shapes the system such that those decisions can still be deferred or changed for as long as possible.

- There are many abstract layers in an architecture (see figure below).

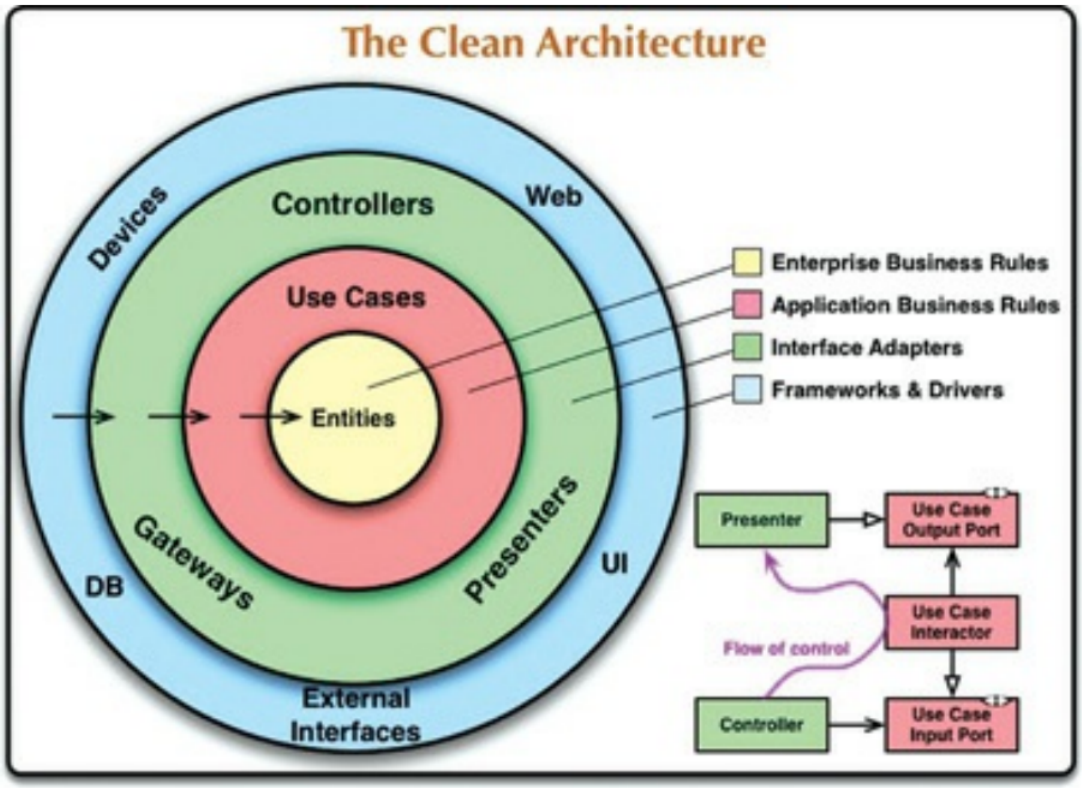

Figure 5.2: The concentric circles in figure below represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies. Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned by the code in an inner circle. That includes functions, classes, variables, or any other named software entity.

  - **Entity Layer**. It can be an object with methods, or it can be a set of data structures and functions. It doesn't matter so long as the entities can be used by many different applications in the enterprise. If you don't have an enterprise and are writing just a single application, then these entities are the business objects of the application. They encapsulate the most general and high-level rules. They are the least likely to change when something external changes.

  - **Use Case Layer**. It contains application-specific business rules. It encapsulates and implements all of the use cases of the system. These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their Critical Business Rules to achieve the goals of the use case. We do not expect changes in this layer to affect the entities. We also do not expect this layer to be affected by changes to externalities such as the database, the UI, or any of the common frameworks. The use cases layer is isolated from such concerns.
  - **Interface Adapters Layer.** It is a set of adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as the database or the web.
  - **Frameworks and Drivers Layer**. It is generally composed of frameworks and tools such as the database and the web framework. Generally you don't write much code in this layer, other than glue code that communicates to the next circle inward. The frameworks and drivers layer is where all the details go. The web is a detail. The database is a detail. We keep these things on the outside where they can do little harm.

- Architectural boundaries exist everywhere. We, as architects, must be careful to recognize when they are needed. We also have to be aware that such boundaries, when fully implemented, are expensive. At the same time, we have to recognize that when such boundaries are ignored, they are very expensive to add in later—even in the presence of comprehensive test-suites and refactoring discipline.

- O Software Architect, you must see the future. You must guess - intelligently. You must weigh the costs and determine where the architectural boundaries lie, and which should be fully implemented, and which should be partially implemented, and which should be ignored. But this is not a one-time decision. You don't simply decide at the start of a project which boundaries to implement and which to ignore. Rather, you watch. You pay attention as the system evolves. You note where boundaries may be required, and then carefully watch for the first inkling of friction because those boundaries don't exist.

- In every system, there is at least one component that creates, coordinates, and oversees the others. Let's call it Main. It is the initial entry point of the system. Nothing, other than the operating system, depends on it. Its job is to create all the Factories, Strategies, and other global facilities, and then hand control over to the high-level abstract portions of the system. Think of Main (for example function) as a plugin to the application a plugin that sets up the initial conditions and configurations, gathers all the outside resources, and then hands control over to the high-level policy of the application. Since it is a plugin, it is possible to have many Main components, one for each configuration of your application. For example, you could have a Main plugin for Dev, another for Test, and yet another for Production. You could also have a Main plugin for each country you deploy to.

Component Principles

- If the SOLID principles tell us how to arrange the bricks into walls and rooms, then the component principles tell us how to arrange the rooms into buildings. Large software systems, like large buildings, are built out of smaller components.
- Regardless of how they are eventually deployed, well-designed components always retain the ability to be independently deployable and, therefore, independently developable.

Component Cohesion

- There are 3 principles of component cohesion. They tend to fight each other (see figure below). The REP and CCP are inclusive principles: Both tend to make components larger. The CRP is an exclusive principle, driving components to be smaller. It is the tension between these principles that good architects seek to resolve. A good architect finds a position in that tension triangle that meets the current concerns of the development team, but is also aware that those concerns will change over time. For example, early in the development of a project, the CCP is much more important than the REP, because develop-ability is more important than reuse.

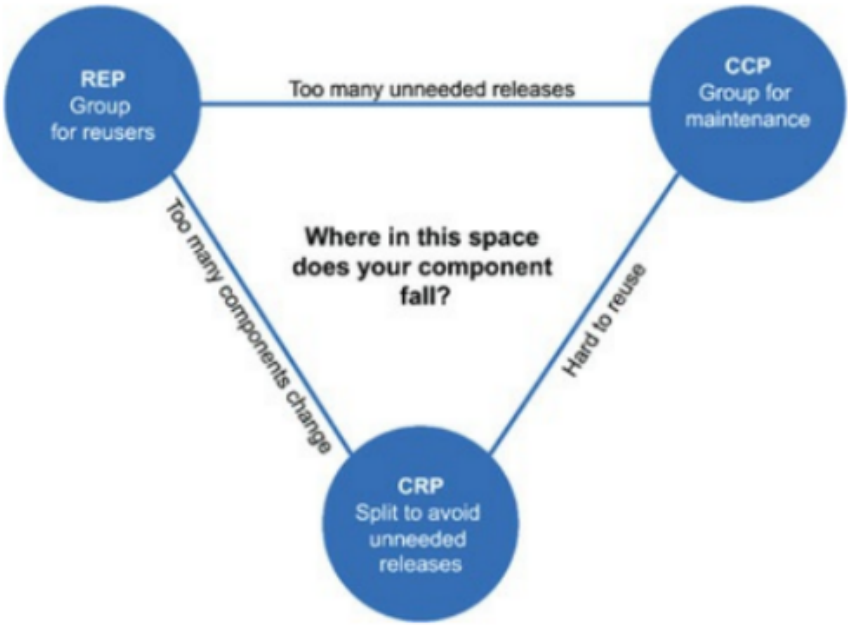

Figure 5.3: Cohension principles tension diagram. It shows how the 3 principles interact with each other. The edges of the diagram describe the cost of abandoning the principle on the opposite vertex.

- Generally, projects tend to start on the right hand side of the triangle, where the only sacrifice is reuse. As the project matures, and other projects begin to draw from it, the project will slide over to the left. This means that the component structure of a project can vary with time and maturity.
- The balance is almost always dynamic. That is, the partitioning that is appropriate today might not be appropriate next year. As a consequence, the composition of the components will likely jitter and evolve with time as the focus of the project changes from develop-ability to reusability.

The Reuse/Release Equivalence Principle (REP)

- Group for reusers.
- People who want to reuse software components cannot, and will not, do so unless those components are tracked through a release process and are given release numbers.
- From a software design and architecture point of view, this principle means that the classes and modules that are formed into a component must belong to a cohesive group. The component cannot simply consist of a random hodgepodge of classes and modules; instead, there must be some overarching theme or purpose that those modules all share.
- Classes and modules that are grouped together into a component should be releasable together. The fact that they share the same version number and the same release tracking, and are included under the same release documentation, should make sense both to the author and to the users.

The Common Closure Principle (CCP)

- Group for maintenance. Classes that change together are packaged together.
- This is the Single Responsibility Principle restated for components. It says that a component should not have multiple reasons to change.
- For most applications, maintainability is more important than reusability. If the code in an application must change, you would rather that all of the changes occur in one component, rather than being distributed across many components. If changes are confined to a single component, then we need to redeploy only the one changed component. Other components that don't depend on the changed component do not need to be re-validated or redeployed.
- The CCP prompts us to gather together in one place all the classes that are likely to change for the same reasons. If two classes are so tightly bound, either physically or conceptually, that they always change together, then they belong in the same

component. This minimizes the workload related to releasing, re-validating, and redeploying the software.

- This principle is closely associated with the Open Closed Principle (OCP). Indeed, it is "closure" in the OCP sense of the word that the CCP addresses. The OCP states that classes should be closed for modification but open for extension. Because 100% closure is not attainable, closure must be strategic. We design our classes such that they are closed to the most common kinds of changes that we expect or have experienced.
- The CCP amplifies this lesson by gathering together into the same component those classes that are closed to the same types of changes. Thus, when a change in requirements comes along, that change has a good chance of being restricted to a minimal number of components.
- The CCP tells us to separate classes into different components, if they change for different reasons. Both principles can be summarized by the following sound bite: Gather together those things that change at the same times and for the same reasons. Separate those things that change at different times or for different reasons.

The Common Reuse Principle (CRP)

- Split to avoid unneeded releases. Classes that are used together are packaged together.
- Don't force users of a component to depend on things they don't need.
- The Common Reuse Principle (CRP) is yet another principle that helps us to decide which classes and modules should be placed into a component. It states that classes and modules that tend to be reused together belong in the same component.
- When we depend on a component, we want to make sure we depend on every class in that component. Put another way, we want to make sure that the classes that we put into a component are inseparable—that it is impossible to depend on some and not on the others. Otherwise, we will be redeploying more components than is necessary, and wasting significant effort.
- Therefore the CRP tells us more about which classes shouldn't be together than about which classes should be together. The CRP says that classes that are not tightly bound to each other should not be in the same component.

Component Coupling

- As the application continues to grow, we start to become concerned about creating reusable elements. At this point, the CRP begins to influence the composition of

the components. Finally, as cycles appear, the ADP is applied and the component dependency graph jitters and grows. If we tried to design the component dependency structure before we designed any classes, we would likely fail rather badly. We would not know much about common closure, we would be unaware of any reusable elements, and we would almost certainly create components that produced dependency cycles. Thus the component dependency structure grows and evolves with the logical design of the system.

- Any component that we expect to be volatile should not be depended on by a component that is difficult to change. Otherwise, the volatile component will also be difficult to change.
- The following principles deals with relationships between components.

The Acyclic Dependencies Principle (ADP)

- Allow no cycles in the component dependency graph. The art of architecture often involves forming the regrouped components into a directed acyclic graph. In a good architecture, the direction of those dependencies is based on the level of the components that they connect. In every case, low-level components are designed so that they depend on high-level components.
- If there are cycles in the dependency graph, such cycles make it very difficult to isolate components. Unit testing and releasing become very difficult and error prone. In addition, build issues grow geometrically with the number of modules. Also, it can be very difficult to work out the order in which you must build the components.
- It is always possible to break a cycle of components and reinstate the dependency graph as a DAG:
  1. Apply DIP - create an interface between them! See the image below.
  2. Create a new component that both classes depend on (so let's imagine that in component dependency graph, component A and component B has dependency B->A which brings the circular dependency into the whole system, because if they would be as A->B, then no circular dependency would be possible).

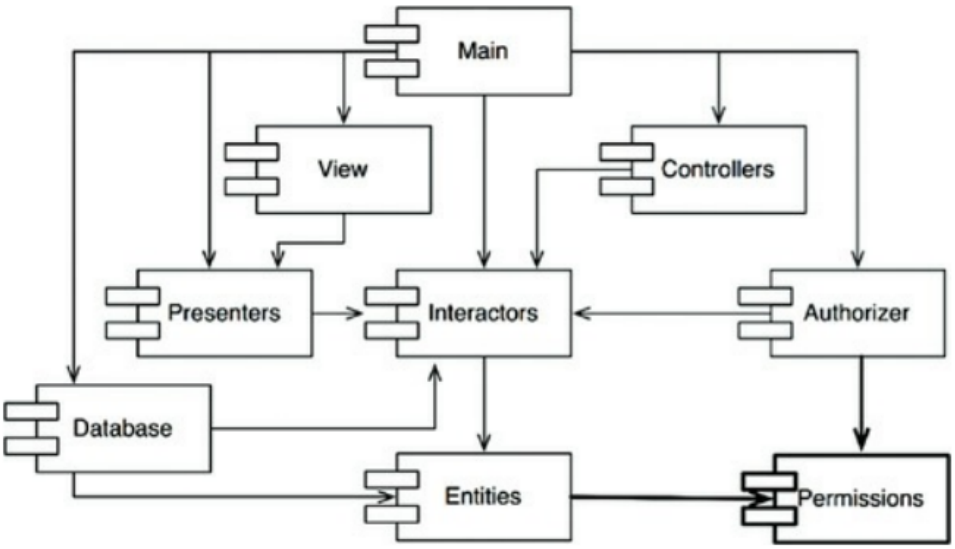

Figure 5.4: New component.

Stable Dependencies Principle (SDP)

- Depend in the direction of stability.
- We ensure that modules that are intended to be easy to change are not depended on by modules that are harder to change. One sure way to make a software component difficult to change, is to make lots of other software components depend on it. A component with lots of incoming dependencies is very stable because it requires a great deal of work to reconcile any changes with all the dependent components.

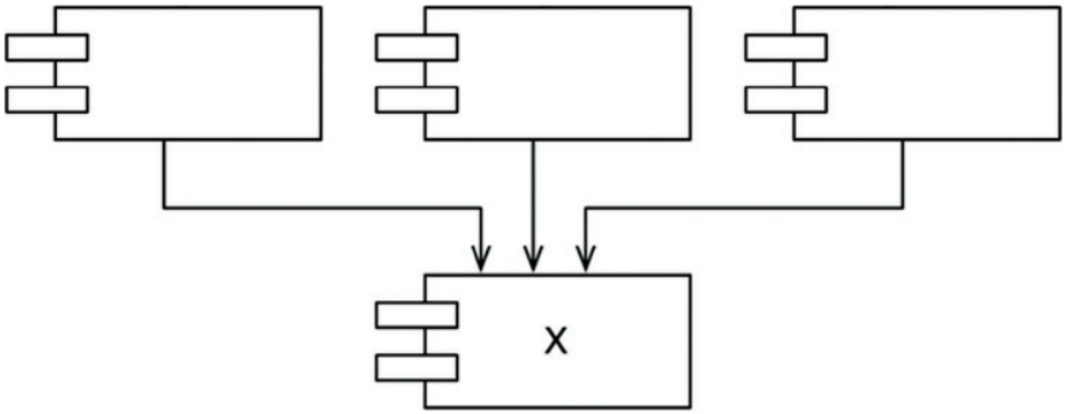

Figure 5.5: A component X is a *stable component.* Three other components depend on X, so it has three good reasons not to change. We say that X is responsible to those 3 components. Conversely, X depends on nothing, so it has no external influence to make it change. We say it is independent. Its dependents make it hard to change the component, and its has no dependencies that might force it to change.

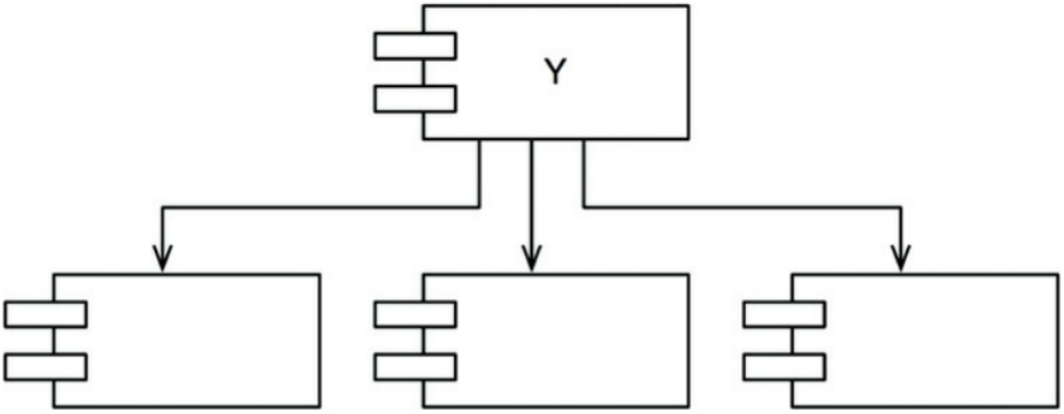

Figure 5.6: A component Y is an *unstable component.* No other components depend on Y, so we say that it is irresponsible. Y also has three components that it depends on, so changes may come from three external sources. We say that Y is dependent.

- The SDP says that the Instability metric of a component should be larger than the Instability metrics of the components that it depends on. That is, I metrics should decrease in the direction of dependency.

  - Instability can be calculated as $I = Fan\,out/(Fan\,in + Fan\,out)$. It is a number between 0 and 1 and 0 indicates maximally stable component, 1 indicates unstable component.
  - Fan_out are outgoing dependencies (number of classes inside of a given component that depend on classes outside the component),
  - Fan_in are incoming dependencies (number of classes outside this component that depend on classes within the component).

- Not all components should be stable. If all the components in a system were maximally stable, the system would be unchangeable. This is not a desirable situation. We want to design our component structure so that some components are unstable and some are stable.

The Stable Abstractions Principle (SAP)

- Abstractness increases with stability.
- This sets up a relationship between stability and abstractness. On the one hand, it says that a stable component should also be abstract so that its stability does not prevent it from being extended. On the other hand, it says that an unstable component should be concrete since it its instability allows the concrete code within it to be easily changed.
- Thus, if a component is to be stable, it should consist of interfaces and abstract classes so that it can be extended. Stable components that are extensible are flexible and do not overly constrain the architecture.

Design a Code Organization

Implementation details - this is the source of the devil. We want to get rid of it. There are 4 ways of organizing code. Let's consider, as for an example, that there is domain-related code, and then 2 implementation details - web and database.

Package by Layer

- The simplest one, traditional horizontal layered architecture where we separate our code based on what it does from a technical perspective.
- For example, in this typical layered architecture, we have one layer for the web code, one layer for our "business logic," and one layer for database (with interfaces implemented with each layer/package). So it can be used as a way to group similar types of things.
- It is a good way how to start. However, later, it is not sufficient to have a few separated packages (for example), and you have to modularize further.

- The purpose of a layered architecture is to separate code that has the same sort of function. Web stuff is separated from business logic, which is in turn separated from data access. The big problem here is that we can cheat by introducing some undesirable dependencies, yet still create a nice, acyclic dependency graph. In this architecture, it is possible to bypass the domain-related component, for example if web is using database directly. Bypassing the business logic layer is undesirable, especially if that business logic is responsible for ensuring authorized access to individual records, for example. This organization is often called a relaxed layered architecture, as layers are allowed to skip around their adjacent neighbor(s). This architecture is not ideal.

Package by Feature

- This is a vertical slicing, based on related features or domain concepts. There is for example just one package that implements all related - domain, database, and web.
- Uncle Bob often sees software development teams realize that they have problems with horizontal layering ("package by layer") and switch to vertical layering ("package by feature") instead. In his opinion, both are sub-optimal.

Ports and Adapters

- In this architecture, business/domain-focused code is independent and separate from the technical implementation details such as frameworks and databases.
- The "inside" region contains all of the domain concepts, whereas the "outside" region contains the interactions with the outside world (e.g., UIs, databases, third-party integrations). The major rule here is that the "outside" depends on the "inside" - never the other way around.
- So here would be one package for code related to a given domain, then another one to database, and another one for web. They would interact through interfaces implemented in domain package. The difference between this and "package by layer" is, that here, interfaces are implemented in a given domain package, not in each package.

Package by Components

- It's a hybrid approach to above three, with the goal of bundling all of the responsibilities related to a single coarse-grained component into a single package.
- In essence, this approach would bundle up the "business logic" and database code into a component.

- A key benefit of the "package by component" approach is that if you're writing code that needs to do something with domain-related code, there's just one place to go the given component for it. Inside the component, the separation of concerns is still maintained, so the business logic is separate from database, but that's a component implementation detail that consumers don't need to know about.

5.1 Single-Tiered / Monolithic Architecture

- User interface and data access code are combined into a single program from a single platform.
- A monolithic application is self-contained, and independent from other computing applications. The design philosophy is that the application is responsible not just for a particular task, but can perform every step needed to complete a particular function.
- Some personal finance applications are monolithic in the sense that they help the user carry out a complete task, end to end, and are private data silos rather than parts of a larger system of applications that work together.
- In its original use, the term "monolithic" described enormous main frame applications with no usable modularity. This – in combination with rapid increase in computational power and therefore rapid increase in the complexity of the problems which could be tackled by software – resulted in unmaintainable systems and the "software crisis".

5.2 Multi-Tiered / Multi-Layered Architecture

- One of the most common architecture pattern, also known as the n-tier architecture pattern.
- It is a client-server architecture in which presentation, application processing, and data management functions are physically separated.
- Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation logic or business logic).
- This architectural pattern doesn't specify the number and types of layers, but it is usually 3 or 4 of them: presentation (UI), application (service layer), business (domain layer), persistence (data access layer - the only thing communicates with database). Business and persistence are sometimes put together. But big applications may have even 5 or more layers.
- While the concepts of layer and tier are often used interchangeably, one fairly common point of view is that there is indeed a difference. This view holds that a layer is a logical structuring mechanism for the elements that make up the software solution, while a tier is a physical structuring mechanism for the system infrastructure. For example, a three-layer solution could easily be deployed on a single tier, such as a personal workstation. So basically, multi-layer architecture can be a single-tier monolith.

- One of the powerful features of the layered architecture pattern is the separation of concerns among components. Components within a specific layer deal only with logic that pertains to that layer. For example, components in the presentation layer deal only with presentation logic, whereas components residing in the business layer deal only with business logic. This type of component classification makes it easy to build effective roles and responsibility models into your architecture, and also makes it easy to develop, test, govern, and maintain applications using this architecture pattern due to well-defined component interfaces and limited component scope.

- The layers of isolation concept means that changes made in one layer of the architecture generally don't impact or affect components in other layers: the change is isolated to the components within that layer, and possibly another associated layer. This means that each layer is independent of the other layers, thereby having little or no knowledge of the inner workings of other layers in the architecture.

- The layered architecture pattern is a solid general-purpose pattern, making it a good starting point for most applications, particularly when you are not sure what architecture pattern is best suited for your application.

- Most developers and architects will resort to the de-facto standard traditional layered architecture pattern (also called the n-tier architecture), creating implicit layers by separating source-code modules into packages. Unfortunately, what often results from this practice is a collection of unorganized source-code modules that lack clear roles, responsibilities, and relationships to one another.

- **Advantages**
  - High testability
  - Easier development (it is not so complex to implement)

- **Disadvantages**
  - Architecture sinkhole anti-pattern. Be careful on this pattern, that describes the situation where requests flow through multiple layers of the architecture as simple pass-through processing with little or no logic performed within each layer.

    Every layered architecture will have at least some scenarios that fall into the architecture sinkhole anti-pattern. The key, however, is to analyze the percentage of requests that fall into this category. The 80-20 rule is usually a good practice to follow to determine whether or not you are experiencing the architecture sinkhole anti-pattern. It is typical to have around 20 percent of the requests as simple pass-through processing and 80 percent of the requests having some business logic associated with the request. However, if you find that this ratio is reversed and a majority of your requests are simple

pass-through processing, you might want to consider making some of the architecture layers open, keeping in mind that it will be more difficult to control change due to the lack of layer isolation.

- It tends to lend itself toward monolithic applications, even if you split the presentation layer and business layers into separate deployable units. So quickly responding to changes in code can be hard. Also deployment can be difficult for larger applications (there may be even a must of re-deployment). Also performance and scalability may be in danger. It is much expensive to scale.
- Applications lacking a formal architecture are generally tightly coupled, brittle, difficult to change, and without a clear vision or direction. As a result, it is very difficult to determine the architectural characteristics of the application without fully understanding the inner-workings of every component and module in the system.

5.3 Client-Server Architecture

- The server component will provide services to multiple client components. Clients request services from the server and the server provides relevant services to those clients. Furthermore, the server continues to listen to client requests.
- Requests are typically handled in separate threads on the server.

5.4 Master-Slave Pattern

- The master component distributes the work among identical slave components, and computes a final result from the results which the slaves return.

5.5 Broker Pattern

- This pattern is used to structure distributed systems with decoupled components. These components can interact with each other by remote service invocations. A broker component is responsible for the coordination of communication among components.
- Servers publish their services to a broker. Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.

5.6 Peer-to-Peer Architecture

- Individual components are known as peers. Peers may function both as a client, requesting services from other peers, and as a server, providing services to other

peers. A peer may act as a client or as a server or as both, and it can change its role dynamically with time.

- Performance depends on the number of nodes, and security is difficult to be guaranteed.
- It is highly robust in the failure of any given node.
- Highly scalable in the terms of resources and computing power.

5.7 Model-View-Controller Pattern

- This architectural pattern has 3 parts:
  - model, that contains the core functionality and data
  - view, that displays the information to the user (more than one view may be defined)
  - controller, that handles the input from the user
- Django uses this pattern.
- MVC is different from the layered architecture. Layered architecture does not allow coupling like in MVC, where MVC components could talk to each other. In contrast, layered architecture only allows message passing between layers. MVC architecture is mostly used for presentation, but layered architecture is focused on the entire system.

5.8 Representational State Transfer (REST)

- As described in a dissertation by Roy Fielding, REST is an "architectural style" that basically exploits the existing technology and protocols of the Web. RESTful is typically used to refer to web services implementing such an architecture. So RESTful service is a service layer that follows the REST architecture and HTTP protocol methods. Service layer is a protocol independent interface to our application logic. It is a common interface to your application logic that different clients like a web interface, a command line tool or a scheduled job can use.[1]
- So REST is the architecture and RESTful an adjective. Since it is not a formally defined protocol there are many opinions on the details of implementing REST APIs. However, the following five constraints must be present for any application to be considered RESTful:[2]

[1] https://stackoverflow.com/questions/1568834/whats-the-difference-between-rest-restful
[2] https://blog.feathersjs.com/design-patterns-for-modern-web-apis-1f046635215

  - **Client-server:** A client-server architecture allows a clear separation of concerns. The client is responsible for requesting and displaying the data while the server is taking care of data storage and application logic. One advantage is that both sides can be developed separately as long as the agreed-upon request format is followed.
  - **Statelessness**: Communication between client and server is stateless. This means that every client request contains all the information necessary for the server to process the request. This further reduces server complexity since no global state (other than a possibly shared database) is necessary and improves scalability since any request can be processed by any server.
  - **Caching**: Stateless client-server communication can increase server load since some information may have to be transferred several times so requests that only retrieve data should be cache-able.
  - **Layered system**: A key feature of most networked systems. In the context of REST, this means that a client can not necessarily tell if it is directly communicating with the server or an intermediate (proxy).
  - **Uniform interface**: REST defines a set of well defined operations that can be executed on a resource.
- Sometimes, using REST is not the best choice. There are many alternatives, such as Websockets, or WebRTC, but it really depends on a given situation.

5.9 Event-Driven Architecture

- It is distributed asynchronous architecture pattern used to produce highly scalable applications.
- It is highly adaptable and can be used for small applications and as well as large, complex ones. The event-driven architecture is made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events.
- It is a relatively complex pattern to implement, primarily due to its asynchronous distributed nature.
- Perhaps one of the most difficult aspects of the event-driven architecture pattern is the creation, maintenance, and governance of the event-processor component contracts.
- It consists of 2 main topologies:
  - *The Mediator*

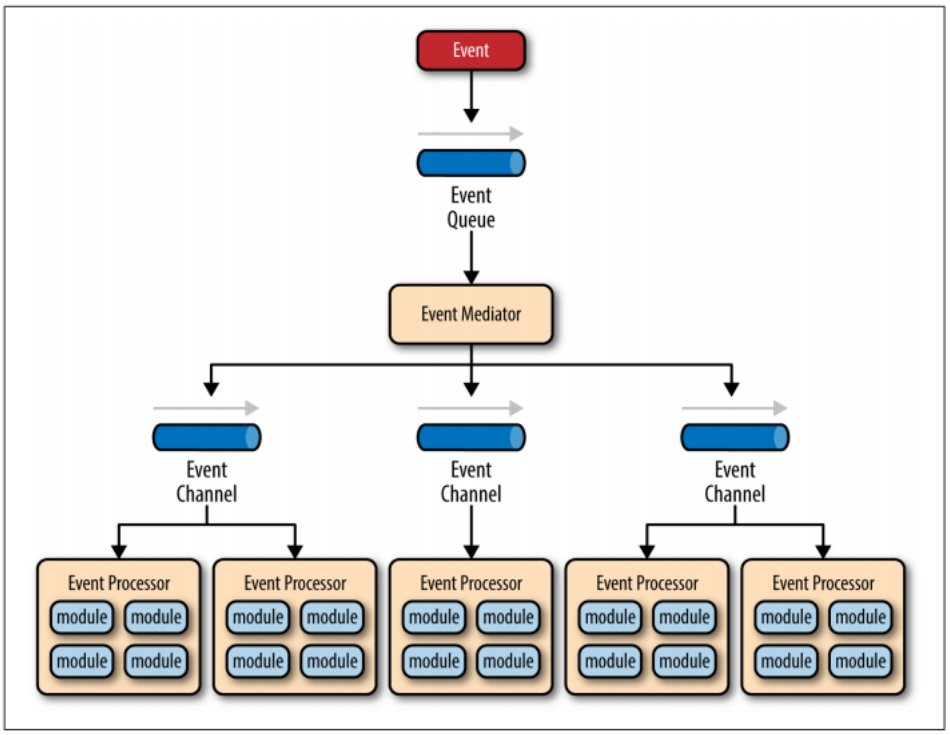

Figure 5.7: Event-driven architectural pattern with mediator topology.

* This is commonly used when you need to orchestrate multiple steps within an event through a central mediator.
* There are 4 main types of components: event queues, an event mediator, event channels, and event processors. The flow starts with a client sending an event to an event queue, which is used to transport the event to the event mediator. It receives the initial event, and orchestrates that event by sending additional asynchronous events to event channels to execute each step of the process. It is important to note that the event mediator doesn't actually perform the business logic necessary to process the initial event; rather, it knows of the steps required to process the initial event. Event processors listen on the event channels, receive the event from event mediator, and execute specific business logic to process the event. They are self-contained, independent, highly decoupled architecture components that perform a specific task in the application or system. It is important to keep in mind that in general, each event-processor component should per form a single business task and not rely on other event processors to complete its specific task. Event channels are used by the event mediator to asynchronously pass specific processing events related to each step in the initial event to the event processors. The event channels can be for example message queues.

* It is common to have anywhere from a dozen to several hundred event queues in an event-driven architecture. It can be message queue, web service endpoint, or so.
* There are 2 types of events: initial event (original event received by the mediator) and processing event (these are generated by the mediator and received by event-processing components).

– *The Broker*

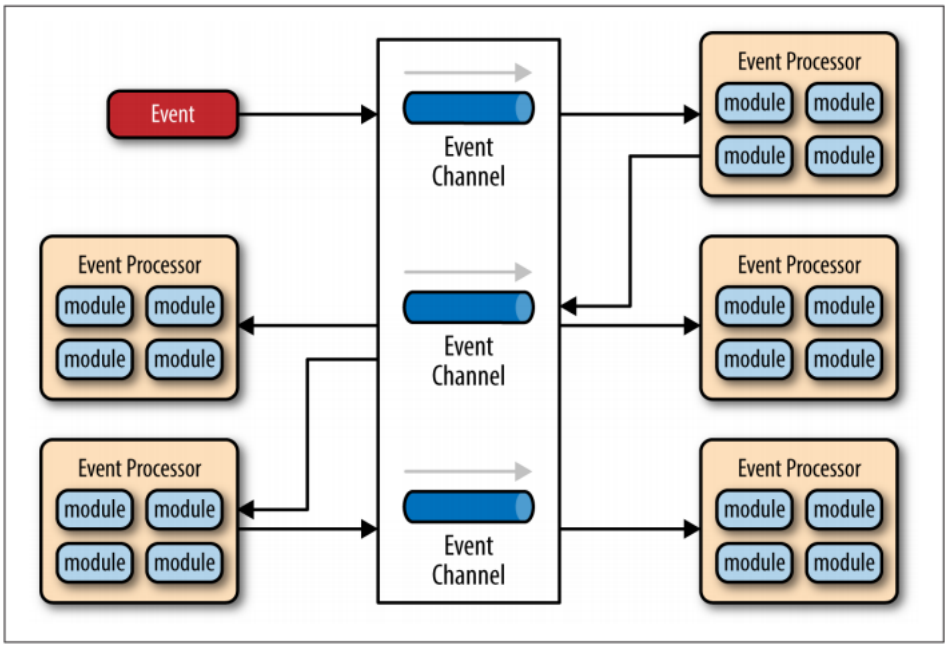

Figure 5.8: Event-driven architectural pattern with broker topology.

* This is used when you want to chain events together without the use of a central mediator. Rather, the message flow is distributed across the event processor components in a chain-like fashion through a lightweight message broker.
* This topology is useful when you have a relatively simple event processing flow and you do not want (or need) central event orchestration.
* There are 2 main types of architecture components within the broker topology:
  - *A broker component.* It can be centralized or federated and contains all of the event channels that are used within the event flow. The event channels contained within the broker component can be message queues, message topics, or a combination of both.

        - *An event processor component.* Each event processor component is responsible for processing an event and publishing a new event indicating the action it just performed.
    - Broker topology is all about the chaining of events to perform a business function. The best way to understand the broker topology is to think about it as a relay race. In a relay race, runners hold a baton and run for a certain distance, then hand off the baton to the next runner, and so on down the chain until the last runner crosses the finish line. Once an event processor hands off the event, it is no longer involved with the processing of that specific event.

- **Advantages**
  - The overall ability to respond quickly to changing environment is pretty good. Since event-processor components are single-purpose and completely decoupled from other event processor components, changes are generally isolated to one or a few event processors and can be made quickly without impacting other components.
  - Deployment is also relatively easy due to decoupled nature of the components. The broker topology tends to be easier to deploy than the mediator topology, primarily because the event mediator component is somewhat tightly coupled to the event processors: a change in an event processor component might also require a change in the event mediator, requiring both to be deployed for any given change.
  - Performance is in general very good, because of its asynchronous capabilities; in other words, the ability to perform decoupled, parallel asynchronous operations outweighs the cost of queuing and dequeuing messages.
  - Scalability is also high, because of highly independent and decoupled event processors. Each event processor can be scaled separately, allowing for fine-grained scalability.

- **Disadvantages**
  - Testing can be difficult mostly by the asynchronous nature of this pattern.
  - Development can be somewhat complicated due to the asynchronous nature of the pattern as well as contract creation and the need for more advanced error handling conditions within the code for unresponsive event processors and failed brokers.

5.10 Microkernel Architecture

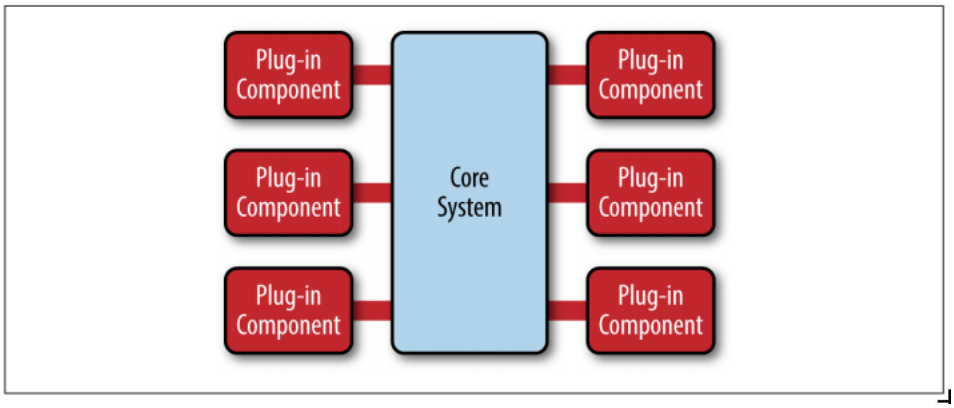

Figure 5.9: Microkernel architectural pattern.

- It is sometimes referred to as the plug-in architecture pattern.
- It is a natural pattern for implementing product-based applications (they are ones that are packaged and made available for download in versions as a typical third-party product).
- The microkernel architecture pattern allows you to add additional application features as plug-ins to the core application, providing extensibility as well as feature separation and isolation.
- It consists of two types of architecture components: a core system and plug-in modules. Application logic is divided between independent plug-in modules and the basic core system, providing extensibility, flexibility, and isolation of application features and custom processing logic.
- The core system of the microkernel architecture pattern traditionally contains only the minimal functionality required to make the system operational. Many operating systems implement the microkernel architecture pattern, hence the origin of this pattern's name. The core system needs to know about which plug-in modules are available and how to get to them. One common way of implementing this is through some sort of plug-in registry. This registry contains information about each plug-in module, including things like its name, data contract, and remote access protocol details.
- The plug-in modules are stand-alone, independent components that contain specialized processing, additional features, and custom code that is meant to enhance or extend the core system to produce additional business capabilities. Generally, plug-in modules should be independent of other plug-in modules, but you can certainly design plug-ins that require other plug-ins to be present. Either way, it is

important to keep the communication between plug-ins to a minimum to avoid dependency issues. Plug-in modules can be connected to the core system through a variety of ways, including OSGi (open service gateway initiative), messaging, web services, or even direct point-to-point binding (i.e., object instantiation). The architecture pattern itself does not specify any of these implementation details, only that the plug-in modules must remain independent from one another.

- Perhaps the best example of the microkernel architecture is the Eclipse IDE. Downloading the basic Eclipse product provides you little more than a fancy editor. However, once you start adding plug-ins, it becomes a highly customizable and useful product. Internet browsers are another common product example using the microkernel architecture: viewers and other plug-ins add additional capabilities that are not otherwise found in the basic browser (i.e., core system).
- **Advantages**
  - One great thing about the microkernel architecture pattern is that it can be embedded or used as part of another architecture pattern. For example, if this pattern solves a particular problem you have with a specific volatile area of the application, you might find that you can't implement the entire architecture using this pattern. In this case, you can embed the microservices architecture pattern in another pattern you are using (e.g., layered architecture).
  - The overall ability to respond quickly to changing environment is pretty good. Changes can largely be isolated and implemented quickly through loosely coupled plug-in modules. In general, the core system of most microkernel architectures tends to become stable quickly, and as such is fairly robust and requires few changes over time.
  - Deployment is relatively easy. Depending on how the pattern is implemented, the plug-in modules can be dynamically added to the core system at runtime (e.g., hot-deployed), minimizing downtime during deployment.
  - This pattern is relatively easy to test. Plug-in modules can be tested in isolation and can be easily mocked by the core system to demonstrate or prototype a particular feature with little or no change to the core system.
  - This pattern does not naturally lend itself to high-performance applications, in general, most applications built using the microkernel architecture pattern perform well because you can customize and streamline applications to only include those features you need.
- **Disadvantages**
  - Because most microkernel architecture implementations are product based and are generally smaller in size, they are implemented as single units and hence not highly scalable. Depending on how you implement the plug-in modules, you can sometimes provide scalability at the plug-in feature level, but overall this pattern is not known for producing highly scalable applications.

– The microkernel architecture requires thoughtful design, and it is rather complex to implement. Internal plug-in registries, plug-in granularity, and the wide choices available for plug-in connectivity all contribute to the complexity involved with implementing this pattern.

5.11 Space-Based Architecture

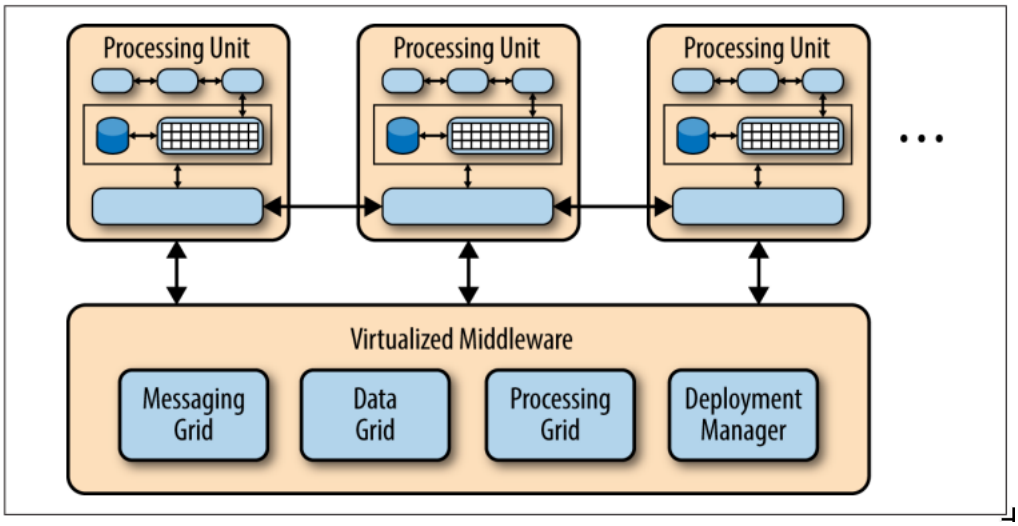

Figure 5.10: Space-based architecture pattern.

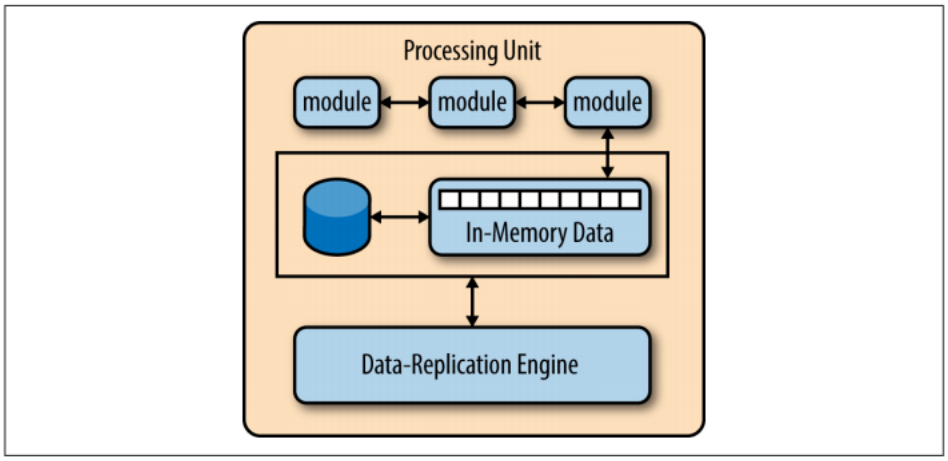

Figure 5.11: A single processing unit component as a part of space-based architecture pattern.

- It is also sometimes referred to as the cloud architecture pattern.

- The space-based architecture pattern is specifically designed to address and solve scalability and concurrency issues. It is also a useful architecture pattern for applications that have variable and unpredictable concurrent user volumes. Solving the extreme and variable scalability issue architecturally is often a better approach than trying to scale out a database or retrofit caching technologies into a non-scalable architecture.
- Most applications that fit into this pattern are standard websites that receive a request from a browser and perform some sort of action.
- High scalability is achieved by removing the central database constraint and using replicated in-memory data grids instead. Application data is kept in memory and replicated among all the active processing units. Processing units can be dynamically started up and shut down as user load increases and decreases, thereby addressing variable scalability. Because there is no central database, the database bottleneck is removed, providing near-infinite scalability within the application.
- Although the space-based architecture pattern does not require a centralized data-store, one is commonly included to perform the initial in-memory data grid load and asynchronously persist data updates made by the processing units.
- There are 2 primary components within this architecture pattern:
  - *A processing unit*
    - This component contains the application components (or portions of the application components). This includes web-based components as well as backend business logic. The contents of the processing unit varies based on the type of application—smaller web-based applications would likely be deployed into a single processing unit, whereas larger applications may split the application functionality into multiple processing units based on the functional areas of the application.
    - It typically contains the application modules, along with an in-memory data grid and an optional asynchronous persistent store for fail-over.
    - It also contains a replication engine that is used by the virtualized middleware to replicate data changes made by one processing unit to other active processing units.
  - *A virtualized middleware*
    - It contains components that control various aspects of data synchronization and request handling. Included in the virtualized middleware are the messaging grid, data grid, processing grid, and deployment manager.
    - It is essentially the controller for the architecture and manages requests, sessions, data replication, distributed request processing, and process-unit deployment.

  - It basically contains 4 components:
    - *Messaging grid.* The messaging grid manages input request and session information. When a request comes into the virtualized-middleware component, the messaging-grid component determines which active processing components are available to receive the request and forwards the request to one of those processing units. The complexity of the messaging grid can range from a simple round-robin algorithm to a more complex next-available algorithm that keeps track of which request is being processed by which processing unit.
    - *Data grid.* It is perhaps the most important and crucial component in this pattern. The data grid interacts with the data replication engine in each processing unit to manage the data replication between processing units when data updates occur. Since the messaging grid can forward a request to any of the processing units available, it is essential that each processing unit contains exactly the same data in its in-memory data grid.
    - *Processing grid.* It is an optional component within the virtualized middleware that manages distributed request processing when there are multiple processing units, each handling a portion of the application.
    - *Deployment manager.* This component continually monitors response times and user loads, and starts up new processing units when load increases, and shuts down processing units when the load decreases. It is a critical component to achieving variable scalability needs within an application.

- **Advantages**
  - It is a good architecture choice for smaller web-based applications with variable load.
  - The overall ability to respond quickly to changing environment is pretty good. Because processing units (deployed instances of the application) can be brought up and down quickly, applications respond well to changes related to an increase or decrease in user load (environment changes). Architectures created using this pattern generally respond well to coding changes due to the small application size and dynamic nature of the pattern.
  - Although space-based architectures are generally not decoupled and distributed, they are dynamic, and sophisticated cloud-based tools allow for applications to easily be "pushed" out to servers, simplifying deployment.
  - High performance is achieved through the in-memory data access and caching mechanisms build into this pattern.

  - High scalability come from the fact that there is little or no dependency on a centralized database, therefore essentially removing this limiting bottleneck from the scalability equation.

- **Disadvantages**
  - The space-based architecture pattern is a complex and expensive pattern to implement.
  - It is not well suited for traditional large-scale relational database applications with large amounts of operational data.
  - Achieving very high user loads in a test environment is both expensive and time consuming, making it difficult to test the scalability aspects of the application.
  - Sophisticated caching and in-memory data grid products make this pattern relatively complex to develop, mostly because of the lack of familiarity with the tools and products used to create this type of architecture. Furthermore, special care must be taken while developing these types of architectures to make sure nothing in the source code impacts performance and scalability.

5.12 Service-Oriented Architecture (SOA)

- One thing all service-based architectures have in common is that they are generally distributed architectures, meaning that service components are accessed remotely through some sort of remote access protocol (such as REST, AMQP, SOAP, and so on).

- Distributed architectures offer significant advantages over monolithic and layered-based architectures, including better scalability, better decoupling, and better control over development, testing, and deployment.

- Maintaining service contracts, choosing the right remote-access protocol, dealing with unresponsive or unavailable services, securing remote services, and managing distributed transactions are just a few of the many complex issues you have to address when creating service-based architectures.

- Rather than use ACID transactions, service-based architectures rely on BASE transactions. BASE is a family of styles that include basic availability, soft state, and eventual consistency. Distributed applications relying on BASE transactions strive for eventual consistency in the database rather than consistency at every transaction.

- Service components within a microservices architecture are generally single-purpose services that do one thing really, really well. With SOA, service components can

range in size anywhere from small application services to very large enterprise services. In fact, it is common to have a service component within SOA represented by a large product or even a subsystem.

- Service granularity has the most potential impact on your choice of which architecture pattern is best suited for your situation !! SOA vs Microservices. If you are able to break down the business functionality of your application into very small, independent parts, then the microservices pattern is a likely candidate for your architecture.

- SOA is built on the concept of a share-as-much-as-possible architecture style, whereas microservices architecture is built on the concept of a share-as-little-as-possible architecture style. One way to achieve a bounded context and minimize dependencies in extreme cases is to violate the Don't Repeat Yourself (DRY) principle and replicate common functionality across services to achieve total independence. Another way is to compile relatively static modules into shared libraries that service components can use in either a compile-time or runtime binding.

- SOA, being a share-as-much-as-possible architecture, relies on both service orchestration and service choreography to process business requests.

- In some cases you might find that the microservices pattern is a good initial architecture choice in the early stages of your business, but as the business grows and matures, you begin to need capabilities such as complex request transformation, complex orchestration, and heterogeneous systems integration. In these situations you will likely turn to the SOA pattern to replace your initial microservices architecture. Of course, the opposite is true as well—you may have started out with a large, complex SOA architecture, only to find that you didn't need all of those powerful capabilities that it supports after all.

- If you find yourself in a heterogeneous environment where you need to integrate several different types of systems or services using different protocols, chances are that you will need to look toward SOA rather than microservices. However, if all of your services can be exposed and accessed through the same remote-access protocol (e.g., REST), then microservices can be the right choice.

5.13 Microservices Pattern

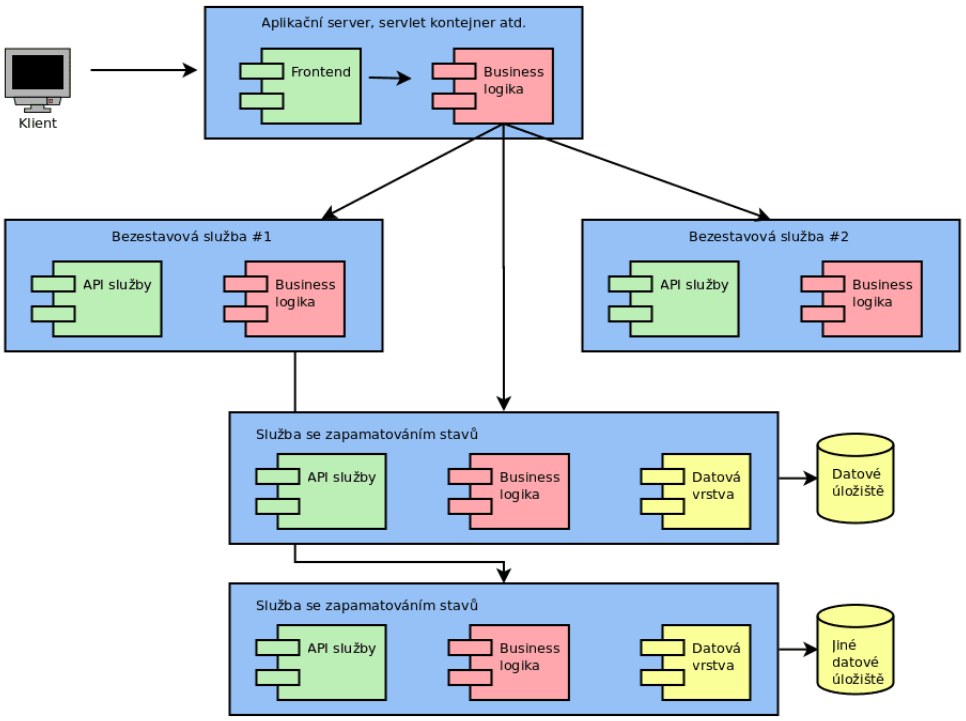

Figure 5.12: Microservices architecture [CZ].

- Each service has only one job (Single Responsibility Principle).[3]
- More complex results are retrieved by combining services.
- Database per microservice (for metadata for example).
- Martin Fowler's definition: *"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API."*
- Again, Martin Fowler: *"Don't even consider microservices unless you have a system that's too complex to manage as a monolith."*
- Martin Fowler[4]:
  - *"Almost all the successful microservice stories have started with a monolith that grew too big and was broken up."*

[3] http://microservices.io/patterns/microservices.html
[4] https://martinfowler.com/bliki/MonolithFirst.html

  - *"Almost all the cases I've heard of a system that was built as a microservice system from scratch, it has ended up in serious trouble."*

- The microservices architecture style naturally evolved from two main sources: monolithic applications developed using the layered architecture pattern and distributed applications developed through the service-oriented architecture pattern.

- While the SOA pattern is very powerful and offers unparalleled levels of abstraction, heterogeneous connectivity, service orchestration, and the promise of aligning business goals with IT capabilities, it is nevertheless complex, expensive, ubiquitous, difficult to understand and implement, and is usually overkill for most applications. The microservices architecture style addresses this complexity by simplifying the notion of a service, eliminating orchestration needs, and simplifying connectivity and access to service components. In the mid-2000's, service-oriented architecture (SOA) took the IT industry by storm. SOA can be a big, expensive, complicated architecture style that took too long to design and implement. Microservices architecture holds the promise of being able to address some of the problems associated with large, complex SOAs as well as the problems found with big, bloated monolithic applications.

- They are about optimizing for speed. Also about how to faster develop and deploy. Also the key is to manage (and reduce) dependencies.

- The microservices pattern is better suited for smaller, well partitioned web-based systems rather than large-scale enterprise wide systems. The lack of a mediator (messaging middleware) is one of the factors that makes it ill-suited for large-scale complex business application environments.

- *"Start with a small number of larger services first."*, Sam Newman. Just watch out for transaction issues and too much inter-service communication, particularly with microservices. These are good indicators that your services might be too fine-grained.

- Microservices architecture favors service choreography over service orchestration, primarily because the architecture topology lacks a centralized middleware component.
  - Service orchestration refers to the coordination of multiple services through a centralized mediator.
  - Service choreography refers to the coordination of multiple service calls without a central mediator. The term inter-service communication is sometimes used in conjunction with service choreography. With service choreography, one service calls another service, which may call another service and so on, performing what is also referred to as service chaining.

- There is a lot of disadvantages[5], such as complex networking, developers need to learn a lot more stuff, lot of servers and databases to maintain (infrastructure overhead). It is especially a disadvantage in smaller teams. The whole system is fragmented, and it is needed to connect all parts. If the whole system is wrongly designed, when a single microservice when fail, the whole system will fail. Communication between processes is slower than inter-process communication (direct function calls etc).
- It is needed to have as small dependencies between each individual microservices as possible!
- So there is Monolithic Architecture as one extreme, Microservices Architecture on the other side of the extreme, and SOA as something in between.
- One security design implemented in microservices that works well is to delegate authentication to a separate service and place the responsibility for authorization in the service itself. Although this design could be modified to delegate both authentication and authorization to a separate security service, I prefer encapsulating the authorization in the service itself to avoid chattiness with a remote security service and to create a stronger bounded context with fewer external dependencies.
- SOA was "invented" before Microservices, and it is also based on techniques that splits application to individual components, that should be independent from each other and should communicate to each other. They should be ideally stateless. Microservices can be seen as even more extreme version of SOA, in which we are splitting the application even more (in SOA, there are "micromonoliths" with a single shared database), so that it is possible to develop, maintain, and deploy such components independently. In Microservices, we usually don't have application server, because it is an overkill for it. Also, in SOA, there are a lot more communication protocols and in Microservices, we are trying to have simple messages. In SOA, the application should be ideally written in 1 language, but in Microservices, there is no restriction.
- We need to constantly assess the way that we do things to ensure that we're on the right track. Even the classic PlanDo-Check-Act (PDCA) process is a variation of the feedback loop. In software (as with everything we do in life) the longer the feed-back loop, the worse the results are. And this happens because we have a limited amount of capacity for holding information in our brains, both in terms of volume and duration.
- Avoid dependencies and orchestration - one of the main challenges of the microservices architecture pattern is determining the correct level of granularity for the service components. If you find you need to orchestrate your service components from within the user interface or API layer of the application, then chances are

[5] https://www.youtube.com/watch?v=1xo-0gCVhTU

your service components are too fine-grained. Similarly, if you find you need to perform inter-service communication between service components to process a single request, chances are your service components are either too fine-grained or they are not partitioned correctly from a business functionality standpoint.

- A fairly common practice in most business applications implementing the microservices architecture pattern, trading off the redundancy of repeating small portions of business logic for the sake of keeping service components independent and separating their deployment.
- If you find that regardless of the level of service component granularity you still cannot avoid service-component orchestration, then it's a good sign that this might not be the right architecture pattern for your application. Because of the distributed nature of this pattern, it is very difficult to maintain a single transactional unit of work across (and between) service components. Such a practice would require some sort of transaction compensation framework for rolling back transactions, which adds significant complexity to this relatively simple and elegant architecture pattern. Inter-service communication, which could force undesired couplings between components, can be handled instead through a shared database.
- One final consideration to take into account is that since the micro-services architecture pattern is a distributed architecture, it shares some of the same complex issues found in the event-driven architecture pattern, including contract creation, maintenance, and government, remote system availability, and remote access authentication and authorization.
- Some might think that the discussion around microservices is about scalability. Most likely it's not (well, in Netflix or Amazon, there are different scalability requirements than in smaller companies). It's all about again improving our lead time and reducing the time between our releases. With microservices, we're trying to split a piece of this huge monolithic codebase into a smaller, well-defined, cohesive, and loosely coupled artifact.
- A very mature software deployment pipeline is an absolute requirement for any microservices architecture. Some indicators that you can use to assess pipeline maturity are the amount of manual intervention required, the amount of automated tests, the automatic provisioning of environments, and monitoring.
- Any refactoring of functionality between services is much harder than it is in a monolith. But even experienced architects working in familiar domains have great difficulty getting boundaries right at the beginning. By building a monolith first, you can figure out what the right boundaries are, before a microservices design brushes a layer of treacle over them.
- The monolith-first is also called the strangler pattern. Having a stable monolith is a good starting point because one of the hardest things in software is the identification of boundaries between the domain model—things that change together, and

things that change apart. Create wrong boundaries and you'll be doomed with the consequences of cascading changes and bugs.

- From a domain model perspective, microservices are all about boundaries: we're splitting a specific piece of our domain model that can be turned into an independently releasable artifact. With a badly defined boundary, we will create an artifact that depends too much on information confined in another microservice. We will also create another operational pain: whenever we make modifications in one artifact, we will need to synchronize these changes with another artifact. From the beginning, it's very difficult to guess which parts of the system change together and which ones change separately. However, after months, or more likely years, developers and business analysts should have a better picture of the evolution cycle of each one of the bounded contexts.

- There are many different success stories about using NoSQL databases in different contexts, and some of these contexts might fit your current enterprise context, as well. But even if it does, we still recommend that you begin your microservices journey on the safe side: using a relational database. First, make it work using your existing relational database. Once you have successfully finished implementing and integrating your first microservice, you can decide whether you (or) your project will be better served by another type of database technology.

- Using the Decentralized Data Management characteristic of microservices architectures, each one of our microservices should have its own separate database—which could possibly, again, be a relational database or not. However, a legacy monolithic relational database is very unlikely to simply migrate the tables and the data from your current schema to a new separate schema or database instance, or even a new database technology. We want to evolve our architecture as smoothly as possible: it requires baby steps and carefully considered migrations in each one of these steps to minimize disruption and, consequently, downtime. Moreover, a microservice is not an island; it requires information provided by other services, and also provides information required by other services. Therefore, we need to integrate the data between at least two separate services: one can be your old monolithic application and the other one will be your new microservice artifact.

- Sometimes you are not building microservices, but distributed monolith, which is the worst. You can ask these questions for determining the answer on what you are building (at least few of the following points):[6]
  - A change to one microservice often requires changes to other microservices
  - Deploying one microservice requires other microservices to be deployed at the same time
  - Your microservices are overly chatty

[6] https://www.simplethread.com/youre-not-actually-building-microservices/

  - The same developers work across a large number of microservices
  - Many of your microservices share a datastore
  - Your microservices share a lot of the same code or models

- Having a single, non-distributed codebase can be a huge advantage when starting out with a new system. It allows you to more easily reason about your code, allows you to more easily test your code, and it allows you to move quickly and change quickly without having to worry about orchestration between services, distributed monitoring, keeping your services in sync, eventual consistency, all of the things you'll run into with microservices.

- **Production-ready microservices, requirements**
  - Stability - Stable development cycle and deployment process.
  - Reliability - Reliable deployment process; planning, mitigating, and protecting against the failures of dependencies; reliable routing and discovery.
  - Scalability - Well-defined quantitative and qualitative growth scales, Identification of resource bottlenecks and requirements, Careful, accurate capacity planning, Scalable handling of traffic, The scaling of dependencies, and Scalable data storage.
  - Fault Tolerance and Catastrophe Prepareness
    * Potential catastrophes and failure scenarios are identified and planned for.
    * Single points of failure are identified and resolved.
    * Failure detection and remediation strategies are in place.
    * The microservice is tested for resiliency through code testing, load testing, and chaos testing.
    * Traffic is managed carefully in preparation for failure.
    * Incident and outages are handled appropriately and productively.
  - Performance
    * Proper task handling and processing.
    * Efficient utilization of resources.
  - Monitoring
    * Proper logging of all important and relevant information.
    * Useful graphical displays (dashboards) that are easily understood by any developer in the company and that accurately reflect the health of the services.
    * Alerting on key metrics that is effective and actionable.
  - Documentation

    - Thorough, updated, and centralized documentation containing all of the relevant and essential information about the microservice.
    - Organizational understanding at the developer, team, and ecosystem levels.

- **Advantages**
  - The overall ability to respond quickly to changing environment is pretty good. Due to the notion of separately deployed units, change is generally isolated to individual service components, which allows for fast and easy deployment.
  - Services are generally deployed as separate units of software, resulting in the ability to do "hot deployments" any time during the day or night.
  - Due to the separation and isolation of business functionality into independent applications, testing can be scoped, allowing for more targeted testing efforts. Also, since the service components in this pattern are loosely coupled, there is much less of a chance from a development perspective of making a change that breaks another part of the application, easing the testing burden of having to test the entire application for one small change.
  - Because the application is split into separately deployed units, each service component can be individually scaled, allowing for fine-tuned scaling of the application.
  - Because functionality is isolated into separate and distinct service components, development becomes easier due to the smaller and isolated scope. There is much less chance a developer will make a change in one service component that would affect other service components, thereby reducing the coordination needed among developers or development teams.

- **Disadvantages**
  - While you can create applications implemented from this pattern that perform very well, overall this pattern does not naturally lend itself to high-performance applications due to the distributed nature of the microservices architecture pattern.

Patterns, Recommendations, or Solutions

Pattern Topologies

- API REST-based topology, is useful for websites that expose small, self-contained individual services through some sort of API. In this topology, these fine-grained service components are typically accessed using a REST-based interface implemented through a separately deployed web-based API layer.
- Application REST-based topology, which differs from the previous one in that client requests are received through traditional web-based or fat-client business

application screens rather than through a simple API layer. User-interface layer of the application is deployed as a separate web application that remotely accesses separately deployed service components (business functionality) through simple REST-based interfaces. The service components in this topology differ from those in the API-REST-based topology in that these service components tend to be larger, more coarse-grained, and represent a small portion of the overall business application rather than fine-grained, single-action services. This topology is common for small to medium-sized business applications that have a relatively low degree of complexity.

- Centralized messaging topology, which is similar to the previous application REST-based topology except that instead of using REST for remote access, this topology uses a lightweight centralized message broker. It is vitally important when looking at this topology not to confuse it with the service-oriented architecture pattern or consider it "SOA-Lite." The lightweight message broker found in this topology does not perform any orchestration, transformation, or complex routing; rather, it is just a lightweight transport to access remote service components. The centralized messaging topology is typically found in larger business applications or applications requiring more sophisticated control over the transport layer between the user interface and the service components. The benefits of this topology over the simple REST-based topology discussed previously are advanced queuing mechanisms, asynchronous messaging, monitoring, error handling, and better overall load balancing and scalability. The single point of failure and architectural bottleneck issues usually associated with a centralized broker are addressed through broker clustering and broker federation (splitting a single broker instance into multiple broker instances to divide the message throughput load based on functional areas of the system).

Saga

- Saga solves a problem how to maintain data consistency across multiple microservices (since each have a separate database) without using distributed transactions.[7]
- A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.
- There can be choreography-based saga, and orchestration-based saga.
- However, the programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.

[7] https://microservices.io/patterns/data/saga.html

- In order to be reliable, a service must atomically update its database and publish a message/event. It cannot use the traditional mechanism of a distributed transaction that spans the database and the message broker. Instead, it must use one of the patterns listed below:
  - The Database per Service pattern creates the need for this pattern.[8] There could be also alternatives, such as tables per service, or schema per service. If database per service is implemented, then joining data together can be done on application (another service that would retrieve data from other services and connects the information together), or with Command Query Responsibility Segregation (CQRS).
  - CQRS - maintain one or more materialized views that contain data from multiple services. The views are kept by services that subscribe to events that each services publishes when it updates its data. For example, the online store could implement a query that finds customers in a particular region and their recent orders by maintaining a view that joins customers and orders. The view is updated by a service that subscribes to customer and order events.
  - Ways to atomically update state and publish messages/events:
    - *Event sourcing*[9] or
    - *Transactional Outbox*[10]
- **Event sourcing** persists the state of a business entity such an Order or a Customer as a sequence of state-changing events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic. The application reconstructs an entity's current state by replaying the events. Applications persist events in an event store, which is a database of events. The store has an API for adding and retrieving an entity's events. The event store also behaves like a message broker. It provides an API that enables services to subscribe to events. When a service saves an event in the event store, it is delivered to all interested subscribers.
  - Some entities, such as a Customer, can have a large number of events. In order to optimize loading, an application can periodically save a snapshot of an entity's current state. To reconstruct the current state, the application finds the most recent snapshot and the events that have occurred since that snapshot. As a result, there are fewer events to replay.
  - The CQRS must often be used with event sourcing.
  - **Advantages**

[8] https://microservices.io/patterns/data/database-per-service.html
[9] https://microservices.io/patterns/data/event-sourcing.html
[10] https://microservices.io/patterns/data/transactional-outbox.html

    - Event sourcing solves one of the key problems in implementing an event-driven architecture and makes it possible to reliably publish events whenever state changes.
    - Event sourcing-based business logic consists of loosely coupled business entities that exchange events. This makes it a lot easier to migrate from a monolithic application to a microservice architecture.
  - **Disadvantages**
    - Learning curve, and that the event store is difficult to query since it requires typical queries to reconstruct the state of the business entities. That is likely to be complex and inefficient. As a result, the application must use Command Query Responsibility Segregation (CQRS) to implement queries. This in turn means that applications must handle eventually consistent data.

- **Transactional outbox**
  - An alternative solution is Event sourcing.
  - Problem: How to reliably/atomically update the database and publish messages/events?
  - The main benefit is, that the service publishes high-level domain events.
  - A service that uses a relational database inserts messages/events into an outbox table as part of the local transaction. A service that uses a NoSQL database appends the messages/events to attribute of the record (e.g. document or item) being updated. A separate Message Relay process publishes the events inserted into database to a message broker.
  - It is potentially error prone since the developer might forget to publish the message/event after updating the database. Or, the Message Relay process might publish a message more than once.

Kappa Architecture

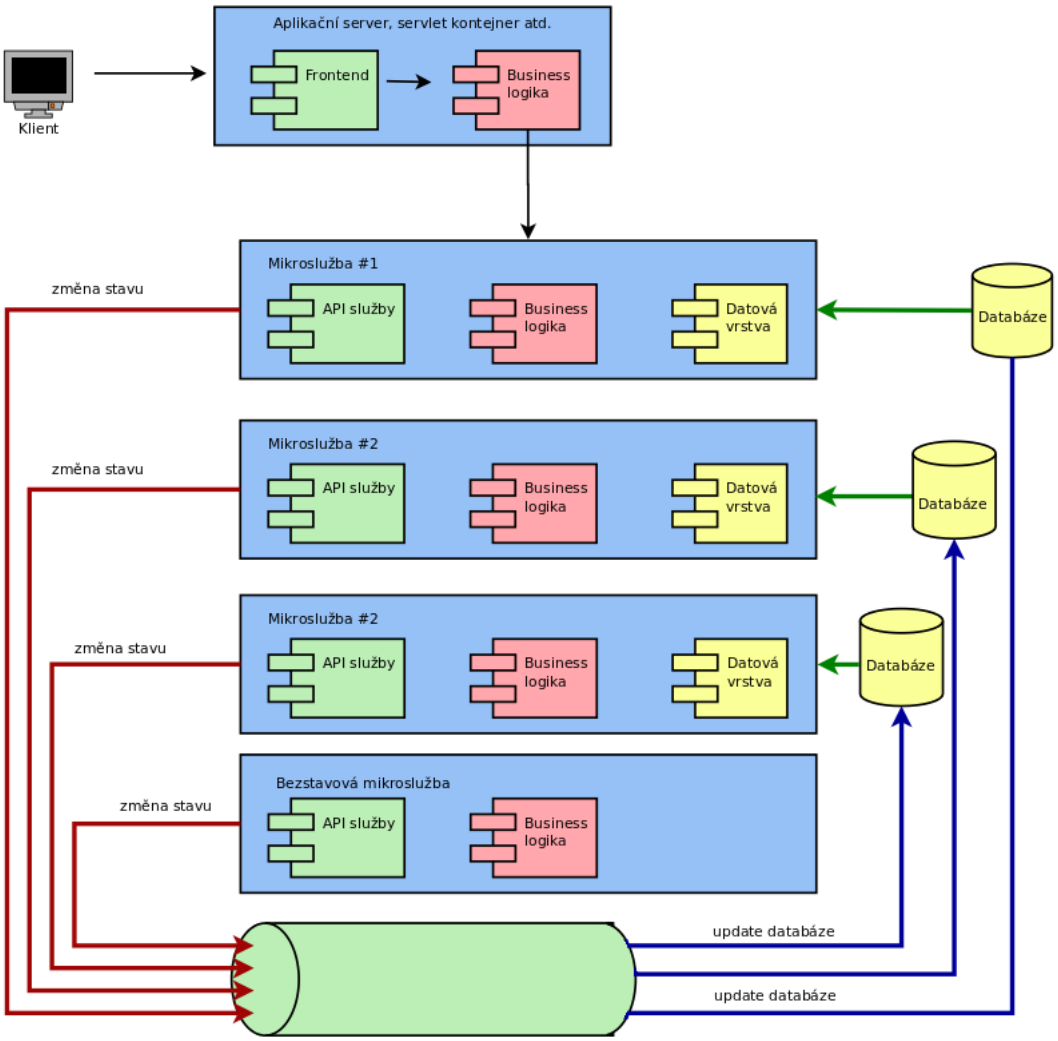

Figure 5.13: Microservices architecture with Kappa topology [CZ].

- The main problem this architecture solves is data distribution and synchronization of databases of each individual microservices.
- Database role is inverted in this architecture. Databases are not used for actual data, but just for materialized view on data. Actual data (and even historical ones) are stored in message broker (Apache Kafka used as a message broker for example). Each microservice reads information from message broker, processes this information, and applies the result on its database.

- It is based on that primary source of information about state of application are not SQL/NoSQL databases, but events recorded to streaming platform (Apache Kafka for example).
- Databases themselves are also used in this architecture, as materialized views. Every microservice that owns its database, sequentially receives each individual events and changes the content of its database based on the data in the event. So such databases have data that can be delayed.
- Besides of the streaming platform, it is still good that microservices are not communicating directly with each other, but by some intermediator, such as message broker.
- Ideally, each microservice is autonomous and it also handles change of it state. But it is good if a microservice is stateless. If shouldn't share its internal state, or any state, all it should share is API (it is kind of encapsulation).
- **Advantages**
  - Microservices are isolated, there is no need to use any orchestration.
  - If there is a new microservice added, its database will be filled with data very simply - just with filling all events from streaming platform.
  - Small blackout of some microservice does not lead to data loss, but can lead to slowing down of some work from user point of view.
  - Migration of database, changing its schema or moving to totally different database is not that difficult because of streaming platform.
  - Audit log - the whole application has it because of the streaming platform.

Lambda Architecture

- Very straightforward architecture, in which the central piece is streaming platform (such as Apache Kafka) or a message broker.
- Data acquired from broker are typically processed in several components (microservices). One component is optimized for obtaining results in real time, another one is for batch processing, another component may aggregate results from the previous component, and so on.

6 Cloud Technologies

Cloud Computing

- **Cloud computing is the on-demand delivery of compute power, database storage, applications, and other IT resources through a cloud services platform via the internet with pay-as-you-go pricing.**
- The moment you need services like storage and compute or networking, they're available immediately without any advanced contract.
- It's not just a substitute for what you have on-premises. Look at the idea of IT as a series of actions: some of them are important to your business, and some of them are common everywhere. Everybody needs compute, storage, and identity management. AWS exists to eliminate those undifferentiated heavy lifting tasks that your IT department needs, that everyone's IT department needs. This means your business can spend time working on what is strategically unique to you rather than repetitive common tasks that everyone has to do.
- AWS has a high level of security with many security features written especially for your operations team. There are automation suites designed to deploy all of your applications, all of your databases, all of your environments automatically. Also there is a database as a service. A lot of other services, such as virtual reality, game development, IoT, machine learning, the list is bigger than 140 items.

Types of Cloud Computing

- There is a range of deployment models, from all on-premises to fully deployed in the cloud. Many users begin with a new project in the cloud, and they might integrate some on-premises applications with these new projects in a hybrid architecture. They might decide to keep some legacy systems on-premises. Over time, they might migrate more and more of their infrastructure to the cloud, and they might eventually reach an all-in-the-cloud deployment.[1]
- There are 3 main types of **Cloud Computing models**
  - *Infrastructure as a Service (IaaS):* this typically provides access to networking features, computers (virtual or on dedicated hardware), and data storage space. IaaS provides you with the highest level of flexibility and management control over your IT resources and is most similar to existing IT resources that many IT departments and developers are familiar with today.

[1] https://aws.amazon.com/types-of-cloud-computing/

  - *Platform as a Service (PaaS):* this removes the need for organizations to manage the underlying infrastructure (usually hardware and operating systems) and allow you to focus on the deployment and management of your applications. This helps you be more efficient as you don't need to worry about resource procurement, capacity planning, software maintenance, patching, or any of the other undifferentiated heavy lifting involved in running your application.
  - *Software as a Service (SaaS):* this provides a completed product that is run and managed by the service provider. In most cases, people referring to SaaS are referring to end-user applications. With a SaaS offering you do not have to think about how the service is maintained or how the underlying infrastructure is managed; you only need to think about how you will use that particular piece software.

- There are 3 types of **Cloud Computing Deployment Models**
  - *Cloud:* A cloud-based application is fully deployed in the cloud and all parts of the application run in the cloud. Applications in the cloud have either been created in the cloud or have been migrated from an existing infrastructure to take advantage of the benefits of cloud computing. Cloud-based applications can be built on low-level infrastructure pieces or can use higher level services that provide abstraction from the management, architectonic, and scaling requirements of core infrastructure.
  - *Hybrid:* this is a way to connect infrastructure and applications between cloud-based resources and existing resources that are not located in the cloud. The most common method of hybrid deployment is between the cloud and existing on-premises infrastructure to extend, and grow, an organization's infrastructure into the cloud while connecting cloud resources to internal system.
  - *On-premises*: using virtualization and resource management tools, this is sometimes called "private cloud". On-premises deployment does not provide many of the benefits of cloud computing but is sometimes sought for its ability to provide dedicated resources. In most cases this deployment model is the same as legacy IT infrastructure while using application management and virtualization technologies to try and increase resource utilization.

6.1 Amazon Web Services

- Since 2006, Amazon Web Services has been the world's most comprehensive and broadly adopted cloud platform.
- AWS offers over 90 fully featured services for compute, storage, networking, database, analytics, application services, deployment, management, developer, mobile, Internet of Things (IoT), Artificial Intelligence (AI), security, hybrid and enterprise applications, from 44 Availability Zones (AZs) across 16 geographic regions in the U.S., Australia, Brazil, Canada, China, Germany, India, Ireland, Japan, Korea, Singapore, and the UK.
- **The AWS Cloud infrastructure is built around Regions and Availability Zones.** AWS Regions provide multiple, physically separated, and isolated Availability Zones that are connected with low latency, high throughput, and highly redundant networking.
- AWS also offers managed compute options, like Amazon Lightsail, that allow you to use compute capacity without worrying about provisioning or managing the underlying hardware. In addition, AWS has other options that go beyond raw server capacity. It offers container services that allow you to use Docker through Elastic Container Service, or ECS, or Kubernetes through EKS. We also offer pure serverless solutions, like AWS Lambda. With the flexibility of AWS compute services, you can run virtually any application in the cloud.
- **AWS Lambda vs Amazon EC2**[2]
  - EC2 requires management and provisioning of the environment. Each EC2 instances runs not just a full copy of an operating system, but a virtual copy of all the hardware that the operating system needs to run. In contrast, what AWS Lambda requires is enough system resources and dependencies to run a specific program.
  - The main difference between AWS Lambda vs EC2 (virtual server-based resources) is the responsibility of provisioning and use cases to name a few. AWS Lambda pricing is one of the biggest factors as well.
  - With the computing resources like AWS Lambda, the computing resources can scale and descends automatically based on real-time demands.
  - The architecture of applications built using functions like AWS Lambda is popularly called serverless architecture. AWS Lambda is a splendid example of how the overhead of the operation team is going to be a distant memory.
  - EC2 is a virtual cloud infrastructure service offered by AWS. This provides on-demand computing resources through which you can create powerful servers in the cloud.

[2] https://www.simform.com/aws-lambda-vs-ec2/

  - The entire hardware of EC2 is fragmented into multiple resources which are offered in the form of instances which are scalable in terms of computing memory and processing power.
  - With Amazon EC2, you have the facility of provisioning virtual machines as per your applications' requirements. Such facility is provided on a utility based subscription model where the user is billed as per their consumption of resources.
  - Lambdas use ECS and these containers are not available to configure manually. On the other hand, Lambdas are exposed through API Gateway which functions as a URL router to your Lambdas.
  - **Setup & Management Environment**
    * **AWS Lambda:** Whether you need to set up a multiple or single environment, you do not need to do much of a work. You are not required to spin up or provision containers or make them available for your applications, scaling is fully automated.
    * **Amazon EC2:** With EC2, setting up includes logging in via SSH and manually installing Apache and doing a git clone. Along with that, you need to install and configure all the required software in a manner which is automated and reproducible.
  - **Performance**
    * **AWS Lambda:** As per the official documentation, AWS Lambda has the timeout of 300 seconds. This limits the type of tasks lambda can deal with. Long-running functions and complex tasks aren't a good fit. Also, Lambda has a package size limit of 50 MB. More to that, "/tmp" file storage has a limit of 512 MB.
    * **Amazon EC2:** This has pretty flexible options. You can definitely work with long running tasks since instances are available for different types of requirements with different configurations. This makes EC2 a better option over Lambda. Managing dependencies in EC2 isn't a big problem since it doesn't have constraints when it comes to temporary storage. Though what you should consider is the size of software packages and corresponding instance CPU. This is because your CPU may undergo burden if it's not configured for the same.

- **The serverless architecture**
  - Is a way to build and run applications and services without having to manage the infrastructure behind it. Your application still runs on servers, of course, but the server management is done by AWS. You no longer have to provision, scale, and maintain services to run your applications, databases, or storage systems. With the serverless architecture, you can execute your code only

when needed, and scale automatically from a few requests per day to thousands of requests per second. And you only pay for the compute time you consume. There is no charge when your code's not running.

  - Serverless is the native architecture of the cloud that enables you to shift more of your operational responsibilities to AWS, increasing your agility and innovation. Serverless allows you to build and run applications and services without thinking about servers. It eliminates infrastructure management tasks such as server or cluster provisioning, patching, operating system maintenance, and capacity provisioning. You can build them for nearly any type of application or back-end service, and everything required to run and scale your application with high availability is handled for you.
  - **AWS Lambda** is serverless compute service from Amazon.
  - So there are 4 main benefits:
    * No server management.
    * Flexible scaling.
    * Pay for value.
    * Automated high availability.

- **AWS CloudFormation** provides a common language for you to describe and provision all the infrastructure resources in your cloud environment via JSON file. CloudFormation allows you to use programming languages or a simple text file to model and provision, in an automated and secure manner, all the resources needed for your applications across all regions and accounts. This gives you a single source of truth for your AWS resources. AWS CloudFormation is available at no additional charge, and you pay only for the AWS resources needed to run your applications.

- **AWS Database Migration Service** migrate databases to AWS quickly and securely. The source database remains fully operational during the migration, minimizing downtime to applications that rely on the database. The AWS Database Migration Service can migrate your data to and from most widely used commercial and open-source databases. There is no need to install any drivers or applications, and it does not require changes to the source database in most cases. You can begin a database migration with just a few clicks in the AWS Management Console. You only pay for the compute resources used during the migration process and any additional log storage. Migrating a terabyte-size database can be done for as little as $3.

- **AWS Cost Explorer** lets you visualize, understand, and manage your AWS costs and usage over time. You can create custom reports (including charts and tabular data) that analyze cost and usage data, both at a high level (e.g., total costs and usage across all accounts) and for highly specific requests.

- **AWS Trusted Advisor** is an online resource to help you reduce costs, increase performance, and improve security by optimizing your AWS environment. Trusted Advisor provides real-time guidance to help you provision your resources by following our best practices.
- **A Region**
  - Region is a geographically self-contained area where all of the resources you need for your application, all the compute, all the storage, are contained. All resources you need for your application are there.
  - A region is a collection of availability zones. It can consist of one or more data centers. And you don't have to worry about the distance between them because AWS connects those availability zones with a proprietary high speed fiber network, multiple lines between every availability zone so you can treat it as a single area. But you run your application simultaneously across all of the availability zones. The idea is it doesn't matter what might happen to an availability zone, because there might be some natural disaster, a hurricane, a tornado, an earthquake that we don't want you to worry about. It doesn't matter if there's a temporary loss of connectivity to an availability zone because your application runs in both of them at the same time. This is how you can not only be effective, be scalable, but also highly available all while running in a single region of your choice.
  - There are many many regions all over the world, and you need to decide which is the best for your business:
    * Latency: where are your customers located?
    * Cost: not every region is priced the same.
    * Compliance: legal restrictions (such as GDPR from Europe).
    * Service availability: sometimes a new feature is released and is not available for some regions (it will run eventually, but that can take even a few months).
- **Amazon Virtual Private Cloud (VPC)**
  - In AWS, VPC is used for isolating a single application from the millions other applications running on AWS.
  - Amazon VPC lets you provision a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network that you define. You have complete control over your virtual networking environment, including the selection of your own IP address range, the creation of subnets, and the configuration of route tables and network gateways. You can use both IPv4 and IPv6 in your VPC for secure and easy access to resources and applications. You could create up to five non-default VPCs per AWS account per Region.

- A VPC spans all the Availability Zones in the Region. After creating a VPC, you can add one or more subnets in each Availability Zone. When you create a subnet, you specify the CIDR block for the subnet, which is a subset of the VPC CIDR block. Each subnet must reside entirely within one Availability Zone, and it can't span Availability Zones.[3]
- The point of VPC is to provide a frame/box that all of your application lives inside, and the idea is nothing comes in the box, nothing gets out of the box, without your specific permission, and whether you're filtering by network protocol, or port, or IP address, or by user or other information, you maintain complete control of all the assets inside your VPC.
- When you create a VPC, you also then divide the space inside the VPC into subnets.
- An important concept that's used in networking on AWS is CIDR, or Classless Inter-Domain Routing. CIDR network addresses are allocated in a virtual private cloud (VPC) and in a subnet by using CIDR notation.
- But VPC stops all traffic in and all traffic out, and if we're going to put a web server in there, well, that means nobody can talk to it. We have to add a IGW (Internet Gateway) and attach it to the VPC and then she'll create a route table and associate that with the subnet, so that any communication that wants to talk to assets in this subnet, can come in and out of that IGW.
- Then, once we defined VPC, subnet, IGW (associated with a given subnet), then we can launch EC2 instance in that subnet.
- Also, we can add a database to it. But a database shouldn't go in the same public access subnet where my web servers are, because I never want anyone from the outside, at least, in my business case, to access a database directly. So, we're going to make another subnet inside my VPC (private one). But we will not associate this new subnet to IGW. We want that only our web server can communicate with the database - and it can do it, because they are in the same VPC.
- If you want to achieve high-availability, you should create another subnet (with different Availability Zone), so that you have two pairs: two private and two public subnets. All 4 are in the same VPC (we don't have to change this, because VPC is already multi-availability Zone structure). And we have to associate our route table (Public Route Table) to both public subnets.
- Now you need ELB - Elastic Load Balancer, that will balance the input load. So you have to associate ELB with two EC2 instances, so that it doesn't matter which one gets the traffic.
- There's one more type of gateway we can add into your VPC and that's called a virtual private gateway. And a virtual private gateway, or a VGW, can

[3] https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Subnets.html

be created and attached, and this can even be associated with your private subnets. So that if you've got a DBA, that is connecting over your own on-premises data center, she can connect through the VGW over a VPN connection and never go through the IGW. And that's it.

- **AWS Lambda**
  - It lets you run code without provisioning or managing servers. You pay only for the compute time you consume - there is no charge when your code isn't running.
  - With Lambda, you can run code for virtually any type of application or backend service - all with zero administration. Just upload your code and Lambda takes care of everything required to run and scale your code with high availability.[4]
  - Lambda natively supports 6 programming languages: Node.js, Python, Java, C#, Ruby, and Go.

AWS Container Services

- **Amazon Elastic Container Service (Amazon ECS)**
  - ECS[5] is a highly scalable, high-performance container orchestration service that supports Docker containers.
  - It allows you to run and scale containerized applications on AWS. With simple API calls, you can launch and stop Docker-enabled applications, query the complete state of your application, and more.
- **Amazon Elastic Container Service for Kubernetes (Amazon EKS)**
  - EKS[6] makes it straightforward to deploy, manage, and scale containerized applications that use Kubernetes on AWS. Amazon EKS runs the Kubernetes management infrastructure for you across multiple AWS availability zones to eliminate a single point of failure.
  - Amazon EKS is certified Kubernetes conformant so you can use existing tooling and plugins from partners and the Kubernetes community.
  - Applications running on any standard Kubernetes environment are fully compatible and can be easily migrated to Amazon EKS.
- **AWS Fargate**
  - Fargate[7] is a compute engine for Amazon ECS and Amazon EKS that allows you to run containers without having to manage servers or clusters.

[4] https://aws.amazon.com/lambda/
[5] https://aws.amazon.com/ecs/
[6] https://aws.amazon.com/eks/
[7] https://aws.amazon.com/fargate/

  - You just define your application as you do for Amazon ECS. You package your application into task definitions, specify the CPU and memory needed, define the networking and IAM policies that each container needs, and upload everything to Amazon ECS. After everything is setup, AWS Fargate launches and manages your containers for you.

Computing Services

- **Amazon Elastic Compute Cloud (Amazon EC2)**
  - EC2 is a web service that provides secure and resizable compute capacity in the cloud. It's designed to make web-scale cloud computing easier for developers.
  - It is a compute service that allows you to provision virtual servers on demand. Each virtual server you provision is called an EC2 instance. Just about anything you can do with a server in a traditional sense, you can do with an EC2 instance.
  - AWS supports a range of operating systems including Linux, Ubuntu, Windows, and more. To select the operating system, you choose an Amazon Machine Image, or what we call an AMI. An AMI contains information about how you want your instance to be configured, including the operating system and possible applications to be installed on that instance.
  - You can launch one or many instances from a single AMI, which would create multiple instances that all share the same configuration.
  - Beyond the operating system, you can also configure the instance type and size, which correspond to the amount of compute, memory, and networking capabilities available per instance. This allows you to control the underlying hardware and the capacity of that hardware with just a few clicks or lines of code.
  - If you choose an EC2 instance type and then later realized a different type would have been better suited for the application, you can easily change the underlying hardware. If you decide that you want to resize your EC2 instance, that isn't a problem in the cloud either. EC2 is a resizable resource with a few clicks in the console, or it can be done programmatically through an API call. This enables you to embrace change over time.
  - Amazon EC2 provides a wide selection of **instance types** that are optimized to fit different use cases. Instance types comprise varying combinations of CPU, memory, storage, and networking capacity. They give you the flexibility to choose the appropriate mix of resources for your applications. Each instance type includes one or more instance sizes, which allows you to scale your resources to the requirements of your target workload.
- **Amazon Lightsail**

  - This is the easiest way to get started with AWS for developers, small businesses, students, and other users who need a simple virtual private server (VPS) solution.
  - Lightsail provides developers compute, storage, and networking capacity, and it also provides capabilities to deploy and manage websites and web applications in the cloud. Lightsail includes everything you need to launch your project quickly--a virtual machine, solid state drive (SSD)-based storage, data transfer, Domain Name System (DNS) management, and a static IP--for a low, predictable monthly price.
  - If you want to just simplify the whole process of EC2. You don't want to go through the process of spinning up an EC2 instance. You just want a solution to running your application.
  - Lightsail has a number of pre-built options, you simply select, launch, and you're done. For example, you might just want a WordPress site. Lightsail has one already built for you.

Storage Services

There are 2 different approaches: object storage, and block storage.

- **Amazon Elastic Block Store (Amazon EBS)**
  - The most common block storage. It provides persistent block storage volumes for use with Amazon EC2 instances in the AWS Cloud. Each Amazon EBS volume is automatically replicated inside an Availability Zone to protect you from component failure, which offers high availability and durability. Amazon EBS volumes offer the consistent and low-latency performance that you need to run your workloads.
  - When you launch your EC2 instance, you're going to need some kind of block storage to go with it. It's part of the boot volume or maybe it's a separate data volume. And AWS has racks of unused storage that you can provision to sizes as large as you need up to many terabytes in size.
  - EBS can be attached to only 1 instance of EC2.
  - When you launch the EC2 instance, the boot volume can attach directly to your EC2 instance, as well as the data volume. These volumes live independent of the EC2 instance themselves. In fact, they may already exist before your EC2 instance launches. When it launches, it simply finds the volume and attaches it the same way you might have an old drive from a laptop.
  - Amazon EBS provides a range of options that allow you to optimize storage performance and cost for your workload. These options are divided into two major categories: SSD-backed storage for transactional workloads, such as databases and boot volumes (performance depends primarily on IOPS), and hard disk drive (HDD)-backed storage for throughput-intensive workloads,

such as MapReduce and log processing (performance depends primarily on MB/s).

- The Elastic Volume feature of Amazon EBS allows you to dynamically increase capacity, tune performance, and change the type of live volumes with no downtime or performance impact. This allows you to easily right-size your deployment and adapt to performance changes.

- **Amazon Simple Storage Service (Amazon S3)**
  - This stores data as objects within resources that are called buckets. You can store as many objects as you want within a bucket, and you can write, read, and delete objects in your bucket. Objects can be up to 5 TB in size.
  - You can control access to both the bucket and the objects (who can create, delete, and retrieve objects in the bucket for example), and view access logs for the bucket and its objects. You can also choose the AWS Region where a bucket is stored to optimize for latency, minimize costs, or address regulatory requirements.
  - With Amazon S3, you pay only for what you use. There is no minimum fee. There is even a calculator[8] for estimating your monthly bill.

- **Amazon Elastic File System (Amazon EFS)**
  - It provides simple, scalable, elastic file storage for use with AWS Cloud services and on-premises resources. It is straightforward to use, and it offers a simple interface that allows you to create and configure file systems quickly and easily.
  - Amazon EFS is designed to provide massively parallel shared access to thousands of Amazon EC2 instances. When an Amazon EFS file system is mounted on Amazon EC2 instances, it provides a standard file system interface and file system access semantics, which allows you to seamlessly integrate Amazon EFS with your existing applications and tools. Multiple Amazon EC2 instances can access an Amazon EFS file system at the same time, thus allowing Amazon EFS to provide a common data source for workloads and applications that run on more than one Amazon EC2 instance.
  - This is designed to be a regionally distributed, meaning it doesn't live inside any one subnet, a regionally distributed file store that can automatically attach to multiple EC2 instances simultaneously, many EC2 instances, including the instances in different VPCs.
  - This way, if you need a corporate directory, a corporate file store where everyone connects to the same document store, EFS can be that solution for you. In fact, using EFS File Sync, you can even have that directory connect to your on-premises data center, allowing your people at your home network to

[8] https://calculator.s3.amazonaws.com/index.html

think they're connecting to home directories when really they're being stored inside EFS on AWS, with all the security and regional distribution that comes automatically.

Database Services

- **Amazon Relational Database Service (Amazon RDS)**
  - It makes it straightforward to set up, operate, and scale a relational database in the cloud. It provides cost-efficient and resizable capacity while automating time-consuming administration tasks such as provisioning hardware, setting up the database, patching, and making backups.
  - Amazon RDS currently supports 6 database engines:
    - Amazon Aurora
    - PostgreSQL
    - MySQL
    - MariaDB
    - Oracle
    - Microsoft SQL Server
- **Amazon DynamoDB**
  - DynamoDB is a fully managed, fast, and scalable NoSQL database solution that delivers reliable performance at any scale.
  - You do not have to manage any of the underlying infrastructure running that database. When you need to start using DynamoDB, you simply create a table, define your throughput needs, and you can start populating it with your data.
  - Its a fully managed cloud database, and it supports both document and key-value store models.
  - To compare this with Amazon RDS, with RDS when you need to use it, you define how much capacity you need in terms of memory and CPU. So you're defining the underlying hardware that we're running your database on. With Dynamo, you simply just tell us how much you need talk to that table by provisioning your throughput needs. With Amazon DynamoDB, you can start small, specify the throughput you need, and easily scale your capacity requirements in seconds, as needed.
  - It automatically partitions data over multiple servers to meet your requested capacity. As your data grows, AWS handles the management of scaling your database. There is no limit on table size, which means you can store any amount of data. DynamoDB synchronously replicates your data across three facilities in an AWS Region to ensure redundancy and availability.

  - A **table** is a **collection of items**, and each item is a **collection of attributes**.
  - The first is the **partition key**, which is a simple primary key, composed of one attribute. For example, if we are storing music data, our primary key should be "Artist". The next to the partition key is a **sort key**. That should be a song title. When a partition and a sort key exist in a table, it is referred to as a "**composite primary key**" and is composed of two attributes. And then we have a set of attributes.
  - A **"secondary index"** is a data structure that contains a subset of attributes from a table, along with an alternate key to support query operations. A table can have multiple secondary indexes, which give you application access to many different query patterns.
    * A **"local secondary index"** is an index that has the same partition key as the base table, but a different sort key.
    * A **"global secondary index"** is an index that has a partition key and a sort key that can be different from those based on the base table.

Monitoring AWS - Amazon CloudWatch

- With Amazon CloudWatch, you can monitor your cloud infrastructure intelligently. CloudWatch will collect data from your cloud-based infrastructure in one centralized location. With this data, you can create statistics, which drive operational procedures using features such as **CloudWatch Alarms**.
- CloudWatch also allows you to visualize the statistics about your environment through dashboards. You can use out-of-the-box dashboards to view built-in or custom metrics, and you can build your own custom dashboards or partner with a wide range of consultants through the Amazon Partner Network.
- **Amazon CloudWatch Events** delivers a near real-time stream of system events that describe changes in AWS resources. Using simple rules that you can quickly set up, you can match events and route them to one or more target functions or streams. CloudWatch Events becomes aware of operational changes as they occur.
- You can use **Amazon CloudWatch Logs** to monitor, store, and access your log files from Amazon EC2 instances, AWS CloudTrail, Amazon Route 53, and other sources. You can then retrieve the associated log data from CloudWatch Logs.

Elastic Load Balancing (ELB)

- It automatically distributes incoming application traffic across multiple targets, such as Amazon EC2 instances, containers, and IP addresses. It can handle the varying load of your application traffic in a single Availability Zone or across multiple Availability Zones.

- ELB offers 3 types of load balancers that all feature the high availability, automatic scaling, and robust security that are necessary to make your applications fault-tolerant:
  - **An Application Load Balancer** operates at the request level (Layer 7), routing traffic to targets - such as EC2 instances, microservices and containers - within Amazon VPC, based on the content of the request. It's ideal for the advanced load balancing of Hypertext Transfer Protocol (HTTP) and Secure HTTP (HTTPS) traffic.
  - **A Network Load Balancer** operates at the connection level (Layer 4), routing connections to targets - such as Amazon EC2 instances, microservices, and containers - within Amazon VPC, based on IP protocol data. It's ideal for load-balancing Transmission Control Protocol (TCP) traffic.
  - **The Classic Load Balancer** provides basic load balancing across multiple Amazon EC2 instances, and it operates at both the request level and the connection level.
- At some point, our N instances aren't going to be able to handle that demand, and we're going to need more EC2 instances. Instead of launching these instances manually, we want to do it automatically. So, we're going to use what's called **Auto Scaling**. Auto Scaling is what allows us to provision more capacity on demand, depending on different thresholds that we set, and we can set those in CloudWatch.
- **Amazon EC2 Auto Scaling** helps you maintain application availability, and it allows you to dynamically scale your Amazon EC2 capacity up or down automatically according to conditions that you define.
- You can also use Amazon EC2 Auto Scaling to dynamically scale Amazon EC2 instances. Dynamic scaling automatically increases the number of Amazon EC2 instances during demand spikes to maintain performance and decrease capacity during lulls, which can help reduce costs. Amazon EC2 Auto Scaling is well-suited to applications that have stable demand patterns, or applications that experience hourly, daily, or weekly variability in usage.

Security

- **Amazon Shared Responsibility Model** - security and compliance are shared responsibilities between AWS and the customer. This shared model can help relieve a customer's operational burden because Amazon operate, manage, and control the components from the host operating system and virtualization layer down to the physical security of the facilities where the service operates. The customer is responsible for (and manages) the guest operating system (including updates and security patches) and other associated application software, in addition to the configuration of the AWS-provided security group firewall.

- **AWS responsibility**: Security of the Cloud: Amazon is responsible for protecting the infrastructure that runs all of the services that are offered in the AWS Cloud. This infrastructure is composed of the hardware, software, networking, and facilities that run AWS Cloud services.
- **Customer responsibility**: Security in the Cloud: Customer responsibility will be determined by the AWS Cloud services that a customer selects. This determines the amount of configuration work the customer must perform as part of their security responsibilities.
- **AWS Identity Access Management (IAM)**
  - An IAM role is an IAM entity that defines a set of permissions for making AWS service requests. IAM roles are not associated with a specific user or group. Instead, trusted entities assume roles such as an IAM user, an application, or an AWS service like EC2.
  - So IAM is a web service that helps you securely control access to AWS resources. Amazon typically uses credentials from IAM Users or IAM Roles to authenticate with AWS when making API calls. They control the permissions for which API actions those Users or Roles can perform with IAM Policies.

Machine Learning Services

- **Amazon Lex**
  - It provides the advanced deep learning functionalities of automatic speech recognition and NLP that powers Amazon Alexa.
  - The only language that is supported in this moment is English.
  - Amazon Lex uses Amazon Polly that uses text-to-speech, so it is possible to build the complete chat bot with Amazon Lex.
- **Amazon Polly**
  - This service provides text to speech functionality.

API

- **Amazon API Gateway**
  - It is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. With a few clicks in the AWS Management Console, you can create an API that acts as a "front door" for applications to access data, business logic, or functionality from your back-end services, such as workloads running on Amazon Elastic Compute Cloud (Amazon EC2), code running on AWS Lambda, or any web application.

  - Amazon API Gateway supports mock integrations for API methods. This feature enables API developers to generate API responses from API Gateway directly, without the need for an integration backend. As an API developer, you can use this feature to unblock dependent teams that need to work with an API before the project development is complete.

- **Amazon CloudFront**
  - It is a global content delivery network (CDN) service that securely delivers data, videos, applications, and APIs to your viewers with low latency and high transfer speeds.
  - CloudFront is integrated with AWS – including physical locations that are directly connected to the AWS global infrastructure, as well as software that works seamlessly with services including AWS Shield for DDoS mitigation, Amazon S3, Elastic Load Balancing or Amazon EC2 as origins for your applications.

7 Interviews

- (Technical interviews) Note down, and replicate the question! After all is written down and properly understood, you can start with solution. So that me and interviewer are not disconnected from each other. Clarify the question! Don't waste time if/when you don't fully understand the question! Write it down and also maybe along with some example! Define inputs and outputs of some algorithm, if suitable.
- Do not talk only about the job that already *exists*, but about the job that you hope the organization will *create* for you. For this, it is necessary to know:
  - what do you like about the organization,
  - what needs and challenges (don't use the word "problems") can be seen in a given field,
  - what skills do you have to help them (with concrete examples from the past), and what is unique about you and these skills, and
  - what will them cost them not to hire you in the long run.
- Observe *50-50* rule. Mix speaking and listening fifty-fifty in the interviews. Answer something between 20 seconds to 2 minutes.
- What is success? (Bessie Anderson Stanley (1879–1952))
  - To laugh often and much;
  - To win the respect of intelligent people and the affection of children;
  - To earn the appreciation of honest critics and endure the betrayal of false friends;
  - To appreciate beauty;
  - To find the best in others;
  - To leave the world a bit better, whether by a healthy child, a garden patch or a redeemed social condition;
  - To know even one life has breathed easier because you have lived;
  - This is to have succeeded.
- **CV and Linkedin**
  - **Show what you did, how you did it, and what the results were.** Ideally, you should try to make the results "measurable" somehow.

  - CV shouldn't be written in the first person.
  - **CV and Linkedin should highlight your accomplishments, not job duties or descriptions.** Don't be task-based, be achievement-based. Write your resume to emphasize what you did well, not what your duties entailed.
  - For US positions, do not include age, marital status, or nationality. This sort of **personal information is not appreciated** by companies, as it creates a legal liability for them.
  - **Being too language focused:** When recruiters at some of the top tech companies see resumes that list every flavor of Java on their resume, they make negative assumptions about the caliber of candidate. There is a belief in many circles that the best software engineers don't define themselves around a particular language. Thus, when they see a candidate seems to flaunt which specific versions of a language they know, recruiters will often bucket the candidate as "not our kind of person.

- **Cover Letter**
  - Brief cover that summarizes the whole long resume. It is a report, and you have to make it personal and specific to a concrete job. Research the companies!

7.1 Questions for Employer

- **Technical questions**
  - Code reviews? Coding standards? Agile team?
  - Who is in team? (seniors, architects, ...)
  - Ratio of testers to developers to PMs? What is their interaction like?
  - Architecture?
  - My duties? What should I accomplish? Report to whom?
  - How would I be evaluated, how often, and by whom?
  - If you don't mind my asking, I'm curious as to why you yourself decided to work at this organization? What don't you like about this company (what is this company's greatest flaw)?
  - What would you expect from me in the first 90 days?
  - Room for initiative and freedom, vs strict specifications from someone.
  - Operating system, etc.?
- **Non-technical questions**
  - **About position**
    - Why has this position become available? (Someone left / was promoted, or they are expanding a team, or creating a new one. Is there a possibility to become teamleader in the future?)
    - How would you define success in this or related position?
    - How do you see this position evolving in the next 3 years?
  - **Company**
    - What business problem are they trying to solve? (Everybody likes a candidate who shows genuine interest, motivation, and curiosity for a problem that is close to their hearts.)
    - What are the company's highest goals for the next year?
    - What significant changes has this company gone through in the past 5 years?
- **Benefits question**
  - Type of employment, flexibility, growth opportunities, perks (table tennis, gym, ...).
  - Benefits, stock options, annual bonuses?
  - The ability to work from home?
  - Possibility to be transferred to another country or team?

7.2 General Things to Know

- **STAR method - Situation, Target, Action, Result** - you can use this for solving almost any problem using this "template".
- Throughout the interview, keep in mind: employers don't really care about your past; they only ask about it, in order to try to predict your future (behavior) with them, if they decide to hire you.
- Do not bad-mouth your previous employer(s) during the interview, even if they were terrible people. Therefore, during the hiring-interview, before you answer any question the employer asks you about your past, you should pause to think, "What fear about the future caused them to ask this question about my past?"
- Basically approach them not as a "job-beggar" but humbly as a resource person, able to produce better work for that organization than any of the people who worked in that position, previously.
- Salary negotiation should only happen when they have definitely said they want you; prior to that, it's pointless. Before accepting a job offer, always ask about salary. However, it has been proven that a person who mention salary (as a number) first, generally loses. Research the range that the employer likely has in mind, and then define an interrelated range for yourself, relative to the employer's range.
- Arrogance is a red flag, but you still want to make yourself sound impressive. So how do you make yourself sound good without being arrogant? By being specific!
- After an interview, when it went good, you may ask: "When may I expect to hear from you?"
- **Flower exercise** - it is a self-inventory technique with 7 petals, because there are 7 ways of thinking about yourself, or 7 ways of describing who you are:
  - **people** - the kinds of people you most prefer to work with
  - **workplace** - your favorite workplace, or working conditions—indoors/outdoors, small company/large company, windows/no windows, etc
  - **skills** - what you can do, and what your favorite functional/transferable skills are
  - **purpose** - your goals or sense of mission and purpose for your life. Alternatively, or in addition, you can get even more particular and describe the goals or mission you want the organization to have, where you decide to work
  - **knowledge** - what you already know—and what your favorite knowledge or interests are among all that stuff stored away in your head

- **salary** - your preferred salary and level of responsibility— working by yourself, or as a member of a team, or supervising others, or running the show— that you feel most fitted for, by experience, temperament, and appetite
- **geography** - your preferred surroundings—here or abroad, warm/cold, north/-south, east/west, mountains/coast, urban/suburban/rural/rustic - where you'd be happiest, do your best work, and would most love to live, all year long, or part of the year, or vacation time, or sabbatical - either now, five years from now, or at retirement

What the Flower Diagram does is describe who you are in all 7 ways, summarized on one page, in one graphic. After all, you are not just one of these things; you are all of these things. The Flower Diagram is a complete picture of you.

7.3 Behavioral Questions

- **Getting to Know You**
  - What motivates you at work?
  - Describe what your preferred supervisor - employee relationship looks like.
  - What two or three things are most important to you in your work?
- **Knowledge & Interests**
  - What do you think are the most pressing issues in this field?
  - What challenges does this position present for you?
  - What do you think it takes to be successful in this organization?
  - What do you know about our company?
- **Readiness & Experience**
  - What is your greatest strength/weakness?
  - Tell me about a problem you have encountered and how you dealt with it?
  - Tell me about a mistake you made and what you learned from it.
  - What experience do you have in this field?
  - How have you prepared yourself to switch fields?
- **Goals, Motivation & Values**
  - Why do you think you will like this field?
  - Describe a time when you saw some problem and took the initiative to correct it rather than waiting for someone else to do it.
  - Give me an example of a time you were able to be creative with your work. What was exciting or difficult about it?
  - Tell me about a time you were dissatisfied in your work. What could have been done to make it better?
- **Teamwork**
  - Describe a time when you worked closely with someone who had a very different personality than you.
  - Tell me about a time you faced a conflict while working on a team. How did you handle the conflict?
  - Describe a time when you struggled to build a relationship with someone important.
  - Tell me about a time you needed to get information from someone who wasn't very responsive. What did you do?

- **Ability to Adapt**
  - Tell me about a time you were under a lot of pressure. What was the situation and how did you get through it?
  - Describe a time when your team or company was undergoing change. How did it impact you, and how did you adapt?
  - Tell me about your very first job. What did you do to learn the ropes?
  - Tell me about a time you failed. How did you deal with this situation?
- **Time Management Skills**
  - Tell me about a long-term project that you managed. How did you keep organized and make sure everything was moving along as planned?
  - Tell me about a time you set a goal for yourself. How did you ensure that you would meet your objective?
  - Give me an example of a time you managed multiple responsibilities. How did you handle it?
- **Communication Skills**
  - Tell me about a time you successfully persuaded someone to understand your perspective at work.
  - Describe a time when you were the primary "expert". How did you ensure that everyone understood you?
  - Describe a time when you could only use written communication to get your ideas across to your team.

7.4 Software Engineering Interview Preparation

- These steps are good to perform during technical question:
  1. **Listen carefully.** Listen to every detail.
  2. **Draw an example.** Never try to solve it in your head and never use minimalist example. Write specific and sufficiently large example, not some special case.
  3. **State a brute force.** Write sub-optimal PoC that works.
  4. **Optimize.** What you can - unused information, space / time trade-offs, using hash table, ...
  5. **Walk through.** Go through your algorithm to see if it actually still works. Write pseudo-code if you like.
  6. **Implement.** Yes, implementation is here, in such later "phase". Start with top left corner, write beautiful code with nicely named variables, good spacing, and so on. This part is very important.
  7. **Test.** See if the implementation works.

OO questions

These are mostly about demonstrating that you understand how to create elegant, maintainable object-oriented code. Poor performance on this type of question may raise serious red flags. These questions are intentionally vague in order to test whether you'll make assumptions or if you'll ask clarifying questions. You may even want to go through the "six Ws": **who, what, where, when, how,** and **why**. There are multiple steps that needs to be taken:

1. Handle ambiguity
2. Define the core objects
3. Analyze relationships
4. Investigate actions

Data Structures

- *Hash Tables (also known as Associative Arrays)*
- They are data structures, (Python *dict* implements it) that are associative arrays, that maps keys to values in effective way (for highly effective lookup).
  - Very common implementation is with an array of linked lists (because of possible collisions) and a hash code function.

  - A hash table uses a hash function to compute an index into an array of slots, from which the correct value can be found.
  - Two different keys could have the same hash code, as there may be an infinite number of keys and a finite number of ints. Two different hash codes could, of course, map to the same index.
  - To retrieve the value pair by its key, you repeat this process. Compute the hash code from the key, and then compute the index from the hash code. Then, search through the linked list for the value with this key. If the number of collisions is very high, the worst case runtime is $O(N)$, where N is the number of keys. However, we generally assume a good implementation that keeps collisions to a minimum, in which case the lookup time is $O(1)$.
  - Alternatively, we can implement the hash table with a balanced binary search tree. This gives us an $O(log\, N)$ lookup time. The advantage of this is potentially using less space, since we no longer allocate a large array. We can also iterate through the keys in order, which can be useful sometimes.

- *Arrays = Lists*
  - They are automatically resizable. The array or list will grow as you append items. In some languages, like Java, arrays are fixed length. The size is defined when you create the array.
  - An array that resizes itself as needed while still providing $O(1)$ access. A typical implementation is that when the array is full, the array doubles in size. Each doubling takes $O(n)$ time, but happens so rarely that its amortized insertion time is still $O(1)$.

- *Linked Lists*
  - A linked list is a data structure that represents a sequence of nodes. In a singly linked list, each node points to the next node in the linked list. A doubly linked list gives each node pointers to both the next node and the previous node.
  - Unlike an array, a linked list does not provide constant time access to a particular "index" within the list. This means that if you'd like to find the Kth element in the list, you will need to iterate through K elements. The benefit of a linked list is that you can add and remove items from the beginning of the list in constant time. For specific applications, this can be useful.
  - A linked list saves memory. It only allocates the memory required for values to be stored. In arrays, you have to set an array size before filling it with values, which can potentially waste memory.
  - Linked list nodes can live anywhere in the memory. Whereas an array requires a sequence of memory to be initiated, as long as the references are updated, each linked list node can be flexibly moved to a different address.

- *Stacks & Queues*
  - A stack uses LIFO (last-in first-out) ordering. There are *push(item), pop(), peek(), isEmpty()* operations. Unlike an array, a stack does not offer constant-time access to the ith item. However, it does allow constant time adds and removes, as it doesn't require shifting elements around.
  - A queue implements FIFO (first-in first-out) ordering. There are *add(), remove(item), peek(), isEmpty()* operations.
  - One place where queues are often used is in breadth-first search or in implementing a cache.
- *Heap* (as a datastructure, not heap in memory)[1]
  - Common implementation of a priority queue. A priority queue contains items with some priority. You can always take an item out in the priority order from a priority queue. You can also use stack or queue for implementation of a priority queue, but it works differently. This is because the priority of an inserted item in stack increases, and the priority of an inserted item in a queue decreases.
  - A heap is one of the tree structures and represented as a **binary tree** (see below).
  - If you implement this structure with an array, you have to manipulate with nodes/items by indices:
    - A root node | $i = 1$, the first item of the array
    - A parent node | $parent(i) = i/2$
    - A left child node | $left(i) = 2i$
    - A right child node | $right(i) = 2i + 1$
  - You need 2 operations to build a heap from an arbitrary array:
    - *min_heapify*: make some node and its descendant nodes meet the heap property. It basically iterates though all non-leaf nodes, from reversed order (*for =n/2 downto 1*), and it calls procedure *build_min_heap*.
    - *build_min_heap*: produce a heap from an arbitrary array.
  - It is possible to implement *heapsort* with *heap:*
    1. Build a heap from an arbitrary array with *build_min_heap*.
    2. Swap the first item with the last item in the array.
    3. Remove the last item from the array.
    4. Run *min_heapify* to the first item.
    5. Back to step 2.

[1] https://towardsdatascience.com/data-structure-heap-23d4c78a6962

- *Trees, Tries, and Graphs*
  - **A tree is a data structure composed of nodes.** Each tree has a root node. (Actually, this isn't strictly necessary in graph theory, but it's usually how we use trees in programming). The root node has 0+ child nodes. Each child node has 0+ child nodes, and so on. The tree cannot contain cycles. The nodes may or may not be in a particular order, they could have any data type as values, and they may or may not have links back to their parent nodes.
  - **A binary search tree** - every node fits a specific ordering property: *all left descendants* <= n < *all right descendants*. This must be true for each node n. Note that this inequality must be true for all of a node's descendants, not just its immediate children.
  - **Balanced trees** - balancing a tree does not mean the left and right sub-trees are exactly the same size. One way to think about it is that a "balanced" tree really means something more like "not terribly imbalanced". It's balanced enough to ensure $O(log\, n)$ times for insert and find, but it's not necessarily as balanced as it could be. Two common types of balanced trees are *Red-black trees* and *AVL trees*.
  - **Complete binary tree** - it is a binary tree in which every level of the tree is fully filled, except for perhaps the last level. To the extent that the last level is filled, it is filled left to right. So if, for example, there is binary tree that has depth = 3, but with just 3 nodes instead of 4, it is complete binary tree if there are leaf nodes from left to right except the the most right one. If leaf node in the middle is missing, or the very left one, it is not complete binary tree.
  - **Full binary trees** - every node has either zero or two children. That is, no nodes have only one child.
  - **Perfect binary tree** - one that is both full and complete. All leaf nodes will be at the same level, and this level has the maximum number of nodes. Note that **perfect trees are rare in interviews and in real life**, as a perfect tree must have exactly $2^k - 1$ nodes (where k is the number of levels). In an interview, do not assume a binary tree is perfect.
  - **Binary tree traversal**
    - In-Order Traversal means to "visit" (often, print) the left branch, then the current node, and finally, the right branch.
    - Pre-Order Traversal visits the current node before its child nodes. Root is always the first node visited.
    - Post-Order Traversal visits the current node after its child nodes. Root is always the last node visited.
  - **Binary heaps**

  - See heap data structure above, they can be also used for implementation of priority queue.
  - Max-heaps are essentially equivalent to min-heaps, but the elements are in descending order rather than ascending order.
  - A min-heap is a complete binary tree (that is, totally filled other than the rightmost elements on the last level) where each node is smaller than its children. So, leave nodes don't have to be placed in total order! Actually, absolutely no order of nodes in this tree is guaranteed, only that a given node must be smaller than each of its children! The root, therefore, is the minimum element in the tree. There are 2 operations: *insert* and *extract_min*. We insert at the rightmost spot so as to maintain the complete tree property. Then, we "fix" the tree by swapping the new element with its parent, until we find an appropriate spot for the element. We essentially bubble up the minimum element. This takes $O(log\, n)$ time, where n is the number of nodes in the heap. Extraction of min: first, we remove the minimum element and swap it with the last element in the heap (the bottom-most, right-most element). Then, we bubble down this element, swapping it with one of its children (always the smaller one) until the min-heap property is restored. This operation also takes $O(log\, n)$ time.

- **Tries (Prefix trees)**
  - A trie is a variant of an n-ary tree in which characters are stored at each node. Each path down the tree may represent a word. The ***** nodes (sometimes called "null nodes") are often used to indicate complete words. Root is special node that has no character value.
  - A node in a trie could have anywhere from 1 through ALPHABET_SIZE + 1 children.
  - Very commonly, a trie is used to store the entire (English) language for quick prefix lookups. While a hash table can quickly look up whether a string is a valid word, it cannot tell us if a string is a prefix of any valid words. A trie can do this very quickly. A trie can check if a string is a valid prefix in $O(K)$ time, where K is the length of the string.
- **Graphs**
  - A tree is actually a type of graph, but not all graphs are trees. Simply put, a **tree is a connected graph without cycles.** A graph is simply a collection of nodes with edges between (some of) them.
  - Graphs can be either **directed** (like the following graph) or **undirected**.
  - The graph might consist of multiple isolated sub-graphs. If there is a path between every pair of vertices, it is called a "connected graph".
  - The graph can also have cycles. An "acyclic graph" is one without cycles.

* They can be implemented using adjacency list (for example, a class *Graph* with one member - a *list of nodes*, and class *Node*, that contain members *value* and *list of children*), or adjacency matrices (NxN boolean matrix, where N is the number of nodes, and $matrix[i][j] = true$ indicates the edge from node i to j). In the adjacency list representation, you can easily iterate through the neighbors of a node. In the adjacency matrix representation, you will need to iterate through all the nodes to identify a node's neighbors.
* The two most common ways to search a graph are **depth-first search** and **breadth-first search**. In depth-first search (DFS), we start at the root (or another arbitrarily selected node) and explore each branch completely before moving on to the next branch. That is, we go deep first (hence the name depth-first search) before we go wide. In breadth-first search (BFS) we start at the root (or another arbitrarily selected node) and explore each neighbor before going on to any of their children. That is, we go wide (hence breadth-first search) before we go deep. DFS is often preferred if we want to visit every node in the graph. If we want to find the shortest path (or just any path) between two nodes, BFS is generally better.
* DFS can be recursive algorithm, and BFS iterative one. If you are asked to implement BFS, the key thing to remember is the use of the queue.
* **Bidirectional search** is used to find the shortest path between a source and destination node. It operates by essentially running two simultaneous breadth-first searches, one from each node. When their searches collide, we have found a path.

Algorithms

- *Breadth-First Search*
- *Depth-First Search*
- *Binary Search*
  - Here, we look for an element x in a sorted array by first comparing x to the midpoint of the array.
  - If x is less than the midpoint, then we search the left half of the array. If x is greater than the midpoint, then we search the right half of the array.
  - We then repeat this process, treating the left and right halves as sub-arrays. Again, we compare x to the midpoint of this sub-array and then search either its left or right side. We repeat this process until we either find x or the sub-array has size O.
- *Bubble Sort*

  - Runtime is $O(n^2)$. Space complexity is $O(1)$.
  - We start at the beginning of the array and swap the first two elements if the first is greater than the second. Then, we go to the next pair, and so on, continuously making sweeps of the array until it is sorted. In doing so, the smaller items slowly"bubble" up to the beginning of the list.

- *Selection Sort*
  - Runtime is $O(n^2)$. Space complexity is $O(1)$.
  - Find the smallest element using a linear scan and move it to the front (swapping it with the front element). Then, find the second smallest and move it, again doing a linear scan. Continue doing this until all the elements are in place.

- *Merge Sort*
  - Runtime is $O(nlog\,n)$. Space complexity is $O(n)$.
  - Merge sort divides the array in half, sorts each of those halves, and then merges them back together. Each of those halves has the same sorting algorithm applied to it. Eventually, you are merging just two single element arrays. It is the "merge" part that does all the heavy lifting.

- *Quick Sort*
  - Runtime is $O(nlog\,n)$ in average, but the worst case is $O(n^2)$. Space complexity is $O(log\,n)$.
  - In quick sort, we pick a median (usually just a random element) and partition the array, such that all numbers that are less than the partitioning element come before all elements that are greater than it. The partitioning can be performed efficiently through a series of swaps.
  - If we repeatedly partition the array (and its sub-arrays) around an element, the array will eventually become sorted. However, as the partitioned element is not guaranteed to be the median (or anywhere near the median), our sorting could be very slow.

- *Radix Sort*
  - Runtime is $O(kn)$, where k is the number of passes of the sorting algorithm, and n is the number of elements.
  - It is a sorting algorithm for integers (and some other data types) that takes advantage of the fact that integers have a finite number of bits. In Radix sort, we iterate through each digit of the number, grouping numbers by each digit.
  - For example, if we have an array of integers, we might first sort by the first digit, so that the Os are grouped together. Then, we sort each of these

groupings by the next digit. We repeat this process sorting by each subsequent digit. until finally the whole array is sorted.

- *The Sieve of Eratosthenes*
  - For generating list of prime numbers by generating all values from 1 to a given max number, and cross-out prime numbers sequentially (first prime number is 2, then 4, then 6, and so on; the next prime number is always incremented from the last one, so then it would be 3, and then 6, 9, and so on). Eventually there will be only those left, which are actual prime numbers.
  - This can be implemented with "flags" list, that contain $True/False$ values for all numbers. Crossing out is labeling from $True$ (from initialization) to $False$.
- (Optional) Very advanced:
  - *MapReduce* is used widely in system design to process large amounts of data. As its name suggests, a MapReduce program requires you to write a Map step and a Reduce step. The rest is handled by the system.
  - *AVL trees* are a way of implementing tree balancing.
  - *Red-black trees* are a type of self-balancing binary search tree. They require a bit less memory than AVL trees, and can rebalance faster (which means faster insertions and removals), so they are often used in situations where the tree will be modified frequently.
  - *B-Trees*: A self-balancing search tree (not a binary search tree) that is commonly used on disks or other storage devices. It is similar to a red-black tree, but uses fewer I/O operations.
  - *Dijkstra's algorithm* is used for calculating the shortest path in graph.
  - *A**: Find the least-cost path between a source node and a goal node (or one of several goal nodes). It extends Dijkstra's algorithm and achieves better performance by using heuristics.
  - *Floyd-Warshall algorithm*: Finds the shortest paths in a weighted graph with positive or negative weight edges (but no negative weight cycles).
  - *Bellman-Ford algorithm*: Finds the shortest paths from a single node in a weighted directed graph with positive and negative edges.
  - *Graph coloring*: A way of coloring the nodes in a graph such that no two adjacent vertices have the same color. There are various algorithms to do things like determine if a graph can be colored with only K colors.
  - *P, NP, and NP-Complete*: P, NP, and NP-Complete refer to classes of problems.
    - P problems are problems that can be quickly solved (where "quickly" means in polynomial time).

* NP problems are those for which the problem instances, where the answer is “yes”, have proofs verifiable in polynomial time; so their solution can be quickly verified. An equivalent definition of NP is the set of decision problems solvable in polynomial time by a non-deterministic Turing machine. This definition is the basis for the abbreviation NP; "non-deterministic, polynomial time." These two definitions are equivalent because the algorithm based on the Turing machine consists of two phases, the first of which consists of a guess about the solution, which is generated in a non-deterministic way, while the second phase consists of a deterministic algorithm that verifies if the guess is a solution to the problem.
* It is easy to see that the complexity class P (all problems solvable, deterministically, in polynomial time) is contained in NP (problems where solutions can be verified in polynomial time), because if a problem is solvable in polynomial time then a solution is also verifiable in polynomial time by simply solving the problem. But NP contains many more problems, the hardest of which are called NP-complete problems.
* NP-Complete problems are a subset of NP problems that can all be reduced to each other (that is, if you found a solution to one problem, you could tweak the solution to solve other problems in the set in polynomial time).

Concepts

- *Bit Manipulation*
- There are 2 right shifts: logical shift and it fills 0 from left side, and arithmetic shift and it keeps the sign bit - so it divides a number basically by 2).
  - Sequence of all 1s in signed integer (as in binary number) is -1.
- *Memory (Stack vs. Heap)*

  Stack is used for static memory allocation and Heap for dynamic memory allocation, both stored in the computer’s RAM. In a multi-threaded situation each thread will have its own completely independent stack but they will share the heap. Stack is thread specific and Heap is application specific. The stack is attached to a thread, so when the thread exits the stack is reclaimed. The heap is typically allocated at application start-up by the runtime, and is reclaimed when the application (technically process) exits. The size of the stack is set when a thread is created. The size of the heap is set on application start-up, but can grow as space is needed (the allocator requests more memory from the operating system).

  By the way, the amount of memory that get’s assigned to an application depends on the computer’s architecture and will vary across most devices, but the variable that remains constant is the five parts of an application’s memory which are the heap,

stack, initialized data segment (global and static variables that are initialized when a file gets compiled), uninitialized data segment (all global and static variables that are initialized to zero or do not have explicit initialization in source code), and the text segment (also known as the code segment, contains the machine instructions which make up your program. The text segment is often read-only and prevents a program from accidentally modifying its instructions).

Stack

- Is a special region of your computer's memory that stores temporary variables created by each function (including the main() function). The stack is a "LIFO" (last in, first out) data structure, that is managed and optimized by the CPU quite closely. Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, all of the variables pushed onto the stack by that function, are freed (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables.
- The advantage of using the stack to store variables, is that memory is managed for you. You don't have to allocate memory by hand, or free it once you don't need it any more. What's more, because the CPU organizes stack memory so efficiently, reading from and writing to stack variables is very fast.
- A key to understanding the stack is the notion that when a function exits, all of its variables are popped off of the stack (and hence lost forever). Thus stack variables are local in nature.
- Summary: the stack grows and shrinks as functions push and pop local variables there is no need to manage the memory yourself, variables are allocated and freed automatically the stack has size limits stack variables only exist while the function that created them, is running.

Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is a more free-floating region of memory (and is larger). To allocate memory on the heap, you must use malloc() or calloc(), which are built-in C functions. Once you have allocated memory on the heap, you are responsible for using free() to deallocate that memory once you don't need it any more. If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes). As we will see in the debugging section, there is a tool called valgrind that can help you detect memory leaks.
- Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer). Heap memory is slightly slower to be read from and written to, because one has to use

pointers to access memory on the heap. Heap size is only limited by the size of virtual memory.

- Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.
- Another performance hit for the heap is that the heap, being mostly a global resource, typically has to be multi-threading safe, i.e. each allocation and deallocation needs to be - typically - synchronized with "all" other heap accesses in the program.

- *Big O (Time & Space complexity)*
- Note1: $1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} = n^2$
- Note2: $2^0 + 2^1 + 2^2 + ... + 2^n = 2^{n+1} - 1$
- Note3: log conversion (base) from base 2 to base b: $log_b k = \frac{log_2 k}{log_2 b}$

Testing

At their surface, testing questions seem like they're just about coming up with an extensive list of test cases. And to some extent, that's right. You do need to come up with a reasonable list of test cases. The interviewers want to do the following from you:

- Big picture understanding.
- Knowing how the pieces fit together.
- Organization.
- Practical point of view.

7.5 Machine Learning Interview Preparation

- Top companies for Data Science: Microsoft, IBM, Facebook, Amazon, Google, Accenture, Apple, Capital One, Uber, B.A Hamilton.

7.6 Topics

Good sources: Machine Learning Interview Preparation (from Udacity).

- **Computer Science Fundamentals and Programming Techniques**

  All from above:

  - Data structures: Lists, stacks, queues, strings, hash maps, vectors, matrices, classes & objects, trees, graphs, etc.
  - Algorithms: Recursion, searching, sorting, optimization, dynamic programming, etc.
  - Computability and complexity: P vs. NP, NP-complete problems, big-O notation, approximate algorithms, etc.
  - Computer architecture: Memory, cache, bandwidth, threads & processes, deadlocks, etc.

- **Probability and Statistics**
  - Basic probability: Conditional probability, Bayes rule, likelihood, independence, etc.
  - Probabilistic models: Bayes Nets, Markov Decision Processes, Hidden Markov Models, etc.
  - Statistical measures: Mean, median, mode, variance, population parameters vs. sample statistics etc.
  - Proximity and error metrics: Cosine similarity, mean-squared error, Manhattan and Euclidean distance, log-loss, etc.
  - Distributions and random sampling: Uniform, normal, binomial, Poisson, etc.
  - Analysis methods: ANOVA, hypothesis testing, factor analysis, etc.

- **Data Modeling and Evaluation**
  - Data preprocessing: Munging/wrangling, transforming, aggregating, etc.
  - Pattern recognition: Correlations, clusters, trends, outliers & anomalies, etc.
  - Dimensionality reduction: Eigenvectors, Principal Component Analysis, etc.
  - Prediction: classification, regression, sequence prediction, suitable error/accuracy metrics.
  - Evaluation: Training-testing split, sequential vs. randomized cross-validation, etc.

- **Applying Machine Learning Algorithms and Libraries**
  - Models: Parametric vs. non-parametric, decision tree, nearest neighbor, neural net, support vector machine, ensemble of multiple models, etc.

  - Learning procedure: Linear regression, gradient descent, genetic algorithms, bagging, boosting, and other model-specific methods; regularization, hyperparameter tuning, etc.
  - Trade-offs and gotchas: Relative advantages and disadvantages, bias and variance, overfitting and underfitting, vanishing/exploding gradients, missing data, data leakage, etc.

- **Software Engineering and System Design**
  - Software interface: Library calls, REST APIs, data collection endpoints, database queries, etc.
  - User interface: Capturing user inputs & application events, displaying results & visualization, etc.
  - Scalability: Map-reduce, distributed processing, etc.
  - Deployment: Cloud hosting, containers & instances, microservices, etc.

8 Linux/Unix

The UNIX operating system requires that every IO device driver provide five standard functions: *open, close, read, write*, and *seek*. The signatures of those functions must be identical for every IO driver.

8.1 General Tools and Packages

- Compilers: cc, gcc, c89, as, gas
- Debugging: dbx, debug, dgb
- Libraries: ar, ld, nm, objdump
- Linkers: ld, ldd
- Optimizations: prof, gprof
- Projects
  - make (recommended content and order: all, install, clean, doc, depend),
  - RCS - an archive of file versions
    - free, enterprise is SCCS
    - CVS - RCS for big projects

8.2 Helper tools for smaller scripting

awk

basename

cat

cxref

cut

date

dmesg

exec

eval

find

flock

grep

getent

getopts

chmod

chown

gunzip

kill

less

lsb_release (and /etc/lsb-release)

lshw

mktemp

mv

parallel

passwd

pidof

popd

pgrep

pushd

ps

readonly

rm

rsync

scp

sed

shift

shred[1]

size

sort

source (vs bash script.sh vs ./script.sh)

sha256sum

ssh

strip

su

tail

tar

top

timeout

tr

type

uniq

wait

wc

wget

which

who

zcat

zip

8.3 Networking

- Iptables:

```
$ iptables -A INPUT -p tcp --dport 22 -j ACCEPT
$ iptables -L
```

- Netstat:

```
$ netstat -tulpn | less
```

- Mounting remote directory of server X:

```
$ sshfs X:/path/dir .
```

- SSH:

```
$ /sbin/service sshd status
$ sudo systemctl start sshd.service
```

- ping

8.4 Services and Processes

- Command *service*:

```
$ service sshd status

$ service sshd start
```

- Command *initctl*:

```
$ initctl restart X

$ initctl status X
```

- Command *journalctl*:

```
$ journalctl # system journal showing cronjobs etc.
```

- List of cronjobs:

```
$ cronjob -l
```

- Kill all stopped processes:

```
$ kill -9 $(jobs -p)
```

- Command *nohup* - executing commands after we left shell prompt. It's possible to do prioritization, check logs, etc.[2].
  - nohup vs & at the end - "*&*" sends SIGHUP and will kill a process, but *nohup* will ignore this signal and process will run even after a user's logout.

[2] http://www.cyberciti.biz/tips/nohup-execute-commands-after-you-exit-from-a-shell-prompt.html

8.5 Bash

- You can write unit-tests for Bash with BATS[3].
- Update of all the packages you have installed:

```
$ pip3 freeze | cut -d "=" -f 1 > requirements.txt
$ sudo pip3 install --upgrade -r requirements.txt
```

- declare, local, global, export

[3] https://github.com/sstephenson/bats

9 Mastering Git

There are many tools that offer code collaboration and version control using Git, such as:

- Gitlab
- GitHub
- Gitolite
- Bitbucket

9.1 Basics & General

- Delete the last pushed commit:

```
$ git reset --hard HEAD~1
$ git push origin HEAD --force
```

- Commands *git remote prune* and *git fetch --prune* do the same thing: delete the refs to branches that don't exist on the remote.[1] This is highly desirable when working in a team workflow in which remote branches are deleted after merge to master. The second command, *git fetch --prune* will connect to the remote and fetch the latest remote state before pruning. It is essentially a combination of commands:

```
$ git fetch --all && git remote prune
```

- The generic *git prune* command deletes locally detached commits. It basically cleans up unreachable or "orphaned" Git objects. Unreachable objects are those that are inaccessible by any refs. Any commit that cannot be accessed through a branch or tag is considered unreachable.

- There is a fast cloning possibility for saving time and disk space. It copies only recent revisions. Git's shallow clone option allows you to pull down only the latest n commits of the repository's history: *git clone --depth [depth] [remote-url]*

  For more options how to deal with repositories that have big files, or very long commit commit history, read the following tutorial: https://www.atlassian.com/git/tutorials/big-repositories

[1] urlhttps://www.atlassian.com/git/tutorials/git-prune

- Delete untracked files in the local working directory, but only with dry run where nothing is actually deleted: *git clean -n*
- **Relative Referencing**[2]
  - Sometimes it's useful to be able to indicate a revision relative to a known position, like HEAD or a branch name. Git provides two operators that, while similar, behave slightly differently.
  - The first of these is the tilde (~) operator. Git uses tilde to point to a parent of a commit, so *HEAD~* indicates the revision before the last one committed. To move back further, you use a number after the tilde: *HEAD~3* takes you back three levels. This works great until we run into merges. Merge commits have two parents, so the ~ just selects the first one. While that works sometimes, there are times when you want to specify the second or later parent. That's why Git has the caret (^) operator.
  - The ^ operator moves to a specific parent of the specified revision. You use a number to indicate which parent. So *HEAD^2* tells Git to select the second parent of the last one committed, not the "grandparent." It can be repeated to move back further: *HEAD^2^^* takes you back three levels, selecting the second parent on the first step.
  - These two operators can be combined together.
- **Revision Ranges**
  - The "double dot" method for specifying ranges looks like it sounds: *git log b05022238cdf08..60f89368787f0e*
  - Triple dot notation uses 3 dots between the revision specifiers. This works in a similar manner to the double dot notation except that it shows all commits that are in either revision that are not included in both revisions.
- **Handling Interruptions**
  - Here, *git stash* and *git stash pop* are your friends. Along with some others such as *git list, git stash apply, git stash show -p stash@{N}, git stash drop* and *git stash pop.* They are all very simple commands.
- **Ref**
  - A ref is an indirect way of referring to a commit. You can think of it as a user-friendly alias for a commit hash. This is Git's internal mechanism of representing branches and tags.
  - Refs are stored as normal text files in the *.git/refs* directory.
  - To change the location of the master branch, all Git has to do is change the contents of the *refs/heads/master* file. Similarly, creating a new branch is simply a matter of writing a commit hash to a new file.

[2] https://realpython.com/advanced-git-for-pythonistas/

  - When passing a ref to a Git command, you can either define the full name of the ref, or use a short name and let Git search for a matching ref. So *git show some-feature* is equal to *git show refs/heads/some-feature.*

- **Refspec**
  - It maps a branch in the local repository to a branch in a remote repository. This makes it possible to manage remote branches using local Git commands and to configure some advanced git push and git fetch behavior.
  - A refspec is specified as *[+]<src>:<dst>*. The *<src>* parameter is the source branch in the local repository, and the *<dst>* parameter is the destination branch in the remote repository. The optional *+* sign is for forcing the remote repository to perform a non-fast-forward update.
  - Refspecs can be used with the git push command to give a different name to the remote branch.
  - For example, the following command pushes the master branch to the origin remote repo like an ordinary *git push*, but it uses *qa-master* as the name for the branch in the origin repo. This is useful for QA teams that need to push their own branches to a remote repo:

    git push origin master:refs/heads/qa-master

- **Reflog**
  - The reflog is Git's safety net. It records almost every change you make in your repository, regardless of whether you committed a snapshot or not. You can think of it as a chronological history of everything you've done in your local repo.
  - So it shows a log of changes to the local repository's HEAD. Good for finding lost work.[3]
  - To view the reflog, run the *git reflog* command.
  - If you want to recover the last change before the last commit from reflog, you can do it with *git checkout HEAD@{1}* command. See a practical example with much verbose explanation here: https://www.atlassian.com/git/tutorials/refs-and-the-reflog

- **Git Log**[4]
  - Formatting log output
    - Seeing all tags, branches, etc in high-level overview: *git log --oneline --decorate*

[3] https://towardsdatascience.com/10-git-commands-you-should-know-df54bea1595c
[4] https://www.atlassian.com/git/tutorials/git-log

    * The *--stat* option displays the number of insertions and deletions to each file altered by each commit. Or, for actual changes, pass *-p* parameter instead.
    * Command *git shortlog* is intended for creating release announcements. It groups each commit by author and displays the first line of each commit message. This is an easy way to see who's been working on what. By default, git shortlog sorts the output by author name, but you can also pass the *-n* option to sort by the number of commits per author.
    * The *--graph* option draws an ASCII graph representing the branch structure of the commit history. This is commonly used in conjunction with the *--oneline* and *--decorate* commands to make it easier to see which commit belongs to which branch. While this is a nice option for simple repositories, you're probably better off with a more full-featured visualization tool such as *gitk* or *Sourcetree*.
    * Or you can use custom formatting, with *--pretty=format:"<string>"* option.
  - Filtering the commit history
    * By amount, with parameter *-n <N>*
    * By date, with parameter for example:
      · *--after="2014-7-1"*
      · *--after="yesterday"*
      · *--after="2014-7-1" --before="2014-7-4"* for outputting all commits in between 2 dates
      · *--since* and *--until* flags are synonymous with *--after* and *--before*, respectively
    * By author, wth *--author="John"* parameter. You can also use regular expressions here.
    * By commit message, with *--grep="pattern"* parameter (you can use *-i* to ignore case differences while pattern matching). This parameter works as the previous one.
    * By files, with *-- <file1> <file2>* parameter.
    * Filtering out displaying merge commits can be achieved by passing the *--no-merges* option.

- **Git Blame**
  - *C*ommand *git blame <file>* is like annotation in PyCharm. You can see who changed what and when in a given file. It can be quite useful.

- **Git Patch**

  - The first time a file is committed to a project in GIT, a copy is stored. For all commits after that, GIT essentially stores instructions telling it how to transform the previous version of the project to the newly committed version (these are diffs).[5]
  - Whenever you checkout a branch, GIT will basically start at the original state of the project, and apply all of these diffs in order, to to get to the desired state.
  - For creating a patch from some commit to another commit, we will use the following: *git diff <from-commit> <to-commit> > patch.diff*
  - After patch file has been created, applying it is easy. Make sure that the branch you have checked out is the one that you want to apply the patch to. Then you can apply the patch using *git apply patch.diff* command.
  - **Warning:** Although applying a patch in this way will exactly replicate content, no commit history will be replicated. This means that even if the patch you create spans several commits, it will appear as a single set of changes when applied. You will lose both the knowledge of how the commits were broken up and also the messages for each commit. Applying the patch did not commit the changes, nor did it bring any of the commit history associated with these changes with it.

- **Git Merge-base**
  - The command *git merge-base <master> <feature>* will determine the most recent common commit between 2 branches, in this case between master and feature branch.

[5] https://www.thegeekstuff.com/2014/03/git-patch-create-and-apply/

9.2 Commit message

- A team's approach to its commit log should be no different. In order to create a useful revision history, teams should first agree on a commit message convention that defines at least the following three things:
  - **Style**. Markup syntax, wrap margins, grammar, capitalization, punctuation. Spell these things out, remove the guesswork, and make it all as simple as possible. The end result will be a remarkably consistent log that's not only a pleasure to read but that actually does get read on a regular basis.
  - **Content**. What kind of information should the body of the commit message (if any) contain? What should it not contain?
  - **Metadata**. How should issue tracking IDs, pull request numbers, etc. be referenced?
- **The seven rules of a great Git commit message**
  1. Separate subject from body with a blank line
  2. Limit the subject line to 50 characters. If you're having a hard time summarizing, you might be committing too many changes at once.
  3. Capitalize the subject line
  4. Do not end the subject line with a period
  5. Use the imperative mood in the subject line. Spoken or written as if giving a command or instruction. Use of the imperative is important only in the subject line. You can relax this restriction when you're writing the body.
  6. Wrap the body at 72 characters
  7. Use the body to explain what and why vs. how. How is not important, code is self-explanatory (if there is very sophisticated change, use code documentation).

9.3 Submodules

Clone and Init

```
$ git clone git@git_hostname:submoduleX

$ git submodule init -- update
```

Pull all submodules from master branch

```
$ git submodule foreach git pull origin master
```

How to delete a submodule

- remove entry from *.gitmodule*
- remove entry from *.git/config* (not necessarily)
- remove path created for a given submodule (but be careful here, there cannot be *'/'* n the end of the path)

```
$ git rm --cached [path/module]
```

Discard changes in submodule

If something went wrong, we can discard changes in submodule and initialize it again. We just have to go to a given directory, and type:

```
$ git submodule deinit -f .

$ git submodule update --init
```

Adding a submodule

```
$ git submodule add lsulak@git_hostname:repo libs/repo
```

9.4 Rebasing

- The aim of rebasing and merging is the same, but both commands are doing things differently. And also history of commits will look differently using one command or another. With rebasing, you may have a perfectly linear project history.[6]
- Let's have an example of incorporating changes from *master* branch to my local *feature* branch (so *master* branch was changed since we checkout the *feature* branch). This will put the *feature* branch on the top of master (so we are rebasing *feature* branch onto *master* branch), but for each commit from the original *feature* branch there will be a change - project history is re-written by creating branch new commits for each original commit in feature branch (alternative is merging with command *git merge feature master*, but that creates a new merge commit in the feature branch, and that is ugly and it master branch is very active, it can pollute your feature branch a lot; but it is easy and non-destructive):

```
$ git checkout feature

$ git rebase master
```

- Rebasing loses the context provided by a merge commit - you can't see when upstream changes were incorporated into the feature (but that is probably not needed in most scenarios).
- **Do not use rebasing on public branches** (=use rebasing only on your branches). From the previous example, rebasing master onto feature - that is bad! The rebase moves all of the commits in master onto the top of feature. The problem is that this only happened in your repository. All of the other developers are still working with the original master. Since rebasing results in brand new commits, Git will think that your master branch's history has diverged from everybody else's. The only way to synchronize the 2 master branches is to merge them back together, resulting in an extra merge commit and 2 sets of commits that contain the same changes = ugly and confusing.
- **So before you use rebasing, ask yourself**: *"Is anyone else looking at this branch?"* If the answer is yes, use a different approach instead of rebasing. But if you and some other person are developing code in a single feature branch, and you want to incorporate his changes into yours, you can still do it - because you will not change or move his commits, only yours (its like "add my changes to what he has already done").
- It is possible to push changes after rebasing with parameter –force (but be careful here). One of the only times you should be force-pushing is when you've performed

[6] https://www.atlassian.com/git/tutorials/merging-vs-rebasing

a local cleanup after you've pushed a private feature branch to a remote repository (e.g., for backup purposes). This is like saying, "Oops, I didn't really want to push that original version of the feature branch. Take the current one instead." Again, it's important that nobody is working off of the commits from the original version of the feature branch.

- It is a good idea to clean up your code with an interactive rebase before submitting your PR or MR (because after the submitting, the branch is public and you would re-write the commit history...but this is violated by lot of people).
- Once all is ready, you can just merge feature branch into master. By performing a rebase before the merge, you're assured that the merge will be fast-forwarded, resulting in a perfectly linear history - because there will be no conflicts or new changes from master.

Interactive rebasing

- Simple flow with local feature branch:

```
$ git rebase --interactive [HEAD~7 or other_branch_name]

<some changes>

$ git rebase --continue

$ git push origin branch --force
```

- There are several options in interactive rebase:
  - squash - combination of multiple commits
  - pick - commit choose/reorder
  - fixup - merge commit with one above and commit message is discarded
  - edit - commit edit/split
  - reword - fix commit message only
- If something went wrong, you can abort rebasing, or you can use command *git reflog*.

```
$ git rebase --abort
```

- Further changes = new commits, and then combination with staged changes with the previous commit (git rebase –continue is for to moving branch HEAD back to the commit we originally had, while also including the new changes we added):

```
$ git commit --amend

$ git rebase --continue
```

- If we want to add some new changes to the previous commit, we can use command *git amend.* So it will add your staged changes to the most recent commit. If nothing is staged, this command just allows you to edit the most recent commit message. Only use this command if the commit has not been integrated into the remote master branch! So first we will perform some changes, then we will put the changes into staging area, and do the following:

```
$ git commit --amend

$ git push –force-with-lease <remote> <branch>
```

9.5 Feature Branch

Merging of a feature branch:

```
$ git checkout some-feature git pull origin devel

$ git checkout devel git merge --no-ff --log some-feature

$ git push origin devel

$ git branch -d some-feature git push origin --delete some-feature
```

9.6 Reset

Resetting changes back (only those that weren't already pushed). So this is for local branch. It is possible to work with individual files. This is a permanent undo. For example, changing a branch to another commit, for example 2 commits back:

```
$ git checkout hotfix
$ git reset HEAD~2
```

- --soft - staged area a working directory are not affected
- --mixed - default choice, staged area will be changed according to commit, but not working directory
- --reset - all is based on a given commit

These 3 parameters do not work with an individual file.

Examples

- Reset staged area:

```
$ git reset --mixed HEAD
```

- Discard staged and unstaged changes since the most recent commit:

```
$ git reset --hard HEAD
```

- Unstage file *foo.py,* but changes will be still present in the working directory:

```
$ git reset HEAD foo.py
```

- Discard commits in a private branch or throw away uncommited changes:

```
$ git reset <Commit-level>
```

9.7 Checkout

A checkout is an operation that moves the HEAD ref pointer to a specified commit. It is possible to work with an individual file, commit, or a branch.

Examples

- Switch between branches or inspect old snapshots:

```
$ git checkout <Commit-level>
```

- Discard changes in the working directory:

```
$ git checkout <File-level>
```

9.8 Revert

- A revert is an operation that takes a specified commit and creates a new commit which inverses the specified commit. git revert can only be run at a commit level scope and has no file level functionality. Revert is considered a safe operation for "public undos" as it creates new history which can be shared remotely and doesn't overwrite history remote team members may be dependent on. So it is safe operation for already pushed commits.
- Contrast this with *git reset*, which does alter the existing commit history. For this reason, *git revert* should be used to undo changes on a public branch, and *git reset* should be reserved for undoing changes on a private branch.
- You can also think of *git revert* as a tool for undoing committed changes, while *git reset* HEAD is for undoing uncommitted changes.

Examples

- Undo commits in a public branch:

```
$ git revert <Commit-level>
```

- reverting a newly created commit:

```
$ git revert HEAD
```

- Find a branch which is currently on HEAD:

```
$ git symbolic-ref --short HEAD
```

9.9 Tags

Examples

- Update:

```
$ git fetch --tags
```

- New tag:

```
$ git tag -a v1.4 -m "comment"
$ git push --tags
```

- Remove tag on some remote branch:

```
$ git push origin :refs/tags/<tagname>
```

- Replace tag on the last commit:

```
$ git tag -fa <tagname>
```

- Push tag onto a remote origin:

```
$ git push origin master --tags
```

9.10 Git Hooks

- Git hooks are scripts that run automatically every time a particular event occurs in a Git repository. They let you customize Git's internal behavior and trigger customizable actions at key points in the development life cycle.[7]
- Common use cases for Git hooks include encouraging a commit policy, altering the project environment depending on the state of the repository, and implementing continuous integration workflows. But, since scripts are infinitely customizable, you can use Git hooks to automate or optimize virtually any aspect of your development workflow.
- All Git hooks are ordinary scripts that Git executes when certain events occur in the repository. They are placed in *.git/hooks* directory of every Git repository.
- All of the *pre-hooks* let you alter the action that's about to take place, while the *post-hooks* are used only for notifications.

Local Hooks

Local hooks affect only the repository in which they reside. As you read through this section, remember that each developer can alter their own local hooks, so you can't use them as a way to enforce a commit policy.

- For entire commit life cycle:
  - *pre-commit* - this one is executed every time you run git commit before Git asks the developer for a commit message or generates a commit object.
  - *prepare-commit-msg* - this is called after the pre-commit hook to populate the text editor with a commit message. This is a good place to alter the automatically generated commit messages for squashed or merged commits.
  - *commit-msg* - this hook is much like the previous one, but it's called after the user enters a commit message. This is an appropriate place to warn developers that their message doesn't adhere to your team's standards.
  - *post-commit* - this one is called immediately after the previous hook. It can't change the outcome of the git commit operation, so it's used primarily for notification purposes.
- Some other extra actions or safety checks
  - *post-checkout* - this hook works a lot like the *post-commit* hook, but it's called whenever you successfully check out a reference with *git checkout*. This is nice for clearing out your working directory of generated files that would otherwise cause confusion.

[7] https://www.atlassian.com/git/tutorials/git-hooks

  - *pre-rebase* - this one is called before *git rebase* changes anything, making it a good place to make sure something terrible isn't about to happen.

Server-side

These reside in server-side repositories (e.g., a central repository, or a developer's public repository). When attached to the official repository, some of these can serve as a way to enforce policy by rejecting certain commits. All of these hooks let you react to different stages of the *git push* process.

- *pre-receive* - this is executed immediately after *git push*, so it is possible to reject changes.
- *update* - this hook is called after the previous one. It's still called before anything is actually updated, but it's called separately for each ref that was pushed.
- *post-receive* - this is called after a successful push operation, making it a good place to perform notifications. For many workflows, this is a better place to trigger notifications than *post-commit* because the changes are available on a public server instead of residing only on the user's local machine. Emailing other developers and triggering a continuous integration system are common use cases for post-receive.

9.11 Cherry Pick

- This command enables arbitrary Git commits to be picked by reference and appended to the current working HEAD.
- Cherry picking is the act of picking a commit from a branch and applying it to another. git cherry-pick can be useful for undoing changes.
- For example, say a commit is accidentally made to the wrong branch. You can switch to the correct branch and cherry-pick the commit to where it should belong.
- Cherry picking can cause duplicate commits and many scenarios where cherry picking would work, traditional merges are preferred instead. With that said git cherry-pick is a handy tool for a few scenarios.
- The most common use case is probably bug fix: a developer creates an explicit commit patching that bug. This new patch commit can be cherry-picked directly to the master branch to fix the bug before it effects more users.
- Let's say, that we want to use commit *A* in master branch. So we will checkout master branch, and then use command *git cherry-pick A*, where, of course, *A* is an SHA of a given commit.[8]
- Cherry picking is another method for moving commits from one branch to another. Unlike merging and rebasing, with cherry-picking you specify exactly which commits you mean. The easiest way to do this is just specifying a single SHA:[9]

  git cherry-pick 4a4f4492ded256aa7b29bf5176a17f9eda66efbb

  This tells Git to take the changes that went into 4a4f449 and apply them to the current branch. This feature can be very handy when you want a specific change but not the entire branch that change was made on.

[8] https://www.atlassian.com/git/tutorials/cherry-pick
[9] https://realpython.com/python-git-github-intro/

9.12 GitLab

Examples

- https://about.gitlab.com/2016/10/25/gitlab-workflow-an-overview/
- Create own branch and start working on feature:

```
$ git checkout devel
$ git pull
$ git checkout -b feature/1234_my_cool_feature
```

- If you are ready for review, open a merge request. Please create a merge request against devel branch as target in original repository.
- Updating merge request, using force push:

```
$ touch new_file.txt
$ git add new_file.txt
$ git commit -m 'new file'
$ git push --force
```

Pro tip: You can amend existing commits instead of adding news at the end by interactive rebase (git rebase -i master 1234_branch)

- On merge request conflicts:

```
$ git checkout devel
$ git pull
$ git rebase devel 1234_branch <resolve conflicts>
$ git push --force
```

9.13 Git branching workflows

Inspired or taken from https://medium.com/@patrickporto/4-branching-workflows-for-git-30d0aaee7bf and https://www.atlassian.com/git/tutorials/comparing-workflows.

GitFlow

- This flow is based on 2 main branches
  - **master** - production code, all development code is merged into master after some time.
  - **develop** - pre-production code, where all finished features are merged here.
- Also, during development cycle, a variety of supporting branches are used:
  - **feature-*** - new features for upcoming releases. May branch off **from develop** and must be merged **into develop.**
  - **hotfix-*** - necessary to act upon an undesired status of **master**. May branch off **from master** and must be merged **into master** and **develop**.
  - **release-*** - these branches support preparation of a new production release. They allow many minor bugs to be fixed and also preparing metadata for a release. May be branch off **from develop** and must be merged **into master** and **develop**.
- This is also known as "A successful Git branching model", from 2010. It was one of the first proposals to use git branches and it has gotten a lot of attention. However, it has few problems:[10]
  - Developers must use the develop branch and not master, master is reserved for code that is released to production. It is a convention to call your default branch master and to mostly branch from and merge to this. Since most tools automatically make the master branch the default one and display that one by default it is annoying to have to switch to another one.
  - The complexity introduced by the hotfix and release branches is another problem. These branches can be a good idea for some organizations but are overkill for the vast majority of them. Nowadays most organizations practice continuous delivery which means that your default branch can be deployed. This means that hotfix and release branches can be prevented including all the ceremony they introduce. An example of this ceremony is the merging back of release branches. Though specialized tools do exist to solve this, they require documentation and add complexity. Frequently developers make a mistake and for example changes are only merged into master and not into the develop branch. The root cause of these errors is that git flow is too complex for most

[10] https://barro.github.io/2016/02/a-succesful-git-branching-model-considered-harmful/

of the use cases. And doing releases doesn't automatically mean also doing hotfixes.

- Using individual (long lived) branches for features also make it harder to ensure that everything works together when changes are merged back together. This is especially pronounced in today's world where continuous integration should be the default practice of software development regardless how big the project is. By integrating all changes together regularly you'll avoid big integration issues that waste a lot of time to resolve, especially for bigger projects with hundreds or thousands of developers.

- A lot simpler is GitHub flow or GitLab flow. Or the cactus model[11] (this is much more simple and makes sure that continuous integration principles are used).

- **Pros**
  - Clean state of branches at any given moment in lifecycle of project.
  - It has many extensions and support on most used git tools.[12]
  - Ideal when there are multiple versions of production.

- **Cons**
  - Git history becomes unreadable.[13]
  - Master/develop split is considered redundant and makes CD and CI harder. It is very complex and over-engineered.
  - **It isn't recommended when it is needed to maintain a single version of production.**

[11] https://barro.github.io/2016/02/a-succesful-git-branching-model-considered-harmful/
[12] https://github.com/nvie/gitflow
[13] https://www.endoflineblog.com/gitflow-considered-harmful

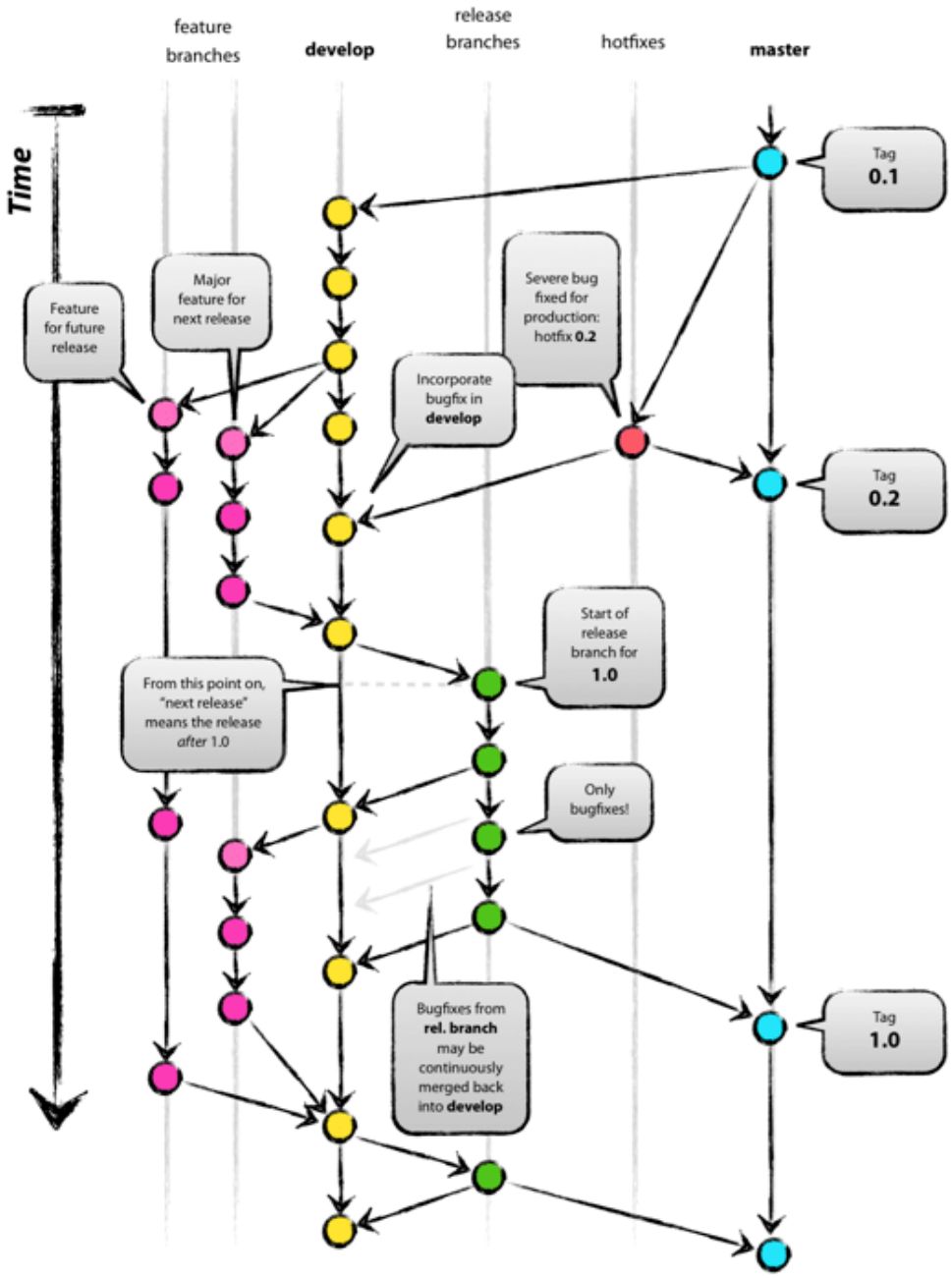

Figure 9.1: Git flow diagram.

GitHub Flow

- Suppose you have a staging environment (**master**), a pre-production environment (**MR from the master branch to pre-production branch**) and a production

environment (merging pre-production into production).

- GitHub flow does assume you are able to deploy to production every time you merge a feature branch. There are many cases where this is not possible.
- This is a lightweight workflow created by GitHub in 2011 and it respects the following principles:
  1. Anything in *master* branch is deployable.
  2. If you are working on something new, create a branch off from *master* and give a proper descriptive name.
  3. Commit to that new branch locally and regularly push your work to the same named remote branch on the server.
  4. When you need a feedback or help, or all changes are ready for merging, open a **PR** (pull request).
  5. After **someone else** has **reviewed and signed off** on the feature, **YOU can merge it into master**.
  6. Once it is merged and pushed into master, you can and **should** deploy immediately.
- **Pros**
  - Friendly for CD and CI.
  - A simpler alternative to Git Flow.
  - **Ideal when it needs to maintain a single version in production.**
- **Cons**
  - Production code can become unstable most easily.
  - Not adequate when it is needed to have release plans.
  - It doesn't resolve anything about deployment, environment, releases, and issues.
  - Not recommended when multiple versions in production are needed.

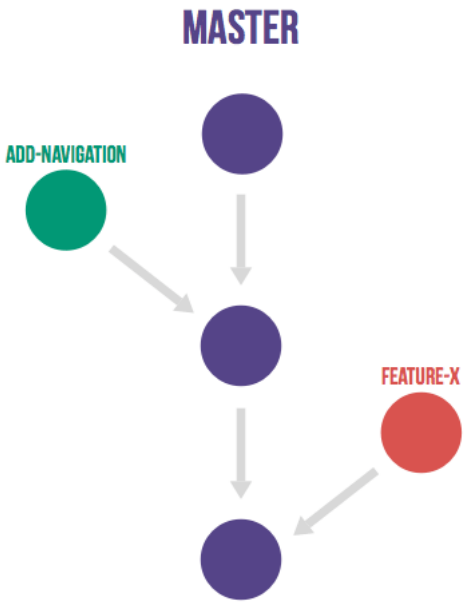

Figure 9.2: GitHub flow diagram.

GitLab Flow

- Combination of feature-driven development and feature branches with issue tracking.
- Created by GitLab in 2014.[14]
- **There can be a project that isn't able to deploy to production every time you merge a feature branch** - in that case, GitHub Flow is not a way, GitLab is!
- BTW, merge or pull requests are created in a git management application and ask an assigned person to merge two branches. Tools such as GitHub and Bitbucket choose the name pull request since the first manual action would be to pull the feature branch. Tools such as GitLab and others choose the name merge request since that is the final action that is requested of the assignee. In this article we'll refer to them as merge requests.
- Merge requests always create a merge commit even when the commit could be added without one. This merge strategy is called "no fast-forward" in git. Git history is not linear, but the advantage is that reverting the entire feature requires reverting only 1 commit (the merge commit).

[14] https://about.gitlab.com/2014/09/29/gitlab-flow/?fbclid=IwAR3qjO-fttOzRLytF6ln6PheiGYE9wwZji3MHgU-iOtPjBbNOC1TGpjAh3I

- It respects the following principles:[15]
  1. Use **feature branches**, no direct commits on **master**.
  2. **Test all commits**, not only ones on **master.**
  3. **Run all the tests on all commits** (you may have them in parallel if they are running longer than 5min).
  4. Perform code reviews before merging into **master.**
  5. Deployments are automatic, based on branches or tags.
  6. Tags are set by the user, not by CI. A user sets a tag and, based on that, the CI will perform an action. You shouldn't have the CI change the repository.
  7. Releases are based on tags. If you tag something, that creates a new release.
  8. Pushed commits are never rebased. If you push to a public branch you shouldn't rebase it since that makes it hard to follow what you're improving, what the test results were, and it breaks cherry-picking. Code should be clean, history should be realistic
  9. Everyone starts from **master**, and targets **master**. You don't have any long branches. You check out master, build your feature, create your merge request, and target master again. You should do your complete review before you merge, and not have any intermediate stages.
  10. Fix bugs in **master** first and release branches afterwards. Fix it in master, then cherry-pick it into another patch-release branch. If you find a bug, the worst thing you can do is fix it in the just-released version, and not fix it in master.
  11. Commit messages reflect intent. You should not only say what you did, but also why you did it. It's even more useful if you explain why you did this over any other options.

- **Pros**
  - It defines how to make CI and CD.
  - Git history will be cleaner, less messy and more readable.
  - Ideal when it is needed to have a single version in production.

- **Cons**
  - More complex than GitHub Flow.
  - If multiple versions of production are needed, this flow can become very complex.

[15] https://about.gitlab.com/2016/07/27/the-11-rules-of-gitlab-flow/

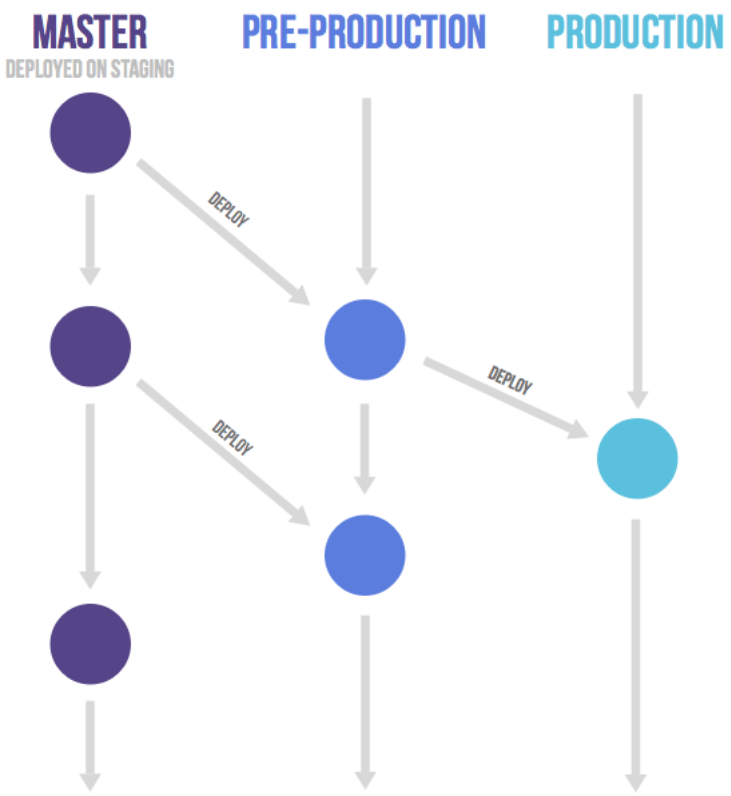

Figure 9.3: Environment branches with GitLab flow

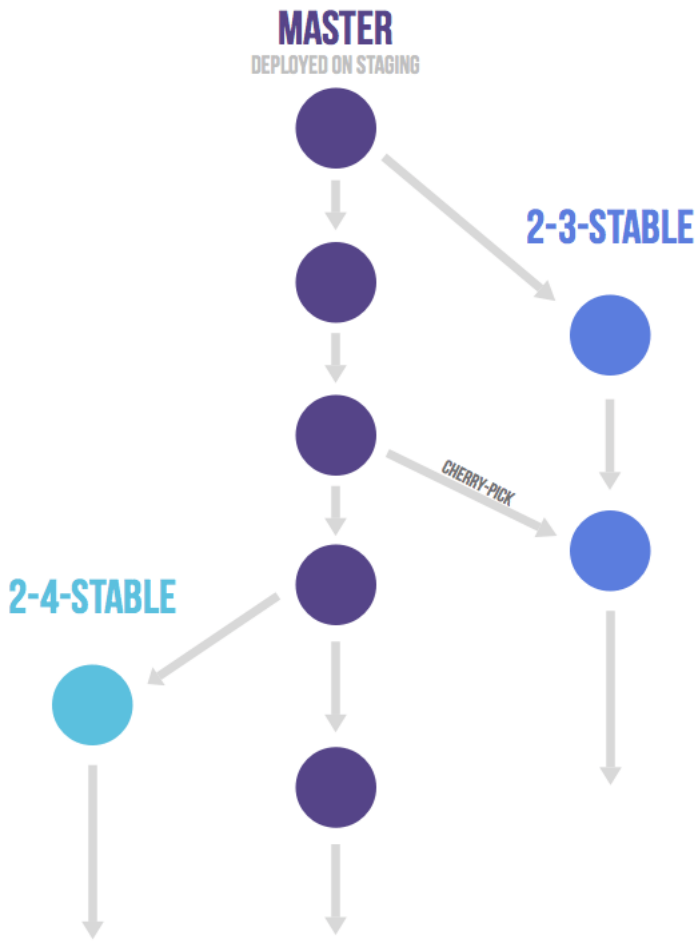

Figure 9.4: Release branches with GitLab flow. Only in case you need to release software to the outside world you need to work with release branches. After a release branch is announced, only serious bug fixes are included in the release branch. If possible these bug fixes are first merged into master and then cherry-picked into the release branch. This way you can't forget to cherry-pick them into master and encounter the same bug on subsequent releases. This is called an 'upstream first' policy that is also practiced by Google and Red Hat. Every time a bug-fix is included in a release branch the patch version is raised (to comply with Semantic Versioning) by setting a new tag. In this flow it is not common to have a production branch.

OneFlow

- You have only one eternal branch in your repository (for example *master*). And then you have feature/hotfix branches, and release branches (all these are being

removed after their usage and integration into master)[16].

- The only condition that needs to be satisfied is that every new production release is based on the previous release. The most difference between OneFlow and GitFlow (above) is, that there is **no develop branch**. Actually, OneFlow was meant to be a replacement for GitFlow.
- **Pros**
  - Clean and more readable git history.
  - Flexible according to team decisions.
  - Ideal when it is needed to have a single version in production.
- **Cons**
  - f your project has a high degree of automation - uses Continuous Delivery, or even Continuous Deployment, for example, then this workflow will most likely be too heavy for you. Perhaps parts of it might still be useful, but other elements (like the release process, for instance) would have to be heavily modified to make sense when releasing on such a very frequent cadence.
  - **Not recommended when it needs to maintain a multiple incompatible release versions** (such as Python2 and Python 3 versions of a project).

[16] https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow

10 Software Licences [SK]

10.1 General

Pri tejto knihe som pouzival zdroje[1][2][3].

V US ak licencia nie je specifikovana, uzivatelia nemozu stahovat, modifikovat a ani distribuovat.

10.2 MIT Licence

Je jednoducha a zhovievava, mozeme robit co chceme pokial tam vlozime povodny copyright a licenciu.

- Povolenia: komercne pouzitie, distribucia, modifikacia, privatne pouzitie, sublicencia.
- Podmienky: include licenciu a copyright.
- Limitacie: zodpovednost.

10.3 Apache Licence, v2.0

Rovnake ako MIT + uzivatelom sa poskytuje patentove pravo expresne.

- Povolenia: komerne pouzitie, distribucia, modifikacia, pouzitie patentu a pre privatne pouzitie, sublicencia a zaruka.
- Podmienky: include licenciu, copyright a zmeny stavu. Ak sa nachadza NOTICE subor, tak ho musime tiez vlozit. Mozeme donho pisat (append).
- Limitacie: pouzitie trademarku (ochrannej znamky - nemusime pouzit meno povodnych autorov pre schvalenie derivovaneho produktu) a zodpovednost.

[1] Licencie - http://choosealicense.com/
[2] OpenSource licencie - https://opensource.org/licenses
[3] Short versions - https://tldrlegal.com

10.4 GNU AGPLv3

Affero General Public License.

- Povolenia: komerne pouzitie, distribucia, modifikacia, pouzitie patentu a pre privatne pouzitie.
- Podmienky: zdrojak musi byt dostupny pri jeho distribuovani, include licenciu a copyright, sietove pouzitie a distribucia, rovnaka licencia (pri modifikacii a nasledom release), zmeny stavu (indikovat zmeny v kode).
- Limitacie: zodpovednost (SW je poskytovany bez zaruky a autor/licence owner nenesie ziadnu zodpovednost za pripadne skody).

10.5 GNU GPLv3 a LGLPv3

General a Lesses General Public License. Je to rovnake ako Apache, plus uzivatel, co si SW skopiruje/pouziva musi dodrzovat rovnake podmienky ako su momentalne dane.

- Rovnake povolenia, podmienky aj obmedzenia. Od predchadzajuceho sa lisi v tom, ze nie je potrebna podmienka sietoveho pouzitia a distribucie.

10.6 Mozilla Public License 2.0

- Povolenia: Komerne pouzitie, distribucia, modifikacia, pouzitie patentu a pre privatne pouzitie, zaruka, sublicencia (schopnost udelit/rozsirit licenciu k SW).
- Podmienky: include licenciu a copyright, rovnaka licencia, zdrojak musi byt dostupny pri jeho distribuovani.
- Limitacie: pouzitie trademarku (ochrannej znamky) a zodpovednost.

10.7 The Unlicense

Licencia bez akychkolvek podmienok.

- Povolenia: Komerne pouzitie, distribucia, modifikacia, pouzitie patentu a pre privatne pouzitie.
- Podmienky: None.
- Limitacie: zodpovednost.

10.8 Zlib-Libpng License (Zlib)

Casto pre open source balicky a zlib kniznicu. Velmi kratka a zhovievava. Modifikovanu verziu SW je treba premenovat.

- Povolenia: Komerne pouzitie, distribucia a modifikacia.
- Podmienky: include copyright, premenovanie (zmenenu verziu je nutne premenovat aby jej nazov nekolidoval s povodnou verziou).
- Limitacie: zodpovednost.

10.9 BSD 2-Clause License (FreeBSD/Simplified)

Skoro bezlimitna volnost, no je nutne skopirovat BSD copyright.

- Povolenia: Komercne pouzitie, distribucia, modifikacia, zaruka.
- Podmienky: include licenciu a copyright.
- Limitacie: zodpovednost

10.10 BSD 3-Clause License (Revised)

Ako predchadzajuca, ale dalsia limitacia - pouzitie trademarku.

10.11 EULA

Asi nie konkretna licencia, vacsinou pre krabicovy SW, je to licencna zmluva koncoveho uzivatela. Zobrazuje, ako moze a nemoze byt SW pouzity. Ma tiez nejake obmedzenia, ako napriklad zdielanie SW s niekym inym.

11 JetBrains IDE (PyCharm) [SK]

11.1 Keyboard Shortcuts

- CTRL+N - najdenie a otvorenie nejakej triedy
- CTRL+SPACE - autocomplete
- ALT+SHIFT+7 - find usages
- CTRL+Q - kratka dokumentacia
- CTRL+B alebo CTRL+Klik - navigacia k deklaracii niecoho
- CTRL+F12 - navigacia v aktualne editovanom subore
- CTRL+D - duplikacia bloku
- CTRL+SHIFT+I - ukaze rychlu definiciu
- CTRL+SHIFT+B - ide do deklaracii typu
- CTRL+SHIFT+J - spoji 2 riadky
- CTRL+P - zoznam validnych parametrov v zatvorke v metode
- CTRL+SHIFT+BACKSPACE - da kurzor na poslednu zmenu, da sa pouzit viac krat
- CTRL+SHIFT+F7 - zvyrazni pouzitie nejakej premennej, pre pohyb medzi nimi F3/Shift+F3
- CTRL+E - zoznam naposledy pouzitych suborov
- ALT+sipky - rychle skakanie cez metody v subore
- CTRL+O - prepisanie metod
- CTRL+ALT+SHIFT+N - otvorenie nejakej konkretnej metody
- ALT+SHIFT+C - prehlad poslednych zmien
- Drzanie tlacitka mysi + ALT - oznacenie stlpcov
- ALT+SHIFT+F10 - Run/Debug dropdown

- CTRL+SHIFT+I - popup okno, pozriet obrazok na mieste kde ukazuje mys
- ALT+ENTER - zoznam akcii, pre refaktorizaciu
- CTRL+SLASH - comment/uncomment riadky
- ALT+9 - zoznam zmien atd, z GITu
- CTRL+ALT+T - kod obalit try/exceptom
- ALT+ENTER - rychlo odstranit problem
- CTRL+TAB - skakanie cez vsetky otvorene subory v editore
- CTRL+K- Commit changes
- CTRL+F a CTRL+R - search / replace
- Ctrl+Alt+T - obalenie kodu niecim...
- Ctrl+Enter - Chytre rozdelenie riadku na 2
- Ctrl+U - chod o nadradenu triedu vyssie
- Shift+Shift - najde hocico
- CTRL+SHIFT+K - pushovanie
- Ctrl+Delete / Ctrl+Backspace - vymazat po koniec / zaciatok slova
- Ctrl+R - replace
- Shift+F10 / F9 - Run / Debug
- Alt+left/right - Prepinanie medzi oknami
- Oznacenie RegExpu, Alt+Enter - do policka zadat string co by sa mal matchnut - vyhodnoti to

12 References

1. Cracking the Coding Interview (6th edition) | Gayle Laakmann McDowell (from 2015)

   https://www.amazon.co.uk/Cracking-Coding-Interview-6th-Programming/dp/0984782850

2. Robert C. Martin series[1]
   - Clean Code: A Handbook of Agile Software Craftsmanship (from 2009)
   - The Clean Coder: A Code of Conduct for Professional Programmers (from 2011)
   - Clean Architecture: A Craftsman's Guide to Software Structure and Design (from 2016)

3. Migrating to Microservice Databases - From Relational Monolith to Distributed Data | Edson Yanaga (from 2017)

   https://developers.redhat.com/books/migrating-microservice-databases-relational-monolith-distributed-data/

4. Microservices in Production - Standard Principles and Requirements | Susan J. Fowler (from 2017)

   https://www.oreilly.com/library/view/microservices-in-production/9781492042846/

5. Software Architecture Patterns | Mark Richards (from 2015)

   https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/

6. Microservices vs. Service-Oriented Architecture | Mark Richards (from 2016)

   https://www.openshift.com/microservices-ebook/

7. 97 Things Every Programmer Should Know | Kevlin Henney (from 2010)

   https://www.oreilly.com/library/view/97-things-every/9780596809515/.

8. AWS Fundamentals: Going Cloud-Native | Coursera (from 2018 by Amazon Web Services)

   https://www.coursera.org/learn/aws-fundamentals-going-cloud-native

[1] https://www.amazon.co.uk/Robert-C-Martin/e/B000APG87E

9. AWS Fundamentals: Building Serverless Applications | Coursera (from 2019 by Amazon Web Services)

   https://www.coursera.org/learn/aws-fundamentals-building-serverless-applications

10. The Data Engineering Cookbook | Andreas Kretz (version 3, 2019)

    https://github.com/andkret/Cookbook

11. Soft Skills: The software developer's life manual | John Sonmez (1st edition, 2014)

    https://www.amazon.com/Soft-Skills-software-developers-manual/dp/1617292397

12. What You Need to Know about Docker | Scott Gallagher (2016)