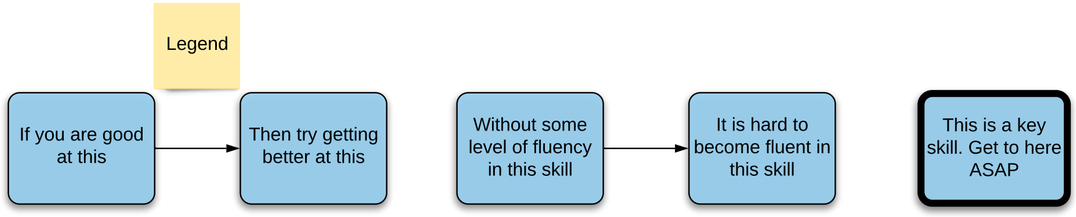
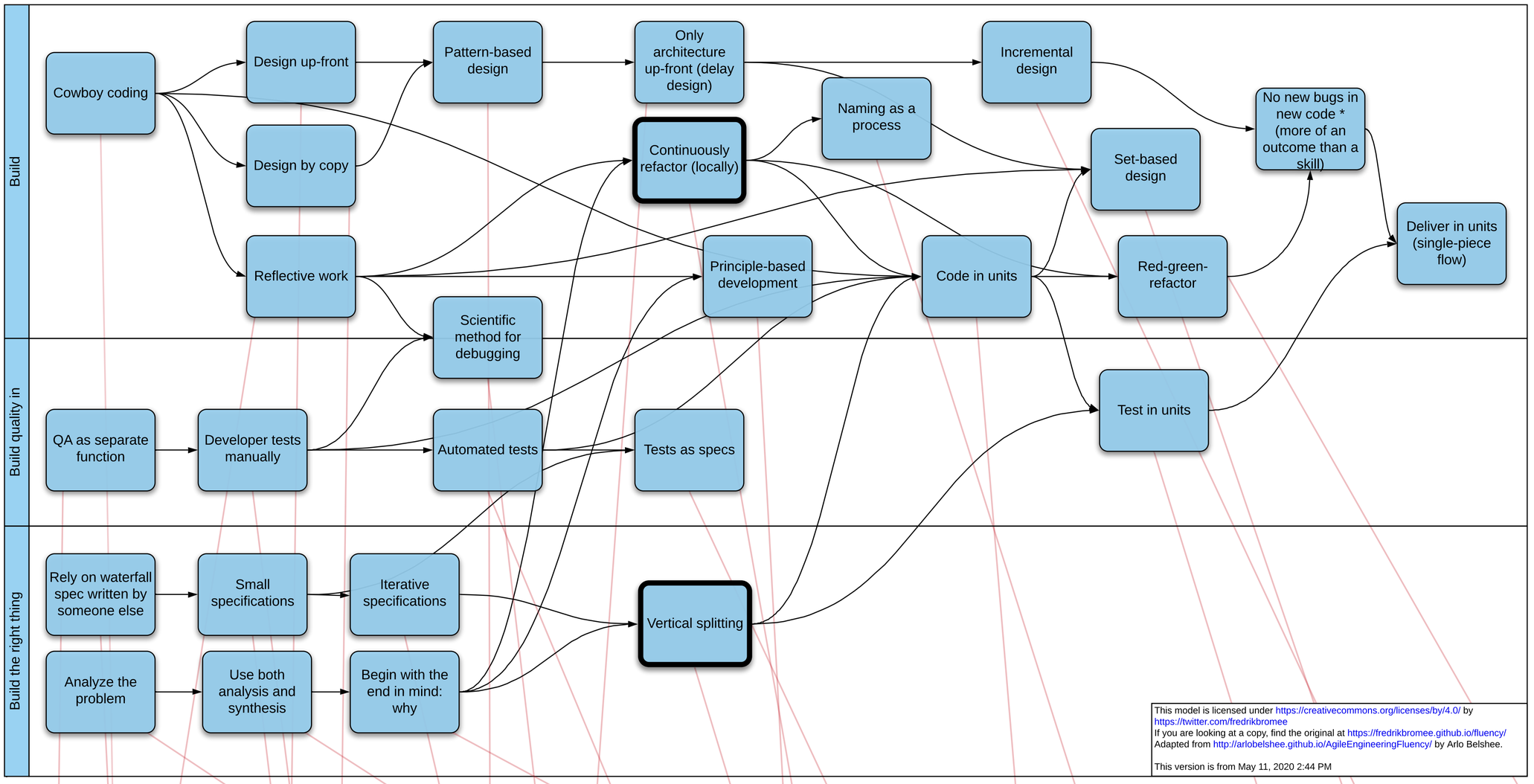
Skills and activities for effective developers

What makes a developer effective? Here is one definition: an effective developer builds
1. the right thing (whatever right is, in *your* context)
2. at the right quality
3. in a maintainable way so that what you build can continue to be changed by you and other developers
All three aspects are needed, none is useful on its own without the others.

This model presents skills and suggested paths for sequence of gaining fluency in these skills, across all three aspects.





Cowboy coding

Sometimes, quickly iterating and pushing a product into production, even if somewhat careless, can be beneficial because the alternative may not be finishing anything at all.

Design up-front

Detailed design down to class-level interaction BEFORE coding. While this is certainly better than hacking away without a plan this has a number of drawbacks:
* it reduces your willingness to embrace evolving reuirements
* it increases the risk of making poor decisions (since coding will give you new information on the problem)

Pattern-based design

There is a large body of work of software design patterns that are proven to work in certain situations. Learn to apply these patterns but beware of added complexity. Move quickly on to continuous refactoring

Continuously refactor

There are a set of code transformations that can be guaranteed to change code without altering its behaviors. They have names. Tools perform these transformations with guaranteed safety.

Learn them by heart so you don't have to think so much to perform them and combine with Reflective work to judge progress.

http://arlobelshee.com/the-core-6-refactorings/

Naming as a process

The skill of naming things is both valuable and possible to learn. A common mistake is to think that it is possible or even beneficial to get it right the first time. Read Arlo's step-based approach and internalize it:

https://www.digdeeproots.com/articles/on/naming-as-a-process/

Incremental design

Trust that your design might change, as you learn more about the problem at hand, and solve related problems. It is natural and expected for a design to evolve as you learn more about the problem, or as the solution is applied to more situations, or if the situation changes. Look at smells in the code as they appear.

Red-green refactor

"Getting software to work is only half the job"

This skill means disconnecting 'get it to work' and 'make the code as good as it can be' in two steps that are tied together in a RAPID loop.

This article is a little bit religious but still good
https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html

Reflective work

This is a key skill needed to improve as a developer. Continuously evaluate the work you do: take a step back and ask yourself or your pair buddy before a commit: is this **good?** Also try to explain why it is good or not. What is the smallest thing that could make it better?

This will allow you to develop taste, and to become aware of what is good in one context might not be good in another.
https://visakanv.com/taste/

Design by copy

Using a similar feature as a blueprint for the design of the current feature. This is great for productivity since it reduces a lot of uncertainty but you might lose opportunities for simplifications if you don't look for similarities and act on them.

Only architecture up-front

By delaying design decisions you give yourself more time to gather context and a better chance of making good design decisions

Principle-based development

As you get better at building, try to objectively judge your work. Why is this design good in this case? Can I apply the same choices in another situation?

This will result in principles, and trade-offs that you are willing to make in certain contexts.

Code in units

Take smaller steps, but take them the whole way to completion.

Set-based design

The ability to delay design decisions unlock a developer super power: set-based design, or sometimes called set-based concurrent engineering.

If you do manage to keep your current work small in scope and yet vertical and value-adding you can and should evaluate more than one solution at a time.

Using this practice drills in being disconnected from any specific solution and gives training in evaluating choices based on merit. Do allow your set of choices to evolve and influence each other.

QA as separate function

A dedicated person or team who's job it is to find defects before the software

Developer tests manually

The developer manually verifies that the code works as expected

Scientific method for debugging

Debugging is a skill that can be practiced as any other skill. Practice using the scientific method: formulate hypothesis, falsify, update hypothesis and repeat. Don't just "change things until it works".

Also very important for debugging: know your tools, practice using them.

Automated tests

Automating tests allow for repeatability and faster feedback loops

Tests as specs

Automated tests should be treated as equal to business code and the same care should be taken to make the test code clear understandable and maintainable.

Good tests act as specifications that are automatically validated, but also serve as documentation and education.

Test in units

Take smaller steps, more often. Strive for writing tests that only test the unit of work that you are working with.

Begin with the end in mind: why

Always begin with why you are doing something. Knowing why something is done allows you to value it and value allows for prioritization and selection.

What part of the problem should I solve first? Which of these two solutions is better one?

Read Sinek or Covey's amazingly corny and at the same time useful book from the 80's:
https://en.wikipedia.org/wiki/The_7_Habits_of_Highly_Effective_People

Rely on spec written by someone else

In waterfall fashion, a spec is written by somebody else and frozen before project start.

Small specifications

Write specifications that are smaller in scope and can be delivered as separate pieces

Iterative specifications

Write specifications that are iteratively and incrementally updated as you learn more about the problem and the solution

Vertical splitting

This is a key skill to get better at since this is what allows the team to make **value** explicit in planning, and in conversations. Having a good grasp of how something gives value, and how splitting it in smaller pieces that can be delivered separately makes the team more predictable in delivering value.

Don't only rely on stake holders to provide a definition of value - this is part of your job. What a customer
* asks for
* wants
* needs
are often three different things. It is your job to understand the problem so these three things become the same.

https://agileforall.com/wp-content/uploads/2018/02/Story-Splitting-Flowchart.pdf
https://qualibrate.com/blog/wp-content/uploads/2018/09/Elephant_Carpaccio_exercise.pdf

Analyze the problem

Learn to use analysis to understand how a problem is solved today. The three steps of analysis are:
1. Break down thing into parts
2. Identify the properties and behavior of each part
3. Aggregate the understanding of the parts into a whole

Use both analysis and synthesis

The problem with analysis is that it can only produce knowledge, HOW things work - not WHY things work the way they do. For that you need synthesis, which also has three steps:
1. Find out which system the thing is a part of, a containing whole. Car -> transportation system, university -> educational system
2. Understand the behavior of the containing whole. What does the transportation system do? Transportation system moves people. How did people move before the car?
3.Dis-aggregate the understanding of the containing whole, by identifying the role and function of the thing in the containing whole

Learn to use both, and to switch between them. Why takes you up, how takes you down into details.

Learn system thinking if you want to get better at this. You can't go wrong listening to Russel Ackoff https://youtu.be/OqEeIG8aPPk