TASK

Thinking Like a Software Engineer II - Diving into a Large Codebase and Writing Maintainable Code

Visit our website

Introduction

Welcome to the Diving into a Large Codebase and Writing Maintainable Code Task!

More often than not, a software engineer is required to work with an already established codebase. For someone who has not worked with large, already existing codebases, facing thousands of lines of code can be a daunting task. This task will explain how to tackle large codebases as well as how to write maintainable code.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact your mentor to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with your mentor. You can also schedule a call or get support via email.

Your mentor is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

What is a Codebase?

So, what exactly is a code base? Simply put a codebase refers to a whole collection of source code that is used to build a particular software system, application or software component. Only human-written source code files are included in a codebase, so therefore source code files that are generated by tools and binary files are not included. Codebases also generally includes configuration and property file as, they are necessary for the build.

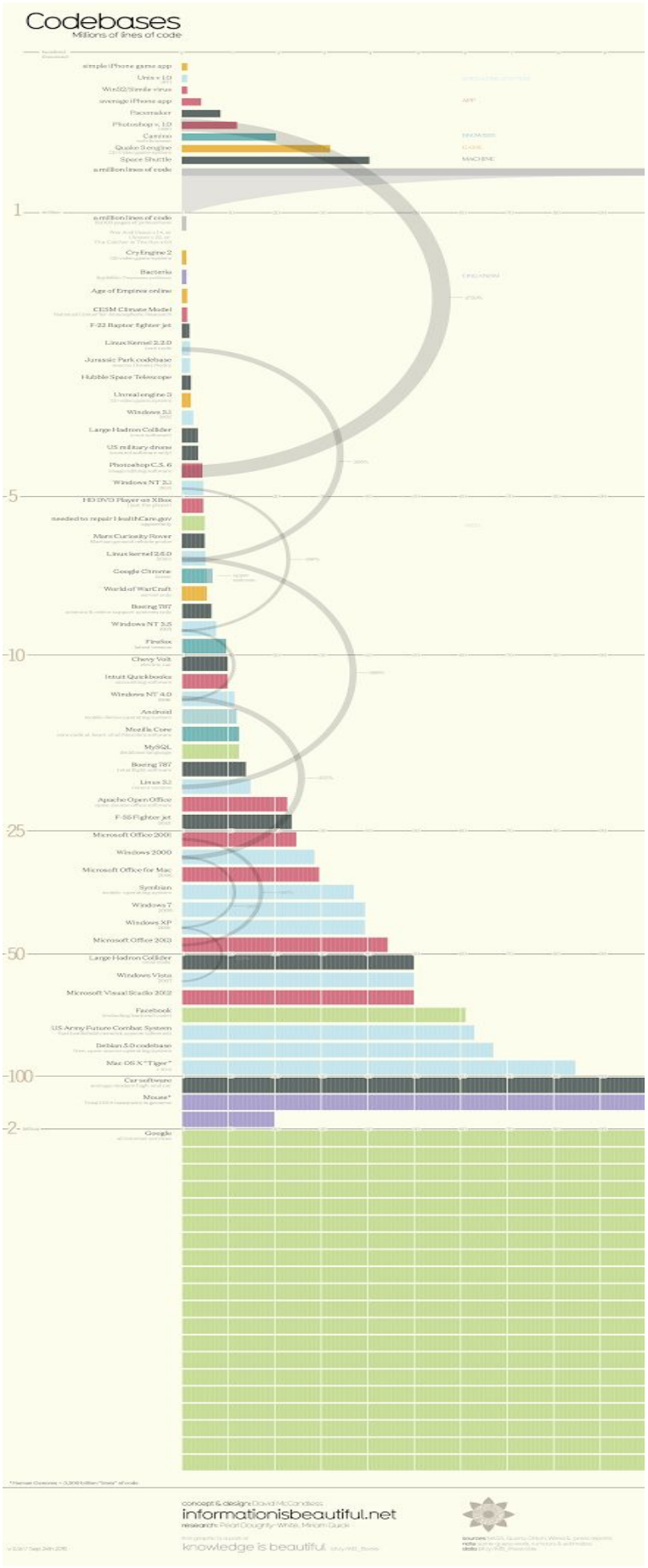
A codebase is normally stored in a source code repository that belongs to a version control system. A source code repository is a place where large amount of source

code are kept, either publicly or privately. They are typically used for backups and versioning, and for projects with multiple developers to handle the various different source code versions and to aid in resolving conflicts when different developers submit overlapping modifications.

A note from the Hyperion Team

Have you ever wondered how many lines of code there are in the Google codebase? Believe it or not there are actually 2 billion lines of code in the Google codebase and all of them fit into a single code repository!

Below is an infographic showing the biggest codebases in history, as measured by lines of code.



Working With Large Codebases

Diving into a large unknown codebase can be intimidating especially if you have never worked with large codebases before. As a software engineer starting out in a new company, you will often need to jump straight into the codebase and become productive as quickly as possible. Fortunately, there are a number of tips that can help you tackle a large codebase and feel less overwhelmed.

1. **Try to understand the purpose of the application:**
   Before even looking at the source code it is important to understand the purpose of the application. By doing this you have a better context for what different objects in the code represent. Understanding the business domain of the application provides context for why features exists and why code is written in certain ways.

2. **Use the application:**
   It is always a good idea to use the actual production application as an end-user. By doing so you are able to get a high-level overview of the application. Have someone who is well versed with the application give you a tour. Someone who knows the features well will be able to give you a lot of insight.

3. **Do a cursory review of the code:**
   Don't worry about how the details at this point, simply focus on the broad picture. Pay attention to how the code is organised.

4. **Talk to the other developers working on the app:**
   It is always beneficial to talk to other people who have worked on the app. They will be able to summarise what the application does and how major features work. They can also warn you about parts of the code that are poorly written and that you should avoid. Most importantly, remember to ask questions if get stuck!

5. **Look at tests:**
   Tests give you a better understanding of what edge cases the application might be handling, or what business logic is important. Integration tests can also tell you what the most important user workflows are.

6. **Review the application environment:**
   At this point, when you are still trying to get comfortable with the code, you do not need to know every detail about how to push to production. However, you do need to understand how to get it running on your development machine.

7. **Start coding:**
   Don't waste time by spending days reading code trying to understand every single component of the application right away. Instead just start diving in with some small bugs or features. This helps you focus on a small section of the application, without worrying about how everything else fits together.

8. **Add documentation:**
   It is essential that you add documentation to make it easier for the next person that might work on the application. You can also start a glossary to keep track of what things mean, either just for yourself to use or, to share with other people.

How to Write Maintainable Code

Before we can change any code, we have to understand what the code currently does, and whether it's supposed to do that. Changing code is difficult if we have no idea how it works.

You will quickly realise that when working with a codebase, one of the biggest problems you face is dealing with unmaintainable code. Someone else's messy code can significantly slow you down and hamper your productivity.

When a developer writes code, they often imagine that they will be the only one working on it in the future but, in reality this is far from the truth. More often than not, numerous developers work on the same codebase throughout the lifecycle of the application. New functionality may be required, changes will be needed for existing features, as well as bug fixes. Different developers will need to be able to make these changes without breaking the existing code. It is therefore extremely important that your code is maintainable.

By following the simple rules below you should be well on your way to writing clean, maintainable code.

1. **Write Code That is Easy to Understand and Debug**

   You code is useless to the next developer to work on it if they can't understand what you have written. If another developer can't understand what you have done, they will probably just throw that code away and start over from scratch since it takes longer to try to understand poorly written code than to write new code. Below is a few best practices.

   Optimise for the reader, not the writer: It is easy to save time writing code by taking shortcuts but it can cause serious frustration and confusion for anyone reading that code later on.

   Include comments: It is not always obvious what certain code does at first glance. It is therefore very important to include good comments to explain what code does. Don't write comments for yourself but rather for someone else who might try to read and understand your code.

   Do the smallest, simple thing to add value: Always focus on one task at a time and concentrate on writing the best code you can for the current task. Designing your
   code in a modular fashion with separate parts is much easier to understand.

   Keep It Simple Stupid: You should never assume that the next person to work on your code has the same level of experience as yourself. You should assume that the reader is a novice and keep you code as simple and straightforward as possible. Leave out any experimentation or excessive optimisation.

   Ensure good logging of code execution: In order for effective debugging, good code logging is required. You need evidence of what was going on when the code was written. When you log from the beginning, it will be easier to pinpoint specific errors and the origins of those errors down the line.

2. **Write Code That is Easy to Modify**

   It must be easy to add functionality and features to code in order for it to be truly maintainable. Making a single change should not break the code in ten different places. Below are a few tips that make you code easier to change in the future.

   Don't repeat yourself: Don't copy and paste code, rather extract the common functionality to be available throughout your codebase.

Separate concerns: You should always try to separate code into modules that each do one distinct thing. There should be as little overlap as possible between modules. Doing everything in a single module makes it difficult to make changes without breaking something else.

Separate code and data: You should always externalise text into separate files.

Avoid using long statements and deep nesting: Don't try writing all your code in one big function since it can be difficult to understand.

3. **Write Code That is Easy to Test**

A good set of tests can serve as documentation, indicating how the code is supposed to behave while making sure that the code actually supports the expected behavior. They also give you confidence that your code still works after you've made changes to it. We will discuss how to write tests later on in this course.

Compulsory Task

Follow these steps:

- Look the poorly written, difficult to read program below:

```
import java.util.Scanner;

public class WhatDoesThisAppDo {
public static void main(String[] args) {
final int NUMBER = 5;
int int2 = 0; int int1 = 0; long startTime =
System.currentTimeMillis();
String string1 = " "; Scanner string2 = new
Scanner(System.in);

while (int1 < NUMBER) {
int number1 = (int)(Math.random() * 10);
int number2 = (int)(Math.random() * 10);
if (number1 < number2) {
```

```
int temp = number1; number1 = number2; number2 = temp;
}
System.out.print(
"What is " + number1 + " - " + number2 + "? ");
int answer = string2.nextInt();
if (number1 - number2 == answer) { System.out.println("You
are correct!"); int2++; // Increase the correct answer count
}
else
System.out.println("Your answer is wrong.\n" + number1
+ " - " + number2 + " should be " + (number1 - number2));
int1++;
string1 += "\n" + number1 + "-" + number2 + "=" + answer +
((number1 - number2 == answer) ? " correct" : " wrong");
}
long endTime = System.currentTimeMillis();
long testTime = endTime - startTime;
System.out.println("Correct count is " + int2 +
"\nTest time is " + testTime / 1000 + " seconds\n" +
string1);
}
}
```

- Firstly, without running the program, determine what the the above code does. Write your answer in a text file called answers.txt.

- What is the output of this programme? (Provide your own input data if necessary) Write your answer in the answers.txt file.

- Rewrite this programme so that it is easier to read. Make sure your code has:
    - descriptive variable names
    - descriptive class names
    - at least 3 comments
    - consistent indentation
    - code Grouping

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.