**1:**

After doing some testing I noticed that **crawl()** runs only once, whereas **parse()** runs every time a page is parsed. Then its individual *durations* get added together as follows:

**data.compute(key, (k, v) -> (v == null) ? elapsed : v.plus(elapsed));**

Since the parallel crawler can handle multiple tasks in parallel during a given time interval, more total durations get added together. Meanwhile, when the sequential crawler is running, only one page can be parsed at a given time, so less total parse time is added together and recorded.

**2a:**

I would explain that the boss’s old PC probably doesn’t have more than one CPU core and can’t take advantage of ForkJoinPool’s ability to execute tasks in parallel. By setting the parallelism level to one, we mimic a machine with just a single CPU core. However, there’s a certain amount of overhead involved when creating subtasks, joining tasks, etc., which makes ForkJoinPool less efficient, at least task-for-task. That’s why the sequential crawler performs better when there’s only one CPU core.

**2b:**

If the boss were to switch to a computer with multiple processor cores, the parallel crawler would crawl more pages than the sequential crawler. My own laptop has 4, which I verified using **System.out.println(Runtime.getRuntime.availableProcessors())**. I changed my parallelism level to 4 and the parallel crawler performed better.

**3a**

The cross-cutting concern addressed by the performance profiler is **performance monitoring**.

**3b**

If a *join point* can be described as an *event* in AOP lingo, then our events are the **@Profiled** method invocations. These methods are events that take place during the execution of our program to which we can then apply a *profiling aspect*, which in this case is the performance profiler itself.

4.

**Builder**: I first noticed the builder pattern in the CrawlerConfiguration file. Later on, I used it myself when implementing parallel crawler. More specifically, I used it to avoid long constructor calls when adding new subtasks of the CrawlInternal class. I like the builder pattern because you can create a very complex object with a line of code as simple as CrawlInternal.Builder.build() (in theory, that is). Depending on how it’s implemented, that one line of code could create an object with a huge number of instance variables without having to explicitly give their values. With the builder pattern, it’s also harder to make mistakes when more than one variable is of the same type. In my case, I didn’t have that problem in the parallel crawler (everything is of a different type in the constructor), so perhaps my implementation didn’t make tons of sense, but I implemented it for educational purposes nonetheless. The reason I didn’t like builder was because it added many more lines of code, which created clutter and could potentially affect readability in the future. The “inner class” creates a confusing picture at first.

**Dynamic proxy:**

The dynamic proxy pattern is used in the implementation of the performance profiler. The profiler consists first and foremost by the profiler interface, which relies the Guice dependency injection framework, and the invocation handler interface (ProfilingMethodInterceptor). Dependency injection and dynamic proxy are interwoven here (the profiler module is a good example of that). I will discuss dependency injection separately below. Regarding dynamic proxy, we have method calls “intercepted” by a proxy instance, which then adds additional behavior to the methods, defined in the invocation handler. In our case, we record the time it takes to complete those methods for performance-profiling purposes. I like dynamic proxy because it allows us to gather information about the performance and use of our code without having to make changes to the code we are monitoring. To test the speed of a method, you don’t actually need to add anything to the original code block. Instead, you add it to a handler and your original code is left undisturbed. I can’t say that I have a strong dislike for dynamic proxies, but I guess I could complain about the boilerplate code in the invocation handler, which seems like it should be simpler.

**Dependency Injection:**

Dependency injection is sprinkled throughout the app. When a class depends on another class (client and service), it’s possible to simply hardcode that dependency. However, what if there is some kind of need to switch out the dependency? Instead, for each service, we can simply have multiple implementations of the same service interface, which promotes reusability. That is how I would explain dependency injection in a few sentences. In our code, we have two such implementations of a WebCrawler interface. The client is WebCrawlerMain and the implemented services are the ParallelWebCrawler and the SequentialWebCrawler, which we can exchange for one another with a little bit of help from the Guice framework. I like dependency injection because it allows me to reuse code such as WebCrawlerMain without having to write two different versions, one for each dependency: I can have just one unified WebCrawler dependency. I guess I don’t like dependency injection because, at least at first, it was abstract. The idea that the different services would implement one interface wasn’t very concrete to me, but I am getting more comfortable with it over time.