Dr. Software

An unfinished journey starting from dirty code

By **Dan Nicolici**

The Universe is way too big for us to understand, but that doesn't mean we shouldn't try.

Dan Nicolici
Cluj-Napoca, Cluj, Romania
May 2021

Prologue

Why this book? I think it's mostly because I want to have that "once and for all" feeling about this seemingly never-ending subject (in my professional life): how to evolve existing, low quality software, to serve the business properly.

This book is about taking a real (well... invented, but the code is as real as you'd find it in the industry) software system, that needs to grow in functionality, but needs to do so in the context of an ongoing business, without disrupting it too much and adding new features with a reasonable development cost.

The subtitle says it's an unfinished journey, because the future never ends (at least as long as we move through this spacetime), not because our goal of evolving the system according to requirements isn't met. And besides, it leaves room for improvement and responds to statements like: "yes, but you could have...".

Enjoy!

Chapter 1 - Software systems

We developed tools, over the course of our evolution, to help us gain a significant advantage over other species and to make our lives better.

Our brain's capacity to create sophisticated technology is what brought us to where we are today. We developed an unprecedented way of living, here on this planet and, in the last few decades, we changed it so drastically and so fast, that we can barely understand the impact it has on our environment and on ourselves.

The advanced technology we use today is mostly created and run by software. Farming tools, communication devices, transportation, scientific research, weapons, etc., they all have software controlling most stages in their processes. So the purpose of software, as the tool that it is, is to operate other tools, which in turn will serve us in solving specific domain problems.

Software development became such a big deal in our society, that ever more people are doing it and want to do it. Unfortunately, a lot of software developers (and dare I say the majority) make software development a purpose in itself, not giving the actual domain problem too much thought (or not at all in many cases). This doesn't make any sense. How can you know your tool is actually helping?
Some might argue that software can be broken down into components, which can be independently built and then integrated at the end, somewhat similar to a car factory. I understand the need for this mindset: it makes reasoning about things, easier. However, because of the "soft" nature of software, it is orders of magnitude easier to customise and change it than it is for car parts. This is actually the reality of software development: constant change. Still, it seems like most of the time, we build software systems that are difficult to change and maintain. Attempts have been made, to change this (e.g., XP, agile, lean, etc.), but somehow we took those ideas and turned them on their head until we ended up where we started, software difficult to change and maintain.
It's difficult to see software as merely a tool, when there are countless hours being spent just to learn a particular programming language or operating system. There are egos at stake, personal targets, preferences, social aspects basically. I don't doubt similar things happen in other industries as well, but the nature of software integrates better with our capabilities. This has been observed quite some time ago already (see Conway's law).

The state of software today

The section title is a bit misleading, because this is certainly going to be an incomplete overview of the software landscape, because it comes only from my personal experience. Obviously, there are vast areas that I haven't touched and areas that I've never even heard of. Nevertheless, I have worked with tens of really big systems and hundreds of smaller ones. I've met thousands of people in the industry and worked directly with less than that. I've had a lot of roles, throughout my professional years, but since I was a kid, I never stopped coding. So please take this with a grain of salt.

All projects that I got into had nasty, unnecessary issues. Almost all of them were started in a rush, to get the business off the ground and had created a parallel business of supporting the customers through all the defects they had. It's almost like the businesses were caught in a startup limbo, for years. Sometimes decades! Working in such a company can take its toll on someone. Most defects in software come from poor engineering. Oh and yes, design, architecture and other fancy words, are just engineering in the software industry, because it is not mature enough to have that level of abstraction, in which the architect draws the picture and the engineer implements it in such a way that it almost 100% of the time is right on the money. No. Far from it. What happens in reality is that the engineers try desperately to make the system work somehow, while maintaining the illusion of the story told by the architect. Also, the architects are so detached from the engineers, that they simply cannot adjust the architecture based on the actual feedback from building the system. So, we end

up with long running, half baked systems, that are being heroically dragged forward by engineers. Wouldn't that be a nice winning conclusion? I bet it would!

How did we get here?

Engineers, actually, let's just call ourselves programmers, shall we? Where were we? Oh, yes... programmers are very much to blame for the state of the software. Even if you don't receive a good design, you should ask the right questions to clarify what's missing and implement it with those missing variables in mind until you have the answers. Even more than that, you can (and should!) challenge things, but make sure you are properly prepared to do so.

A good programmer understands the problem space and the tools at hand for solving that problem. Unfortunately (yes, I'm going to come out and say it), the *majority* of programmers today, are not good programers. Is it their fault? Yes and no. Writing software is quite easy: you just need a computer (or tablet, or phone, whatever...) and a connection to the internet, right? You have a problem to solve, you navigate to certain Q&A, voting-based websites and you get your solution right there, voted by the *majority* of programmers. You see the problem? This approach, almost always leads to a local fix, but generates many other defects in the process through the butterfly effect.

How do we fix it?

It is fixable with a methodical approach. The journey starts with you dropping into the middle of an ongoing process for evolving an existing system. What you usually do, is

take an immediate requirement (e.g., a ticket, a user story, etc.) and try to make sense of the mini-universe around it: business case, infrastructure, tools, code, delivery process. In an ideal situation, everything is obvious and the change you need to do is easy to perform. However, reality is different. You will encounter mostly drafts in every step of the way, starting from specs, to poorly running infrastructure, to flaky tests and unclear code.

In all these cases, the approach I take is this:

1. Run the product (on my dev machine if possible)
2. Run all the automated tests in the project on my dev machine (the ones that can run locally), if any
3. Understand the requirement through practical interaction with the local product (find someone that knows what they need from the system, usually somebody responsible with the product if the ticket is unclear)
4. Understand the area around the code that will be impacted by the requirement
5. Write new shiny code (writing tests first)
6. Link the newly written code with existing code
7. Deliver

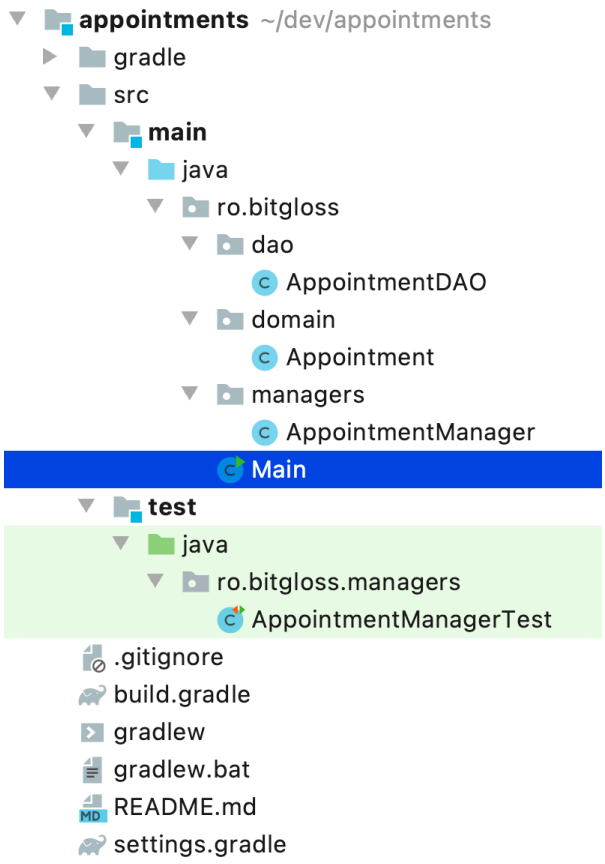
Sure, this list is nice and (somewhat) short, but no doubt you've been through a rough time yourself while attempting one or more of the steps. Don't worry, we'll go through a practical example in the next chapter.

Chapter 2 - Practical example

In this chapter, we'll do a hands-on exercise. We're faced with an appointments system that needs to grow. We're contacted by a local medical practice that has an old and pretty expensive to maintain system that is hard to use due to its infrastructure. Okay. We're a bit scared of those statements, given that the business case seems kind of trivial: appointments. "So what can you already do with this system?", we ask. "Well, we can create new appointments, by entering the date, the doctor and patient names. We can also list all the appointments we have." Okay, now it's really scary. This is abnormally trivial for the "expensive" and "hard to use" terms thrown above.
Let's have a look!

The appointments software

We open up the source code. By the looks of it, it's a java project. We fire up a well known IDE and open the project from there. Sure enough it loads successfully. We feel proud we could get this far. Now we're confident.



Suddenly we remember those “expensive” and “hard to use” terms. It’s a project with 4 java production classes and a test class. Let’s look at the entry point, the Main class (at least we assume it’s the entry point, by its name). We bring it up in the IDE editor and marvel at its simplicity.

```
public class Main {

  public static void main(String[] args) throws IOException, ParseException {
    AppointmentManager appointmentManager =
                                new AppointmentManager(new AppointmentDAO());
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    while (true) {
      System.out.println("Existing appointments:");
      appointmentManager.printAppointments();

      System.out.println("Input new appointment? (y/n)");
      if (br.readLine().equals("n")) break;
      appointmentManager.inputNewAppointment();
    }
  }

}
```

Hold on! *System.out.println*? What is this? A console application? Holly sh..! Yes... "hard to use" makes more sense now. Moving on, looks like there's a crude UI being looped until the end user doesn't want to enter an appointment anymore (there's a yes/no question and a loop break on the no). Fine. Let's just run it and see what happens. Here's the output:

```
Existing appointments:
-------------------------------------------------------------------------------
|         TIME          |         DOCTOR          |         PATIENT         |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Input new appointment? (y/n)
```

What kind of ASCII art is this? It's waiting for input. No, we don't want to enter anything, so let's hit "n" + Enter. Sure enough, the process stops, as we expected. Let's enter an appointment and see what that look like now. So we fire up the Main class again and this time we hit "y" + Enter to see what we're in for:

```
Enter time:
```

Oh boy! What format does it expect? Let's have a look at the source code. We see that AppointmentManager has method called *inputNewAppointment()* and browsing through that method, we find a date format inside:

```
private static DateFormat DF = new SimpleDateFormat("dd/MM/yyyy");
```

We'll come back to this method, but for now, we have what we were looking for. But... this is just a date. There is no time in this format. Hmm... okay, we remember this as an odd thing and carry on. We now know what to type into the terminal. We make an appointment for the 5th of April 2021. The next prompts are for the doctor and patient's names. This looks straight forward, so we go ahead and type those in. Boom! We got this! High five!

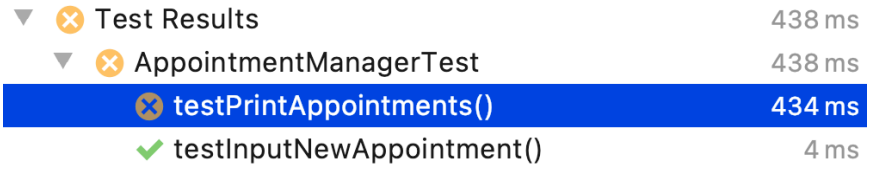
```
Enter time: 05/04/2021
Enter doctor: dr. Smith
Enter patient: John Doe
Existing appointments:
--------------------------------------------------------------------------------
|        TIME        |        DOCTOR        |        PATIENT        |
--------------------------------------------------------------------------------
     05/04/2021                dr. Smith                John Doe
--------------------------------------------------------------------------------
Input new appointment? (y/n)
```

So it shows us what we've got so far and prompts us to enter a new appointment or to quit. Phew! We operated the system. Fantastic!
Time to see what the business wants from this system, right? When consulted, business wants the ability to switch the format the appointments are displayed in. Currently, they show up as a table, but they'd like to see them as lists

too. Also, they would like to be able to add comments when adding a new appointment.
Good! These are pretty straight forward, reasonable requests. I assume business thinks the same and expects the same in terms of development costs.

Setting up the dev flow

Before we start coding anything, we want to make sure we have a good understanding of the local environment we're working in. Running the unit tests is always a good place to start. Let's do just that.



Oh, we have a failed test. Let's see why the test is failing.

```
[-------------------------------------------------------------------------------
| TIME | DOCTOR | PATIENT |
-------------------------------------------------------------------------------
, 02/12/2019, , doctor, , patient,
, EXPIRED , , doctor, , patient,
-------------------------------------------------------------------------------
```

It looks like it expects 02/12/2019, but instead it finds EXPIRED. Bummer... No worries, this happens a lot in real life. Let's examine the failing test's source code.

```
@Test
public void testPrintAppointments() {
  Appointment appointment = new Appointment();
  appointment.setDate(new Date(2019 - 1900, 11, 2));
  appointment.setDoctor("doctor");
  appointment.setPatient("patient");
```

```
  dao.saveAppointment(appointment);
  appointmentManager.printAppointments();
  ArgumentCaptor<Object> argumentCaptor = ArgumentCaptor.forClass(...);
  verify(out, times(14)).print(argumentCaptor.capture());
  List<String> printout = Arrays.asList(
      "-------------------------------------------------------------------------\n"
    , "|        TIME        |        DOCTOR        |        PATIENT        |\n"
    , "-------------------------------------------------------------------------\n"
    , "        ","02/12/2019","        ","        ","doctor","        ","        ","patient","        ","\n"
    , "-------------------------------------------------------------------------\n");
  assertEquals(printout, argumentCaptor.getAllValues());
}
```

(You will notice the invalid Java syntax when creating the argumentCaptor, but that's code intentionally left out to fit the page size and besides, it's not important here).

Let's first comment on the readability of this test. Method name starts with "test" which is redundant, because there already is a @Test annotation. Next, the name of the test could be improved, to let us know what is expected from the print output, but we can live with it for now. The layout of the text is not communicating to us whether it's setting things up or exercising code or verifying outcomes. Let's go top down and figure this out for ourselves.

It starts by creating an appointment object and setting its fields. Oh, here's our date, the one that screwed up the test. Odd looking date, but that's because of the old Java Date API.

Moving on, after the date is all set up, it's being "saved" with the help of a dao object. Usually dao stands for Data Access Object, so no need to look further for now, we just assume it saves it somewhere dark, where data lives.

Next, we finally encounter the actual piece of code that is supposed to be tested, the print stuff. The *appointmentManager* is used to print the appointments that it supposedly retrieves from that data layer we've assumed is lurking around before.

The following statements look like some Mockito magic. They set up a way to capture what's printed to the output stream. What is that output stream anyway? Let's have a look at the overall test setup (@BeforeEach means this method is going to run... you've guessed it, before each test method):

```
@BeforeEach
public void setUp() {
  dao = new AppointmentDAO();
  dao.deleteAllAppointments();
  appointmentManager = new AppointmentManager(dao);
  out = mock(PrintStream.class);
  System.setOut(out);
}
```

Oh, wow! The entire system's output stream is mocked. Ugly! Now we understand that the production code is supposed to print 14 times and to output the exact string that is expected in the test. Ok, the failure does not occur at the line where the number of times the code prints, but man this is brittle test code! The test shouldn't be concerned with how many times print is called, but rather with the final outcome. What if someone decides to produce the same output by concatenating 2 or more strings? The result will be the same, but the test will fail. This is what is meant by a test knowing implementation details of the production code. It's bad! Don't do it!
All right, but what about that EXPIRED we got when running the test? There is more to the story than this test is telling us.
Unfortunately, is it almost always the case with code like this. We'll have to navigate to production code in order to understand the whole story.

Let's have a look at *printAppointments*:

```
public void printAppointments() {
  List<Appointment> appointments = dao.getAllAppointments();
  if (appointments != null) {
    displayLine("-------------------------------------------------------------------------------");
    displayLine("|          TIME          |          DOCTOR          |          PATIENT          |");
    displayLine("-------------------------------------------------------------------------------");
    for (Appointment a : appointments) {
      Date date = a.getDate();
      long time = date != null ? date.getTime() : new Date(1970, 1, 1).getTime();
      // appointments older than 6 months are marked as EXPIRED
      if ((System.currentTimeMillis() - time) / 1000 > 3600 * 24 * 30 * 6) {
        display("          ");
        display("EXPIRED   ");
        display("          ");
      } else {
        display("          ");
        display(DF.format(a.getDate()));
        display("          ");
      }
      display("          ");
      display(a.getDoctor());
      display("          ");
      display("          ");
      display(a.getPatient());
      display("          ");
      displayLine("");
    }
    displayLine("-------------------------------------------------------------------------------");
  } else {
    displayLine("No appointments found");
  }
}
```

Evrika! Appointments older than 6 months are marked as expired. The comment says it clearly. Does the code? Maybe, but it's not clear, hence the comment. And hello 14 print statements as well, because *display* and *displayLine* are nothing but wrappers:

```
private void display(Object o) {
  System.out.print(o);
}

private void displayLine(Object o) {
  display(o + "\n");
}
```

We now know for sure that the ones writing this test never accounted for this business rule, the expired appointments. Do we fix the test? If we have OCD we can fix the test, just to see green overall. But we'll try to refrain ourselves from doing so. Instead, let's focus on the new requirements and build them separately, as much as possible, from the existing system.

Pulling concepts apart

Clearly, the tabular format is hardcoded. Also, there's more going on here than just formatting. There's fetching the data, expiration business logic, printing logic. All these concepts are glued together in a method and thrown in a class named *AppointmentManager* (classes that have Manager in their names, are usually a placeholder for "there are all kinds of actions that happen with X" type of statement, but they are "cleverly" grouped together under one flag: THE MANAGER). Also, there is nothing that would allow us to switch the format right now. Ok, so first let's just pull these concepts apart, in a way that would allow us to compose them and most importantly, plug in different implementations, should we so desire (which will serve our immediate requirements).

Refactor the tabular view

We'd like the display format to be a function of some data source, something like *display(dataSource)* that produces formatted content. It's a very simple idea, but how did I come up with it? For one, in the current implementation, there is no other way of serving the appointments to this

method, but indirectly, through the DAO dependency and only through one single channel: *getAllAppointments*. What we would really like is to not care how we end up with the appointments as long as we have them. Besides, what if we'll need other ways of fetching them? We'll have to open up this method and in the best case scenario, move the fetching logic outside or in the worst case scenario, implement cumbersome fetching logic inside it. So simply extracting the data source as a function parameter saves us, the implementers of the method, from dealing with the dilemma. Then there's the meta-data, which is tightly coupled to the structure of the appointments, the table headers in this case. This is also something we'd like to ignore as method implementers and have it passed in as an argument. Since it's coupled with the actual data, we'll create an abstraction that encompasses both.

Next up is the formatting of whatever is provided by the data source. Since we want different views of the data, it'd be nice to be able to let the view implementation do its own thing (or strategy) while we simply tell it what to do (as opposed to how to do it). The *display* function would then belong to this view concept.

The business logic, which is specific to a property of the model is also clearly polluting the method. At this point, we shouldn't be aware of such details, let alone handle them. Imagine we implement the expiration logic everywhere we need to do something based on it, not necessarily displaying stuff, but maybe also sending mails or other computation. This, by the way, is vastly encountered throughout real production code bases. When the expiration rule changes, we need to change all those places. Horrible! So we'll have to centralise this logic somewhere outside this method.

Let's go ahead and express that in code. We'll create an interface called *DataSource*. If we look again at the *printAppointments* method, we'll notice that it first prints some meta information and only then does it print actual data content. So we want something like this:

```
public interface DataSource {
  List<String> entryDetails();
  Stream<List<String>> stream();
}
```

In the entry details we'll find header information (for the tabular view) and the stream will provide lists with data for each detail. Could we have drafted something more clever, that would couple those list indexes? Maybe... For now, this will do.
Now, on to the formatting stuff. We'd basically like the same data to be viewed from different angles, so let's go ahead and express that in code:

```
public interface View {
  String display(DataSource ds);
}
```

Great! We now stated in code our desire to take some data and view it in a certain way. Nice! This is called design, by the way and it wasn't so scary, was it? I know "design" is a confusing term in software development, but it really doesn't have to be. By the way, namespaces are a nice way to organise code. We can organise previous interfaces into a *data* and a *view* namespace. Let's see how we'd like these newly created interfaced to be used. We'll have a stab at it by writing the documentation of a tabular view. Let's see:

```
public class TabularViewTest {

  @Test
  public void display() {
    DataSource ds = new DataSource() {
      @Override
      public List<String> entryDetails() {
        return Arrays.asList("text");
      }

      @Override
      public Stream<List<String>> stream() {
        return Stream.of(Collections.singletonList("data"));
      }
    };
    String expected =
        "-------------------------\n" +
        "|         text          |\n" +
        "-------------------------\n" +
        "|         data          |\n" +
        "-------------------------\n";

    String actual = new TabularView().display(ds);

    assertEquals(expected, actual);
  }
}
```

The code speaks for itself: we have a *DataSource* that we feed to a *TabularView* and we expect that the view will give us the correct format. The view knows nothing about where the data is coming from or who is going to consume its output. It doesn't even know it exists yet, because we get a compilation error when we write the test. Let's create the class (or better yet, use the tools and tell the IDE to create it for us). The test will still complain about the display method of the view, so let's create that too. We'll end up with this:

```
public class TabularView implements View {
  @Override
  public String display(DataSource ds) {
    return null;
  }
}
```

Now the test compiles just fine. We run it and it fails. Very good! Now we can go ahead and implement the display method to make the test pass. I'm not going to go through the mechanics of implementing this method, because that is not the point here (besides, you can find the code online - check at the end of the book), but it should end up looking something like this (not all implementation details are here, but we shouldn't need them to understand what is going on):

```
public String display(DataSource ds) {
  StringBuilder sb = new StringBuilder();
  String rowSeparator = rowSeparator(ds.entryDetails());
  sb.append(rowSeparator);
  sb.append(headers(ds));
  sb.append(rowSeparator);
  sb.append(data(ds));
  sb.append(rowSeparator);

  return sb.toString();
}
```

Pretty little function describing the mechanics of formatting the data as a table. With this, our test is green. We're happy!

For now, the new code is dead. No code is reaching it, apart from junit. Before we incorporate the shiny new thing, let's build a way of switching between views in the user interface:

```
while (true) {
  System.out.println("Existing appointments:");
  appointmentManager.printAppointments();

  System.out.println("Menu\nl - list view\nt - tabular view"+
                     "\na - add new appointment\nx - exit");
  String choice = br.readLine();
  if (choice.equals("x")) break;
  switch (choice) {
    case "l": appointmentManager.printAppointments(); break;
```

```
    case "t": appointmentManager.printAppointments(); break;
    case "a": appointmentManager.inputNewAppointment(); break;
    default: System.out.println("Invalid choice");
  }
}
```

We've changed the UI to make it easier for the user to switch between views, but we've also changed the addition and exit keys (they used to be only yes/no questions). This is where we go and get feedback from the business. You see, we don't wait for them to provide all the details, but we help them with proposals like these. They will most likely accept them and be happy that you've thought about it.
Anyway, let's get back to what we were developing. Let's see what we've built. Hit run and:

```
Existing appointments:
-------------------------------------------------------------------------------
|        TIME        |        DOCTOR        |        PATIENT        |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Menu
l - list view
t - tabular view
a - add new appointment
x - exit
```

Notice that we've simply provided a way to switch between views, but haven't provided any new views yes. Both *l* and *t* will print the same tabular view.
Notice something else too? The output doesn't have any appointments. But we've saved appointments on previous runs! What happened? Let's have a look at that DAO:

```
public class AppointmentDAO {

  private static List<Appointment> DB = new ArrayList<>();

  public List<Appointment> getAllAppointments() {
    return Collections.unmodifiableList(DB);
  }

  public void saveAppointment(Appointment appointment) {
    DB.add(appointment);
  }

  public void deleteAllAppointments() {
    DB.clear();
  }
}
```

Excuse me? This is an in-memory store! How do they keep track of their appointments? Wait... do they... no way... We must ask somebody from operations what's going on here. We've heard Bob from operations has a hard time maintaining the system. This looks like a tiny console app. Why would he have a hard time...? Maybe because of this in-memory store... Let's talk to Bob. After we talk to Bob, our minds are blown. There is an entire ecosystem built around this console app, just to make sure the data is saved and always available. There are clusters of machines that replicate the in-memory data and backup mechanisms that make sure secondary nodes can take over when the primary goes down. Tons of money are being spent each month to maintain this infrastructure.
Ok, this is fiction, but believe you me, this kind of madness is happening out there more often than you might suspect. Now we understand that running things locally, without the whole production infrastructure, we get no persistence between runs. Well, that's life... We deal with it and move on.

Getting back to where we left off, we need to hook the new tabular view implementation into existing code. Let's open up that *AppointmentManager* and do it:

```
public void printAppointments() {
  List<Appointment> appointments = dao.getAllAppointments();
  if (appointments != null) {
    View view = new TabularView();
    display(view.display(createDataSource(appointments)));
  } else {
    displayLine("No appointments found");
  }
}

private DataSource createDataSource(List<Appointment> appointments) {
  return new DataSource() {
    @Override
    public List<String> entryDetails() {
      return Arrays.asList("TIME", "DOCTOR", "PATIENT");
    }

    @Override
    public Stream<List<String>> stream() {
      return appointments.stream()
          .map(a ->
            Arrays.asList(
                isExpired(a.getDate()) ? "EXPIRED" : DF.format(a.getDate()),
                a.getDoctor(),
                a.getPatient()));
    }
  };
}

private boolean isExpired(Date date) {
  long time = date != null ? date.getTime() : new Date(1970, 1, 1).getTime();
  return (System.currentTimeMillis() - time) / 1000 > 3600 * 24 * 30 * 6;
}
```

We've gotten rid of the formatting and delegated the work to our newly created tabular view. We've also extracted 2 other concepts into their own methods (they're functions really, but technically we still call them methods, because they're tied to the class instance), the data source creation and the expiration business logic. Let's run it:

```
Enter time: 05/04/2021
Enter doctor: dr. Smith
Enter patient: John Doe
Existing appointments:
-------------------------------------------------------------------------------
|       TIME       |       DOCTOR       |       PATIENT       |
-------------------------------------------------------------------------------
       05/04/2021                dr. Smith                John Doe
-------------------------------------------------------------------------------
```

Great! It looks identical to what the system did before. Are we done with the tabular view? Not quite. We have one more step to go through, which will give us great satisfaction: we're going to open the *AppointmentManagerTest* class and delete the *testPrintAppointments()* test. Yes! Delete! I know that test verified the saving and retrieval of the appointments, but it did so using internal mechanisms, by creating it's own flow. This flow is not guaranteed to be the same as the production flow and even if it is, it's just a duplication that might dangerously diverge from the original. This usually happens when the production flow changes, but the test code does not. More often than not, the test will still pass, giving us a false confidence. These tests are useless! We already test the format, so delete and don't look back!

Build a list view

We've refactored the code to not only read better, but to allow us to see and differentiate the concepts we read. It should now be obvious that building a list view is exactly that: building a *ListView*! How nice, to be able to speak through code. We'll first create our expectation of the new view and consult the business whether this fits their expectation (yes, we should be able to easily guide business through our code!):

```
public class ListViewTest {
  @Test
  public void display() {
    DataSource ds = new DataSource() {
      @Override
      public List<String> entryDetails() {
        return Arrays.asList("text, other text");
      }

      @Override
      public Stream<List<String>> stream() {
        return Stream.of(
            Arrays.asList("data", "other data"),
            Arrays.asList("x", "y"));
      }
    };
    String expected = "Details (text, other text):\n- data, other data;\n- x, y;\n";

    String actual = new ListView().display(ds);

    assertEquals(expected, actual);
  }
}
```

”Yes”, they say! “The expected looks exactly like what we want”. Looks like it’s time to write the actual view and to hook it up afterwards. We confidently start writing code to make the test pass (again, details omitted for brevity):

```
public String display(DataSource ds) {
  return header(ds).append(data(ds)).toString();
}
```

The test passes, so we can hook the view into the system. Where can we hook it though? Currently, *printAppointments()* hardcodes the type of view, but we can inject the view, by extracting it as a function parameter:

```
public void printAppointments(View view) {
    ...
    View view = new TabularView();
    display(view.display(createDataSource(appointments)));
...}
```

The manager class can now print appointments in both formats. We have to tell it to do so:

```
public class Main {
  private static ListView listView = new ListView();
  private static TabularView tabularView = new TabularView();

  public static void main(String[] args) throws IOException {
    ...
    while (true) {
     ...
      switch (choice) {
        case "l":
          appointmentManager.printAppointments(listView); break;
        case "t":
          appointmentManager.printAppointments(tabularView); break;
        ...
      }
     ...
  }
```

We've created the two views as dependencies in the Main class (or shall I say namespace, because they're not related to it by anything other than scope) and injected them into the manager, based on the user's choice (key pressed). The more TDD inclined, will no doubt have noticed by now that we are also changing code that has no tests. The class *Main* in this case. This is indeed up for debate and especially so in larger code bases, where things are not as straight forward as in this case. However, this would not be TDD, as one would write the test after the fact anyway and the test would surely be influenced by reading the production code. In other words, it would be a Production Code Driven Test, or PCDT (always wonderful to invent yet another acronym that can spark flamewars, ah well...). Ain't nothing wrong with PCDT. In fact, I use it in many situations to protect against unintentional behavioural changes. A trade-off needs to be made for each situation between: "is the change obvious and are we confident

enough that it will make the system behave as expected (confidence comes from fast system feedback, not ego!)?" and "PCDT will take so long that the time spent on it would not justify adding the feature in the first place". Some systems are so bad, that actually implementing and rolling the feature out to the users, with no testing, is worth more than refactoring the codebase until it allows for testing.
It might seem that the previous statement is defeating the purpose of this book. In a way it is, but it's also a reality check. I don't advocate for fixing all the things. Some things are not worth fixing. But what is really important is for us, as engineers, to have that understanding and to know how to technically advise the business.
Getting back to our appointments, we run the new code, we add an appointment and, when presented with the menu, we select *l*, for list view:

```
Details (TIME, DOCTOR, PATIENT):
- 05/04/2021, dr. Smith, John Doe;
```

It works as expected. Now the only requirement left is the possibility of adding comments together with the rest of the details of a new appointment.
Let's stop for a bit and consider this: is the system in a good shape? Not really... at least the code is not as "accommodating" as we'd like it to be. We've already talked about the *XManager* naming style. So would this be a good opportunity to improve it? What should we base the answer to this question on? We can't just base it on the fact that the code could surely use some love, because we're serving a business purpose after all. This is where usually things go south. Engineering pushes the business to get time for cleaning up the code and business pushes

engineering to ship faster. And guess what? Both parties are right! What's with the contradiction then? Where does it come from? It's usually because the upfront discussions (the ones before the requirements were issued) did not include enough details from both sides to allow for a proper understanding of the desired outcome. I'm digressing here, but bear with me, it's worth it! It might save you a lot of headaches down the road. In every software development scenario there is a minimum of 4 players: users, business, engineering and the system. When discussing changes, business and engineering must ALWAYS triangulate decisions based on the other players. Whenever one of the 3 points are left out from this discussion, they will end up in implied expectations. The implied expectations will almost certainly be different for each party, hence the misunderstanding before shipping the change. Now we know how we should answer our initial question about moving along with refactoring. We've properly triangulated decisions together with the business and we agreed we would have some time for this improvement (of course we have, because I decided so, being the writer of the book; had we not have agreed, we would have had to rush the feature implementation based on the dirty code and ship).

Put it where it belongs!

Printing to a console can move out of the manager. We'll create a new package, called *io*, a new class, called *Console* and move the actual printing operation over there. Let's do it:

```
public class Console {
    public static void print(Object o) {
        System.out.print(o);
    }

    public static void printLine(Object o) {
        print(o + "\n");
    }
}
```

What about the data related operations? Let's just move those to the *AppointmentDAO* class for now. While we're at it, the DAO is a data source, so let's formalise this:

```
public class AppointmentDAO implements DataSource {

  private static List<Appointment> DB = new ArrayList<>();

  ...

  @Override
  public List<String> entryDetails() {
    return Arrays.asList("TIME", "DOCTOR", "PATIENT");
  }

  @Override
  public Stream<List<String>> stream() {
    return DB.stream()
          .map(a ->
                Arrays.asList(
                      a.isExpired() ? "EXPIRED" : a.getFormattedDateString(),
                      a.getDoctor(),
                      a.getPatient()));
  }
}
```

I know there's maybe more going on than it should in the DAO, but let's keep the data related stuff in this namespace. We'll decide later if there's anything better we could do. For now, it's much better than having it in an anonymous concept, like manager. Oh, yeah... and I hope you've noticed that the appointment knows whether it's expired or

not and also has a string representation of its date. This is what it looks like:

```
public class Appointment {

  public static DateFormat DF = new SimpleDateFormat("dd/MM/yyyy");

  ...

  public boolean isExpired() {
    long time = date != null ? date.getTime() : new Date(1970, 1, 1).getTime();
    return (System.currentTimeMillis() - time) / 1000 > 3600 * 24 * 30 * 6;
  }

  public String getFormattedDateString() {
    return DF.format(date);
  }
}
```

The class uses global state to determine something? Blasphemy? Usually yes! Let's try to live with this for a while, as we're still moving parts around. Maybe we'll find a better home for this.
Having moved the printing, data access and business logic to their respective places, we can adapt the main class to make use of them:

```
public static void main(String[] args) throws IOException {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  while (true) {
    Console.printLine("Menu\nl - list view\nt - tabular view...");
    ...
    switch (choice) {
      case "l":
        display(dao, listView);
        break;
      case "t":
        display(dao, tabularView);
        break;
      case "a":
        appointmentManager.inputNewAppointment();
        break;
      ...
    }
  }
}
```

```
private static void display(AppointmentDAO dao, View view) {
  if (dao.appointmentsCount() > 0)
    Console.print(view.display(dao));
  else
    Console.printLine("No appointments found");
}
```

Main uses now the *Console* for output, but not for input. The manager is still used for input. We'd like to find a nice home for the input capability, preferably close to the output capability as they're part of the same I/O (input/output) concept. By the way, we keep hitting that "run tests" button while we move these things around. When the code doesn't compile, it's ok. We make it compile and then run the tests.

We've had to adapt *AppointmentManager* to use the Console for output, as we've moved out its display functions:

```
public void inputNewAppointment() {
  ...
  try {
    Console.print("Enter time: ");
    String date = br.readLine();
    try {
      appointment.setDate(Appointment.DF.parse(date));
    } catch (ParseException e) {
      Console.printLine("invalid date");
    }

    Console.print("Enter doctor: ");
    String doctor = br.readLine();
    appointment.setDoctor(doctor);

    Console.print("Enter patient: ");
    String patient = br.readLine();
    appointment.setPatient(patient);
  } catch (IOException e) {
    // do nothing
  }

  dao.saveAppointment(appointment);
}
```

There are some data operations we can encapsulate in the data access namespace. Currently, *AppointmentManagerTest.testInputNewAppointment()* gets all the data and then uses just a small amount of it:

```
@Test
public void testInputNewAppointment() {
  ...
  assertEquals(1, dao.getAllAppointments().size());
  Appointment appointment = dao.getAllAppointments().get(0);
  ...
}
```

We're not extremely concerned with performance at this point, but we can make those queries explicit in the data layer and let it decide how to answer:

```
@Test
public void testInputNewAppointment() {
  ...
  assertEquals(1, dao.appointmentsCount());
  Appointment appointment = dao.findByIndex(0);
  ...
}
```

For now, we just move the implementation to the DAO as it is:

```
public class AppointmentDAO implements DataSource {
...
  public int appointmentsCount() {
    return DB.size();
  }

  public Appointment findByIndex(int index) {
    return DB.get(index);
  }
...
}
```

Run tests. They pass. Nice!

All that reading from the console scares us a little bit. Let's try something small first. Let's just move the part where we read the user's choice. That means this:

```
public static void main(String[] args) throws IOException {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

  while (true) {
    Console.printLine("...");
    String choice = br.readLine();
    if (choice.equals("x"))
      break;

    switch (choice) {
      ...
    }
  }
}
```

Becomes this:

```
private static final String MENU = "Menu\n" +
      "l - list view\n" +
      "t - tabular view\n" +
      "a - add new appointment\n" +
      "x - exit";


public static void main(String[] args) {
  for (char choice = Console.choice(MENU);
       choice != 'x';
       choice = Console.choice(MENU)) {
    switch (choice) {
      ...
    }
  }
}
```

We moved the mechanics of reading input into the *Console* namespace. Also, an important change is that we've dropped the *IOException* from *main*'s signature. This is a bold move and may have implications when the system exits. We need to be careful with these sort of things, as throwing exceptions has meaning in systems that allow for

it. In this case we would need to see if anything relies on the fact that the system might exit abnormally (with an exception in this case). There might be something that reads the system's output and reacts accordingly, for example by restarting it. In this case, after a bit of asking around and looking at the infrastructure running our application, we've concluded there wasn't such a mechanism in place and it was safe for us to actually handle that *IOException* and translate it in a domain value:

```
public class Console {
    private static BufferedReader SYS_IN_READER = ...(System.in));
    ...

    public static char choice(String menu) {
        printLine(menu);
        try {
            return SYS_IN_READER.readLine().charAt(0);
        } catch (IOException e) {
            e.printStackTrace();
            return 'x';
        }
    }
}
```

We still exit in case of an *IOException*, but we do so gracefully. We log the exception and return a value that means exit. The implementation is just code ported from *Main*, but housed under a name that describes the intent. I'm not hung up on names like *get*Choice for example, because I could simply say that the *choice* is a function of menu and input (input in this case is still an implicit parameter that comes from the global context - which is not so great).

Simplify

We're now going to make things more explicit and get rid of the whole *Manager* confusion. The application displays appointments and creates new appointments. Well, let's express just that:

```
public class Application {

    public static void createNewAppointment(AppointmentDAO dao) {
      ...
    }

    public static void displayAppointments(AppointmentDAO dao, View view) {
      ...
    }
}
```

Move the existing implementation from *Main* to this new class, so that this:

```
public static void main(String[] args) {
  ...
        display(listView);
       ...
        display(tabularView);
        ...
        appointmentManager.inputNewAppointment();
  ...
}
```

which calls *Main.display* and *AppointmentManager.inputNewAppointment*, will only make use of the centralised behaviour in *Application*, like so:

```
public static void main(String[] args) {
  ...
        Application.displayAppointments(dao, listView);
        ...
        Application.displayAppointments(dao, tabularView);
        ...
        Application.createNewAppointment(dao);
  ...
}
```

We've had a brief encounter with *inputNewAppointment*, when we've had to adapt it to use *Console*. Let's look at the whole thing:

```
public void inputNewAppointment() {
  Appointment appointment = new Appointment();
  appointment.setId(System.currentTimeMillis());

  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  try {
    Console.print("Enter time: ");
    String date = br.readLine();
    try {
      appointment.setDate(Appointment.DF.parse(date));
    } catch (ParseException e) {
      Console.printLine("invalid date");
    }

    Console.print("Enter doctor: ");
    String doctor = br.readLine();
    appointment.setDoctor(doctor);

    Console.print("Enter patient: ");
    String patient = br.readLine();
    appointment.setPatient(patient);
  } catch (IOException e) {
    // do nothing
  }

  dao.saveAppointment(appointment);
}
```

It creates a new appointment object, fills it in with user input (in real time) and saves it. What's up with that *setId* stuff? Turns out nobody uses the id for anything, so it's just useless code. Nice! We delete the *ApointmentManager* class and by moving this to the *Application* namespace, we broke the *AppointmentManagerTest.testInputNewAppointment*. This is a good opportunity to look at that test:

```
@Test
public void testInputNewAppointment() {
  String data = "20/10/2018\ndoctor\npatient\n";
  ByteArrayInputStream in = new ByteArrayInputStream(data.getBytes());
  System.setIn(in);
  appointmentManager.inputNewAppointment();
  assertEquals(1, dao.appointmentsCount());
  Appointment appointment = dao.findByIndex(0);
  assertEquals(new Date(2018 - 1900, 9, 20), appointment.getDate());
  assertEquals("doctor", appointment.getDoctor());
  assertEquals("patient", appointment.getPatient());
}
```

The test replaces the global system input stream (because the developer knew appointment manager uses it - this is not visible in the test - bad!) to be able to pass the input from a string. Sometimes (actually, pretty often) you will find this style of testing, where tests use and replace global state. I will show you how we can fix this here, but you won't always have the resources to do so, because you will find out that hundreds of tests are affected by this global state and touching it will have you fix, well... hundreds of tests, which is impractical. In those cases, when you need to write a test that modifies global state, this is what you need to do:

- Capture desired global state before the test
- Modify that state to reflect test preconditions
- Run the test
- Restore the global state to what it was

Ok, you will say, but this way I'll lose the possibility of running those tests in parallel. Yes, that's right! But tests that already relied on global state had a close to zero probability of being run in parallel before anyway.
For now, we'll make the test compile and pass by using the *Application* namespace instead:

```
@Test
public void testInputNewAppointment() {
  ...
  appointmentManager.inputNewAppointment();
  Application.createNewAppointment(dao);
  ...
}
```

We'll also rename the test to *ReadAppointmentTest* and remove *AppointmentManager* from everywhere.

Phew! We're happy we were able to express what the application does better. The implementation that creates a new appointment is still pretty verbose and feels like it should delegate the I/O to *Console*. The I/O simply prompts the user for input and reads strings back from the system's input stream. We can implement this inside *Console*:

```
public static Optional<String> readString(String prompt, String errMessage) {
    print(prompt);
    try {
        return Optional.of(SYS_IN_READER.readLine());
    } catch (IOException e) {
        printLine(errMessage);
        return Optional.empty();
    }
}
```

Uh oh! Did we just change the input error handling semantic? Yep! How so? Well, if you go back to the original code, you will see that there isn't much error handling happening there. In case of input errors when entering a doctor it will ignore the patient. Is that ok? Most likely not! We need to make sure, though, that no other system makes use of that bug (maybe by running a nightly cron job and counting appointments without patients for some reports). Once we do that, we also inform stakeholders we've found and will fix a bug, then we can carry on. Let's use this

function in the creation of a new appointment. Now the *Application* class looks like this:

```
public class Application {

    public static void createNewAppointment(AppointmentDAO dao) {
      Appointment appointment = new Appointment();

      Console.readString("Enter time: ","invalid date")
            .ifPresent(appointment::setDate);
      Console.readString("Enter doctor: ", "")
            .ifPresent(appointment::setDoctor);
      Console.readString("Enter patient: ", "")
            .ifPresent(appointment::setPatient);

      dao.saveAppointment(appointment);
    }

    public static void displayAppointments(AppointmentDAO dao, View view) {
      if (dao.appointmentsCount() > 0)
        Console.print(view.display(dao));
      else
        Console.printLine("No appointments found");
    }
}
```

It reads well. One thing to notice, though, is setting the date from a string. We've had to trade of responsibility for readability, by adding this to the *Appointment* model class:

```
public void setDate(String dateString) {
  try {
    date = DF.parse(dateString);
  } catch (ParseException e) {
    // don't set invalid date
  }
}
```

At least we got to see what the code can look like when it's readable. Still, it's not a nice trade-off. We want that parsing out of the model, but we like this style of reading from the *Console*. Ok then, we'll enhance the *Console* with the ability to read typed input, not just strings.

Let's stop for a second and analyse the previous statement (spoiler alert! We're doing design again). We want to give *Console* a new ability, but which ability does it already have? It can do I/O and specifically, it can do I/O using the system's input and output streams. Ok, so it seems that the only specific thing is the medium of I/O. We'll pull out the part that is not specific, in the form of a contract:

```
public interface IO {

    void print(Object o);
    void printLine(Object o);
    Optional<String> readString(String prompt, String errorMessage);
}
```

The contract specifies what behaviour the *Console* should (and in this case already does) implement. We can have it implementing this interface, but we won't do that just yet, because we would like it to read typed input, more specifically dates. Let's first express that date input reading contract:

```
public interface TypedIO extends IO {

    DateFormat DF = new SimpleDateFormat("dd/MM/yyyy");

    default Date readDate(String prompt, String errorMessage) {
        try {
            return DF.parse(readString(prompt, errorMessage).get());
        } catch (ParseException e) {
            printLine(errorMessage);
            return readDate(prompt, errorMessage);
        }
    }
}
```

Looks familiar? Yes, it's the same parser that we've temporarily misplaced in the model. Now the *Console* can implement this contract:

```
public class Console implements TypedIO
```

We can remove the date parsing from the model. This will break the code in the data access layer:

```
public Stream<List<String>> stream() {
  return DB.stream()
        .map(a ->
              Arrays.asList(
                    a.isExpired() ? "EXPIRED" : a.getFormattedDateString(),
                    a.getDoctor(),
                    a.getPatient()));
}
```

and we need to replace the parsing:

```
public Stream<List<String>> stream() {
  return DB.stream()
        .map(a ->
              Arrays.asList(
                    a.isExpired() ? "EXPIRED" : TypedIO.DF.format(a.getDate()),
                    a.getDoctor(),
                    a.getPatient()));
}
```

Design moment again: is it ok to have data transformation logic in here? Why not? We're not returning the actual format that's stored, but a different representation of it anyway. If it turns out to be a pain in the future, we'll have to revisit this decision, but for now, it looks good.

Let's go ahead and actually make use of the newly created I/O contracts. Currently, our *Application* makes direct use of the *Console*, which binds it to a specific I/O medium. We'd also like to give ourselves the freedom of choosing that medium, should we be required to. So let's decouple the *Application* from it and simply inject I/O through contracts:

```
public static void createNewAppointment(AppointmentDAO dao, TypedIO io) {
  Appointment appointment = new Appointment();
  appointment.setDate(
        io.readDate("Enter time: ","invalid date"));
  io.readString("Enter doctor: ", "")
        .ifPresent(appointment::setDoctor);
  io.readString("Enter patient: ", "")
        .ifPresent(appointment::setPatient);
  dao.saveAppointment(appointment);
}

public static void displayAppointments(AppointmentDAO dao, View view, IO io) {
  if (dao.appointmentsCount() > 0)
    io.print(view.display(dao));
  else
    io.printLine("No appointments found");
}
```

Now we're really applying dependency inversion for both the view and I/O. *Application* becomes almost an abstract algorithm, which has no knowledge of the underlying implementations. Why almost? Because we still have that concrete *AppointmentDAO* being passed in. We'll come back to the design of this part, but for now, let's have look at the wonderful thing that happened in the *ReadAppointmentTest*. It doesn't compile anymore, because we've modified the signature of *Application.createNewAppointment*, by requiring an I/O. But this is awesome, because now we don't have to depend on the system's I/O, we can provide our own! That's the whole idea behind that previous refactoring we did. We'll see the full potential of this later in the book, but for now let's write our own test I/O, for the test to use and, while we're at it, le's also write a corner case test, in which a "bad" date is passed as input:

```
public class ReadAppointmentTest {

  private AppointmentDAO dao;
  private TestIO io;

  class TestIO implements TypedIO {

    List<String> printBuffer = new ArrayList<>();
    Queue<String> readBuffer = new LinkedList<>();

    @Override
    public void print(Object o) {
      printBuffer.add(o.toString());
    }

    @Override
    public void printLine(Object o) {
      printBuffer.add(o + "\n");
    }

    @Override
    public Optional<String> readString(String prompt, String errorMessage) {
      return Optional.of(readBuffer.remove());
    }
  }

  @BeforeEach
  public void setUp() {
    dao = new AppointmentDAO();
    dao.deleteAllAppointments();
    io = new TestIO();
  }
...
```

The test I/O is nothing more than a couple of buffers that are used to mimic the behaviour of such a system, but, more importantly, can be manipulated from our test (which is the actual purpose here). This is what it looks like:

```
...

  @Test
  public void testInputNewAppointment() {
    io.readBuffer.offer("20/10/2018");
    io.readBuffer.offer("doctor");
    io.readBuffer.offer("patient");

    Application.createNewAppointment(dao, io);

    assertEquals(1, dao.appointmentsCount());
```

```
    Appointment appointment = dao.findByIndex(0);
    assertEquals(new Date(2018 - 1900, 9, 20), appointment.getDate());
    assertEquals("doctor", appointment.getDoctor());
    assertEquals("patient", appointment.getPatient());
  }

  @Test
  public void testInputInvalidDateOnNewAppointment() {
    io.readBuffer.offer("xyz");
    io.readBuffer.offer("20/10/2018");
    io.readBuffer.offer("doctor");
    io.readBuffer.offer("patient");

    Application.createNewAppointment(dao, io);

    assertEquals(1, dao.appointmentsCount());
    Appointment appointment = dao.findByIndex(0);
    assertEquals(new Date(2018 - 1900, 9, 20), appointment.getDate());
    assertEquals("doctor", appointment.getDoctor());
    assertEquals("patient", appointment.getPatient());
  }

}
```

Notice that offering input to the test buffer, remotely resembles a user typing input to the console (although it is not, but the metaphor is helpful). We've kept the rest of the (poor) style intact: test names, date algebra. This is important, especially in larger codebases where the programmers invested a lot in that style and it would be pretty tough to change multiple things at once. Usually, I'm leaving things related to style for later on in the process, after I've proven the utility of the more important change. We tend to quickly dismiss this aspect of easing changes in, but it's a very important one.

People are resisting change. All of us! Some more than others, but nevertheless we all do. At its core, this is not a bad thing. It's something that evolved to make us retain a state of wellbeing. Now the variable here is "wellbeing". In software development, we can look at wellbeing from different perspectives. When programmers are reluctant to make changes to an otherwise healthy and thriving system,

they are looking out for the wellbeing of the system and this is great. But when programmers are reluctant to change a poorly performing, difficult to modify system, then they are simply looking out for their own wellbeing, usually by trying to justify poor decisions or being afraid they'll be seen as failures. This most likely means they've had a rough time growing up, by being put down for every mistake they've made. Mistakes should be opportunities for learning. If we are afraid of making mistakes, we will not learn. Let's stop with psychology for now and carry on with programming.
I said we shouldn't change too many aspects at once, but nevertheless, I'll show you my preference for naming tests:

```
@Test
public void input_new_valid_appointment()...

@Test
public void input_invalid_date_on_new_appointment()...
```

I sometimes mention the expectation in the test name, but when I don't, it's implied in the namespace (the class name in this case): *CreateNewAppointmentTest*. Yes, I've renamed *ReadAppointmentTest*. I do this from time to time, when I think a certain name describes the actions better.

I haven't explicitly stated this so far, but I kept mentioning namespaces, contracts, behaviour and so on, but I'm going to address this now. As you might have noticed, I used the *static* qualifier quite a few times up to this point. Why? Because there simply wasn't the need for a specific object instance to be around, for those contexts. Those are basically functions, which have no need for external context other than the arguments passed in. I kept pushing the term namespace, because class represents a template

from which instance objects are created. Functions are smaller building blocks than classes. They can also be tested easier, as they do not have multiple doors, like classes do. Functions can take other functions as parameters and can also return functions. They can easily be composed and rearranged. Yes, you can achieve the same things with objects, but the overhead is quite big and frankly... not worth it. So in general, I try to keep things simple and build all I can with just functions. Let's push the system towards this and simplify it as much as we can. Let's start simple, by mapping those user choices to functions in a more direct and obvious way:

```
private static Map<Character, Runnable>
      FUNCTION_TABLE = new HashMap<Character, Runnable>() {
  {
    put('l', () -> Application.displayAppointments(dao, listView, console));
    put('t', () -> Application.displayAppointments(dao, tabularView, console));
    put('a', () -> Application.createNewAppointment(dao, console));
  }
};

private static Console console = Console.getInstance();
private static ListView listView = new ListView();
private static TabularView tabularView = new TabularView();
private static final AppointmentDAO dao = new AppointmentDAO();


public static void main(String[] args) {
  for (char choice = console.choice(MENU); choice != 'x'; choice =
      console.choice(MENU)) {
    Optional.ofNullable(FUNCTION_TABLE.get(choice))
          .orElse(() -> console.printLine("Invalid choice"))
          .run();
  }
}
```

What's the advantage of this approach? Now the algorithm in main can stay untouched, while we can always add new choice to function entries in the FUNCTION_TABLE. Also, we can observe a separation of concepts: declarative functionality, dependencies (yes, console is a singleton,

deal with it!) and runtime mechanism. All simple, straightforward, without hidden coupling or implicit rules.

What's next? I don't like the fact that I have to wait for user input inside a function parameter:

```
appointment.setDate(io.readDate("Enter time: ","invalid date"));
```

I would rather have the action the system must take, upon successful input, specified as a callback, like so:

```
io.readDate("Enter time: ", "invalid date", appointment::setDate);
```

To me, this is much more expressive, as it suggests that I/O should read the date into a specific appointment field. To make this happen, we need to overload the *readDate* function, to accept a callback:

```
public interface TypedIO extends IO {

   ...

   default Date readDate(String prompt, String errorMessage, Consumer<Date> f){
      Date date = readDate(prompt, errorMessage);
      f.accept(date);
      return date;
   }
}
```

We do the same for *IO.readString*. Now, going back to the *Application* namespace, we'd like to enhance those functions with composition capability. As they are now, they can only be called to produce... well... nothing:

```
public static void createNewAppointment(AppointmentDAO dao, TypedIO io)

public static void displayAppointments(AppointmentDAO dao, View view, IO io)
```

To give them composition capabilities, we could implement them to return functions which we can later compose with other functions at the calling site. Before we write that, let's reason about what should be fixed and what should be variable for those functions we'll return.
For the creation of new appointments, we don't need to know the *dao* and the *io*, as they only represent the "outside" world, which is not essential to the creation algorithm. As a matter of fact, the same goes for the displaying of appointments. However, we'd like to "capture" the view before we start searching for and displaying appointments. Let's see what this looks like in code:

```
public static BiFunction<AppointmentDAO, TypedIO, IO> createNewAppointment()
{
   return (dao, io) -> {
      Appointment appointment = new Appointment();
      io.readDate("Enter time: ", "invalid date", appointment::setDate);
      io.readString("Enter doctor: ", "", appointment::setDoctor);
      io.readString("Enter patient: ", "", appointment::setPatient);

      dao.saveAppointment(appointment);

      return io;
   };
}
```

We chose to return an I/O, for no apparent reason, but to have the function return something. We did the same for displaying appointments

```
public static BiFunction<AppointmentDAO, IO, IO> displayAppointments(View view)
{   return (dao, io) -> {
      if (dao.appointmentsCount() > 0)
         io.print(view.display(dao));
      else
         io.printLine("No appointments found");
      return io; };
}
```

We can now rewrite *Main* to reflect the separation between functionality and environment (we rename those dependencies to java constant standards, using uppercase, to keep more conservative users happy):

```
private static Map<Character, BiFunction<AppointmentDAO, ? super Console, IO>>
FUNCTION_TABLE = new HashMap<>() {
    {
        put('l', Application.displayAppointments(LIST_VIEW));
        put('t', Application.displayAppointments(TABULAR_VIEW));
        put('a', Application.createNewAppointment());
        put('x', (__, ___) -> { System.exit(0); return null; });
    }
};

private static final BiFunction<AppointmentDAO, ? super Console, IO>
INVALID_CHOICE = (__, io) -> {
    io.printLine("Invalid choice"); return io;
};

public static void main(String[] args) {
    while (true) {
        Optional.ofNullable(FUNCTION_TABLE.get(CONSOLE.choice(MENU)))
                .ifPresentOrElse(
                        f -> f.apply(DAO, CONSOLE),
                        () -> INVALID_CHOICE.apply(DAO, CONSOLE));
    }
}
```

We were able to express the function table in a more business readable way. I bet that if we show the function table to a business person, they'll know how to interpret it. We also named the invalid choice, to make it even more obvious in code. All those type signatures are driving us nuts, don't they? Keep reading, we'll address that too, later in the book.

Some languages are very expressive. Java is not one of them, but that should not be a deal breaker, as we can write expressive code in spite of this. We can take advantage of Java's type inference and get rid of the left hand side type declaration by using the *var* keyword. Whenever we use language features which require a different version of the

language than the one that's currently used by the system, we need to make a thorough investigation about upgrading. Upgrading can mean both compile time problems (those are easy to spot - the compiler will complain right away) and runtime problems (these are not easy to spot). Do we need to upgrade just to take advantage of new language features? No, that's not the main reason one should consider upgrading the platform used by the system. New versions usually introduce bug fixes, security patches, performance upgrades.

We upgrade the Java version and we do our due diligence with respect to regression testing. It all still works, so we're good.

There is a functional thing we'd like to address. We've brushed against it before and only addressed it from a technical perspective. It's the error handling for user input. Currently, if the user enters something the system cannot parse or there's something going wrong during the input process, the application will exit (or behave like the input was the exit command "x"). This is not very resilient. We'd like to be proactive and propose a resilience mechanism to the business. When an input error happens, an error message should be shown and the prompt for input should come back.

The business will not refuse these kinds of small improvements, unless the trade-off for delivering them is huge. For example, if the improvement will not have a big positive impact on the users (maybe they are used to the current behaviour and they've modelled their processes around it), but rather will force them to adapt and by doing so become frustrated, the business will gravitate towards not making the change. That's why we need to always discuss the functional changes with them. Remember: this

is not refactoring! Changing functionality is not just changing code shape! Business agrees to the change, so we go ahead with it:

```
public char choice(String menu) {
    printLine(menu);
    try {
        return SYS_IN_READER.readLine().charAt(0);
    } catch (Exception e) {
        printLine("Cannot read line. Try again.");
        return choice(menu);
    }
}
```

Let's step back again and have a look at how the whole story reads. We have an *Application* and a *Main*. This is awkward. Sounds like two of the same thing. I know we were the ones that decided to do it, but it does look kind of silly. What were we thinking? Never mind, we can change it. Let's get rid of that *Main* and have the application take over the role of the entry point, by renaming *Main* to *Application*, that is. What about the old *Application* then? That should really be named *Appointments*, as that is the perfect name for operations on and with... appointments. We can then shorten the operation names as well. Let's see what this look like:

```
public class Appointments {

    public static BiFunction<AppointmentDAO, TypedIO, IO> createNew() {
        return (dao, io) -> {
            var appointment = new Appointment();
            io.readDate("Enter time: ", "invalid date", appointment::setDate);
            io.readString("Enter doctor: ", "", appointment::setDoctor);
            io.readString("Enter patient: ", "", appointment::setPatient);

            dao.saveAppointment(appointment);

            return io;
        };
    }
```

```
    public static BiFunction<AppointmentDAO, IO, IO> display(View view) {
        return (dao, io) -> {
            if (dao.appointmentsCount() > 0)
                io.print(view.display(dao));
            else
                io.printLine("No appointments found");
            return io;
        };
    }
}
```

This will force us to adapt some production and test code. In particular, we'll have to adapt the FUNCTION_TABLE:

```
FUNCTION_TABLE = new HashMap<>() {
    {
        put('l', Appointments.display(LIST_VIEW));
        put('t', Appointments.display(TABULAR_VIEW));
        put('a', Appointments.createNew());
        put('x', (__, ___) -> { System.exit(0); return null; });
    }
};
```

Now it reads even better! If you squint your eyes and look past the Java syntax, it can serve as documentation (which is what clean code is all about).

One last thing, before we implement appointment comments, it would be nice to get rid of those horrible old *Date* objects from the *Appointment* model and the ugly arithmetic the code is doing with them:

```
public boolean isExpired() {
  long time = date != null ? date.getTime() : new Date(1970, 1, 1).getTime();
  return (System.currentTimeMillis() - time) / 1000 > 3600 * 24 * 30 * 6;
}
```

Later Java versions have better date related APIs and we'd like to use them. We have 2 options. Either change the model's date type from *Date* (old API) to *LocalDate* (new API) or wrap the field in a *LocalDate* variable inside

isExpired, do the arithmetic with the new API and leave the field as it is. The second approach is safer, because it won't break APIs and compatibility, but we'll take the first approach anyway, just because I want to discuss the approach. First, let's see what I mean by the first approach. We'll change this:

```
private Date date;
```

into this:

```
private LocalDate date;
```

Granted, there will be a couple of places in the codebase where we need to adapt the date parsing style, but that's trivial. Also, the compiler will make it trivial for us to match the new type everywhere it's being used.

Now, the real reason I wanted to take this route is to discuss what we could potentially break if we do this and how we should approach such a change. Since we are touching the model, we can create incompatibility with old, persisted models. What we also have to think about in such situations is a migration strategy, to avoid runtime errors while the new code encounters old models. Basically there are 2 sides to the model change coin:

1. Adapt old code to new model
2. Adapt new code to old model

In the first case, the compiler can pretty much do the job on its own while in the second case, the programmer needs to pay close attention to the migration strategy (updating

storage models, cron jobs, etc.). In our case, we have an in memory storage (a list), but if you recall from earlier in the book, this is replicated between machines, for fault tolerance and availability. So we'll have to look into that replication mechanism and come up with a migration strategy (a mental exercise for the reader, as the infrastructure is not part of this book's scope... maybe in the next book).
Going back to the expiration question, we can write it better, by making use of the new date API:

```
public boolean isExpired() {
    return date.isBefore(LocalDate.now())
            && Period.between(date, LocalDate.now()).getMonths() > 6;
}
```

No more dodgy year subtractions and seconds multiplication, just a clear exposition of our intention.

Using new language features can be a good opportunity to improve readability. For example we can transform this snippet from *Appointment.display*:

```
if (dao.appointmentsCount() > 0)
    io.print(view.display(dao));
else
    io.printLine("No appointments found");
return io;
```

to this:

```
switch (dao.appointmentsCount()) {
    case 0  -> io.printLine("No appointments found");
    default -> io.print(view.display(dao));
}
```

Since we are simplifying code, let's do something controversial, both because it's going to make code more

readable, but because it's also going to hurt the system in a way that is not obvious until runtime. We'll get rid of the loop in main:

```
public static void main(String[] args) {
    while (true) {
        Optional.ofNullable(FUNCTION_TABLE.get(CONSOLE.choice(MENU)))
                .ifPresentOrElse(
                        f -> f.apply(DAO, CONSOLE),
                        () -> INVALID_CHOICE.apply(DAO, CONSOLE));
    }
}
```

And replace it with a recursive call:

```
public static void main(String[] args) {
    Optional.ofNullable(FUNCTION_TABLE.get(CONSOLE.choice(MENU)))
            .ifPresentOrElse(
                    f -> f.apply(DAO, CONSOLE),
                    () -> CONSOLE.printLine("Invalid choice"));
    main(args);
}
```

Oh, and yes, the INVALID_CHOICE is inlined once again. Now functional programmers do this a lot: recursive calls. In some programming languages, this is the preferred way of looping. However, recursion implies building on top of the stack and if this is not optimised (with tail call optimisation for example), the stack will blow up after a certain amount of recursions. This is exactly what will happen in Java, in this particular case. On the other hand, the code looks cleaner, with less lines and less indentation. No worries, we'll put the loop back in the end, but I wanted to emphasise the fact that these changes are dangerous, especially without proper test harnesses (this will run just fine if we just enter a couple of thousand inputs, but would crash with a proper stress test).

Adding comments to an appointment

We've "massaged" the system enough to allow us to easily continue with the addition of comments on every new appointment and with displaying them. Let's satisfy the TDD inclined readers and refrain from simply adding the comments property to the model first and instead write a test for this first. Should we write a test for the model? What then, test the mutator and accessor for that property? This is silly (as silly as this seems, there's a lot of that going on out there, in the wild)! So what would make more sense then? Maybe verify this through the handling of data, in the data access layer. We don't have a direct test for the DAO yet anyway, so let's write one now:

```
class AppointmentsDataSourceTest {

    private final AppointmentDAO dao = new AppointmentDAO();

    @Test
    public void comment_headers_are_displayed() {
        assertEquals(asList("TIME", ..., "COMMENTS"), dao.entryDetails());
    }


}
```

We add a test making sure that the data source will include comments in the headers. The test will obviously fail, because we haven't added them to the headers yet, so let's go ahead and do that:

```
public class AppointmentDAO implements DataSource {
...
  private static final List<String> HEADERS =
            Arrays.asList("TIME", “DOCTOR”, "PATIENT", "COMMENTS");

...
  @Override
  public List<String> entryDetails() {
    return HEADERS;
  }
...
}
```

We’re not going to die if we add the comments header AND we extract the headers to a constant before we run the test again. Or if you’re hurting, run the test and then extract, but honestly, if indeed it hurts to do the refactoring in one move, step back and reconsider your approach to coding for a bit. Are you focusing on the right aspects?
Anyway, test passes, let’s write the one for the actual data:

```
@Test
public void comments_are_shown_in_details() {
    dao.saveAppointment(new Appointment()
            .withDate(LocalDate.now())
            .withDoctor("D")
            .withPatient("P")
            .withComments("comments"));

    assertTrue(dao.stream().anyMatch(l -> l.contains("comments")));
}
```

We’ve done it! We’ve made the test fail without even running it, because “not compiling is failing”. Constructing data should be easy, so this builder pattern helps us with that. So, to make the test compile, we’ll obviously add those builder methods to the model and, we’ll add the comments property as a *String* (see the assertion). Again, you can do this in very small steps, if you prefer, or all at once, depending on your inclinations:

```
public class Appointment {

    ...
    private String comments;

    ...
    public String getComments() {
        return comments;
    }

    public void setComments(String comments) {
        this.comments = comments;
    }

    ...
    public Appointment withDate(LocalDate date) {
        this.date = date;
        return this;
    }

    public Appointment withDoctor(String doctor) {
        this.doctor = doctor;
        return this;
    }

    public Appointment withPatient(String patient) {
        this.patient = patient;
        return this;
    }

    public Appointment withComments(String comments) {
        this.comments = comments;
        return this;
    }
}
```

We now have the infrastructure to support the addition of comments to an appointment when we create it. We'll enhance the user interface to support that too. As we've gotten used to by now, we'll write the test first, or rather enhance the existing ones:

```
public class CreateNewAppointmentTest {
    ...
    @Test
    public void input_new_valid_appointment() {
    ...
      io.readBuffer.offer("comments");

      Appointments.addNew().apply(dao, io);
      ...
      assertEquals("comments", appointment.getComments());
    }

    @Test
    public void input_invalid_date_on_new_appointment() {
      ...
      io.readBuffer.offer("comments");

      Appointments.addNew().apply(dao, io);
      ...
      assertEquals("comments", appointment.getComments());
    }
}
```

The tests will fail, because we haven't implemented the reading of comments. We're ready to implement it:

```
public static BiFunction<AppointmentDAO, TypedIO, IO> addNew() {
    return (dao, io) -> {
        var appointment = new Appointment();
        ...
        io.readString("Enter comments (if any): ", "", appointment::setComments);
    }
}
```

Run the test again. It passes. Very good! Are we done? Not quite. We need to check the whole system. Fire up the application, enter a few appointments, show them in both formats, it works! Are we done now? If you paid close attention to what I've said earlier about changing the model, you would've already asked yourself what happens when this change goes live with the existing models, which do not have comments and with their synchronisation between machines. If you did ask yourself that, great! If not, just remember to think of all those aspects when

delivering changes. In the current production architecture, we'd have to enhance the serialiser that does the sync.

After we make sure all works as expected, we should deliver. We refactored enough to allow for further similar enhancements. Some might say that the code looks good as it is. Others will disagree. This dispute is pointless as long as the battleground is meta-code (e.g. design patterns, function/class names etc.). The reason we've pushed through with this refactoring is this: business wanted to enhance the appointment model. This should be an easy change. It wasn't! At least not in the original code base. Now we've made it easy. Now is the time to deliver. No more refactoring before any delivery!

This is it! We've delivered the changes.

Chapter 3 - Creating new doors

We achieved our business goal. We've deployed the changes! Now, we can massage our system to make it even easier to change.

Extract a library

Let's consider for a moment that a console user interface is not very attractive, at least not for these kinds of operations. I can only imagine the frustration of users. We should prepare the system for a UI change, one that would be easy to swap in, or even use in parallel with the existing one. This sounds like we could externalise the whole I/O concept to a library. We can even create an open source project to host this library. So everything under the *io* package will be moved out of our project to an open source one (*https://github.com/dannicolici/io*). We'll then simply

import this lib as a dependency and update the imports, wherever the old deleted code was imported.

Objects? Why?

When one picks up a certain tool, without questioning it and then uses it for a very long time, it becomes THE one true tool. This is what happens with object oriented programming (or OOP in short) too.

Try to remember what was the reason you started using OOP. Generally, the promise was that it will model the real world as closely as possible and you will have an easier time modelling business concepts. Do you remember? We have an *Animal* class that can *makeNoise()* and we derive a *Cat* and a *Dog* from it, each of which specialise the *makeNoise()* behaviour (one meows and the other barks).

We had tons of examples with specific *Car* models and all kinds of abstractions that the modellers had fun modelling. Little did they know that they would create a monster that will come back to haunt them. You would have noticed that the emphasis was on class inheritance. Abstractions that derive from other abstractions and so on. This would give rise to extremely brittle programs that would break if you changed one small thing inside an abstraction.
Also, tons of almost identical abstractions were created, because it was impossible to nail it from the get go and it was too complex and dangerous to modify the originals.

Enter OOP principles and design patterns. These are tones of rules that protect the programmers from shooting themselves in the foot. So now you had to learn all those complex technical concepts and get the domain logic right. Phew! That sounds like a lot. It is! It doesn't matter. Programmers are supposed to be engineers, so they need to do the due diligence and keep up with technology. And so we did. Many of us.

But the elephant in the room is this: OOP was never meant to be what it became. This statement comes directly from the man that coined the term OOP, Alan Kay. I strongly advise the reader to dig through some history of OOP. Alan Kay envisioned objects (n.b. not classes) as biological cells. They had their own internal processes and interacted with other cells through a specific interface. He called this message passing. This was the main idea behind OOP, not inheritance. This would have allowed for small computational units to be sent out in the world and talk to each other in a meaningful way. So you could have tons of objects that could tell each other to execute what they were created for. Never would an object ask for internals of another. Now fast forward to today and see if this is at all what OOP means. There is a language/platform that is still true to the original OOP concept today and that is Erlang/BEAM. Erlang's processes are exactly the objects Alan Kay was proposing. What is awesome about Erlang is that a process can be written using functions alone. This is great! Functions are great! Please note that functions in programming are not the same as those in mathematics. Why is that? Because they exist in a medium and they can access and modify that medium. This is where most programmers have a hard time with functional programming (or FP in short). FP is based on lambda

calculus, a mathematical language created by Alonzo Church in the 1930's. In lambda calculus, everything is expressed as a function, even numbers (now you understand the link between the anonymous computations we pass around in our programs and the term "lambda"). Clever FP people have thought of a trick to express the runtime medium of the function as a function too. This has originated in the Haskell community and has taken a more formal route, one that tries to create an algebraic language, which programmers can rely upon to do the correct algebra within their domain (given they know how to model their domain for that in the first place - not an easy task).
I took a side road for a few seconds back there, but I believe it's worth awakening the reader's interest in those areas. Getting back to the section's question: *why objects?*, I hope you now understand the reason behind the original concept of an object and also understand why most OOP languages nowadays fail to deliver that concept. The answer to *why objects?* is because they are simple to use, test, scale and reason about. What we build in Java, for example, by using classes, results in something that fails to meet at least one of those things. Do we have something we can use though? Sure! Let's have a look!

Reduce inheritance

We're going to move away from inheritance towards a simpler way of expression. Earlier, we have introduced the *View* interface, to abstract a concept. Why did we need to abstract the concept? Because a caller would be able to use the abstraction, without the need to understand what hides behind. But there is another way of achieving this, using

functions (for now, Java's *Function* object). So, let's delete the *View* interface. This will break the *ListView* and *TabularView* classes. The only function in the *View* interface was a function from a *DataSource* to a *String*. We can formalise this by implementing this already existing Java interface: *Function*.

```
public class ListView implements Function<DataSource, String> {

  @Override
  public String apply(DataSource ds) {
    return header(ds).append(data(ds)).toString();
  }
  ...
}

public class TabularView implements Function<DataSource, String> {
  ...

  @Override
  public String apply(DataSource ds) {
    StringBuilder sb = new StringBuilder();
    String rowSeparator = rowSeparator(ds.entryDetails());
    sb.append(rowSeparator);
    sb.append(headers(ds));
    sb.append(rowSeparator);
    sb.append(data(ds));
    sb.append(rowSeparator);

    return sb.toString();
  }
  ...
}
```

Now instead of the old *display* function, we'll use the new *apply* function in the callers. I won't show this easy refactoring for all callers, but for *Appointments*. Basically it's just a matter of importing the *Function* interface and replacing calls to *display* with *apply*. Sometimes this technique comes in handy: first rename the function inside the class (using IDE tools) and then you'll only need to import the interface in the callers.

```
public class Appointments {

    ... display(Function<DataSource, String> view) {
        return (dao, io) -> {
            ... view.apply(dao));
            return io;
        };
    }
}
```

For some reason, the creators of Java didn't add at least the syntactic sugar to allow for straightforward function calls: *f(a)* vs. *f.apply(a)* when working with function objects (what a strange thing to say... function object!). We can go a step further and not derive from the function *Function* interface at all, but simply declare the format as a function:

```
public class ListView {

  public static Function<DataSource, String> format =
                                        ds -> header(ds).append(data(ds)).toString();
  ...
}
```

We'll obviously need to declare the *header* and *data* private functions as *static*. The same thing will happen with the *TabularView* class:

```
public class TabularView {

  ...

  public static Function<DataSource, String> format = ds -> {
    StringBuilder sb = new StringBuilder();
    String rowSeparator = rowSeparator(ds.entryDetails());
    sb.append(rowSeparator);
    sb.append(headers(ds));
    sb.append(rowSeparator);
    sb.append(data(ds));
    sb.append(rowSeparator);

    return sb.toString();
  };
  ...
}
```

Callers won't have to create instances of these 2 classes anymore, in order to make use of their behaviour. They'll simply use these new functions we've just created. This is what the call site looks like in the *Application* class:

```
FUNCTION_TABLE = new HashMap<>() {
    {
        put('l', Appointments.display(ListView.format));
        put('t', Appointments.display(TabularView.format));
        ...
    }
};
```

If you've ever read the famous design patterns book by the "GoF" (gang of four) team, you would have noticed the "favour composition over inheritance" phrase. I have an addendum to that: "favour function composition over object composition".

In the *Appointments* namespace, we return the 2 functions that they themselves return an *IO*, but really they "consume" the I/O operations inside (a.k.a. side effects). In the spirit of FP purism, we should really just add a description of the I/O computation and return that. We won't do that now, but instead emphasise the fact that these "functions" are really just procedures. They consume some input, creating some side effects and return nothing. Java has something known as a *Consumer* which is well suited for expressing this concept. In our case, we need a *Consumer* of 2 things, or a *BiConsumer*.

```
public class Appointments {

   public static BiConsumer<AppointmentDAO, TypedIO> addNew() {
      return (dao, io) -> {
         ...
      };
   }

   public static BiConsumer<AppointmentDAO, IO> display(
                                          Function<DataSource, String> view) {
      return (dao, io) ->
            ...
   }
}
```

In both cases, the functions return a *BiConsumer*, or put differently a side effect. Again, Java is not very consistent with its API and wherever we had the caller executing the returned function, by calling *apply*, we have to modify by calling *accept* instead. For example:

```
Appointments.addNew().~~apply~~accept(dao, io);
```

We can rewrite these two functions by removing Java's syntactic sugar, i.e. *name(params)*, like so:

```
public class Appointments {

   public static BiConsumer<AppointmentDAO, TypedIO>
         addNew = (dao, io) -> {...};

   public static Function
         <Function<DataSource, String>, BiConsumer<AppointmentDAO, IO>>
         display = view -> (dao, io) -> ...;

}
```

Now we simply have a function and a procedure inside a namespace. We can reuse and compose them with other functions without all the OOP ceremony and pitfalls:

```
import static ro.bitgloss.Appointments.*;
...
FUNCTION_TABLE = new HashMap<>() {
    {
        put('l', display.apply(ListView.format));
        put('t', display.apply(TabularView.format));
        put('a', addNew);
        ...
    }
};
...
```

Pretty neat, right? We'll push through with this idea of replacing class abstractions with functions. Just like we did with the *View* interface, we'll delete the *DataSource* interface as well. This is a bold move and a lot of things will explode. Let's fix the explosions first and we'll see what happens afterwards. Let's look into the data access layer first. We'll have to drop the inheritance from *DataSource*, and we'll have to make the *entryDetails* and *stream* separately available now. It's a sort of normalisation of an abstraction if you will. We'll also do an improvement in the way the content is returned. Currently, every time the content is returned, a stream is eagerly produced. We'd like the option to have a grasp on the content stream, but not have it produced every time we call the *AppointmentsDAO.stream()* function. We'll therefore return a *Supplier* of *Stream*, which the caller might or might not ask to supply the stream. This way, the appointment stream will be lazily produced. The same lazy concept can be applied to saving an appointment, so instead of immediately saving the appointment, we give the caller a function which saves the appointment, to execute at its own convenience. A *Consumer* is used for this side effect. Since the data access layer will now be just a namespace, we'll make all the functions inside it static, as they won't

need an instance to hang on to anymore. Let's look at the brand new data access layer, shall we?

```
public class AppointmentDAO {

  private static List<Appointment> DB = new ArrayList<>();
  public static final List<String> HEADERS = Arrays.asList(...);

  public static int appointmentsCount() {
    return DB.size();
  }

  public static Appointment findByIndex(int index) {
    return DB.get(index);
  }

  public static Consumer<Appointment> save =
                                appointment -> DB.add(appointment);

  public static void deleteAllAppointments() {
    DB.clear();
  }

  public static Supplier<Stream<List<String>>> content = () ->
     DB.stream()
          .map(a ->
                Arrays.asList(
                      a.isExpired() ? "EXPIRED" : a.getDate().format(TypedIO.DF),
                      a.getDoctor(),
                      a.getPatient(),
                      a.getComments()));
}
```

Note that we've moved the headers to a constant, as they are exactly that: constant. Getting the count of all appointments is not changed, since in this case it's an O(1) operation and the same goes for *findByIndex*. There! We fixed the mess we've made in the data access layer, by removing the *DataSource* abstraction. By the way, welcome to the "wonderful" world of type systems! We'll make it worse before we make it better, don't worry.

The next mess we'll fix is in the view namespaces. We'll take the *ListView* first. It's not very painful. We simply need

to do the same normalisation as before. Let's see what that looks like:

```
public class ListView {

  public static BiFunction<List<String>, Supplier<Stream<List<String>>>, String>
                listFormat = (headers, content) ->
                              header(headers).append(data(content)).toString();

  private static StringBuilder data(Supplier<Stream<List<String>>> content) {
    ...content.get().forEach(...);...
  }

  private static StringBuilder header(List<String> hs) {
    ...hs.forEach(...);...
  }
}
```

Pretty straightforward again, it's just that the signature of that *listFormat* grew pretty ugly. Hang in there! Let's do the same for the tabular view:

```
public class TabularView {
    ...
    public static BiFunction<List<String>, Supplier<Stream<List<String>>>, String>
                  tabularFormat = (headers, content) -> {
                          String rowSeparator = rowSeparator(headers);
                          return rowSeparator +
                                  headers(headers) +
                                  rowSeparator +
                                  data(content) +
                                  rowSeparator;
                  };

    private static StringBuilder data(Supplier<Stream<List<String>>> content) {
        ...content.get().forEach(...);...
    }

    private static StringBuilder headers(List<String> hs) {
        ...hs.forEach(...);...
    }
    ...
}
```

Same ugly type signature for *tabularFormat*. We'll fix it, I promise!

Next up, the *Appointments* namespace. We'll do the *addNew* first. This is now returning a *BiConsumer*, so a side effect, but we'll transform it into a function:

```
public static Function<Consumer<Appointment>, Consumer<TypedIO>> addNew =
      dao ->
            io -> {
               var appointment = new Appointment();
               io.readDate("Enter time: ", "invalid date", appointment::setDate);
               io.readString("Enter doctor: ", "", appointment::setDoctor);
               io.readString("Enter patient: ", "", appointment::setPatient);
               io.readString("Enter comments (if any): ", "",
                                                   appointment::setComments);

               dao.accept(appointment);
            };
```

Whoa there! What just happened? Let's simply read the type signature and I promise that will make it easier to understand. It's a function that takes in an appointment consumer and returns an (typed) I/O consumer. We named the first input *dao*, but really, that's just because we leak caller details in a higher abstraction. We should probably have named it *appointmentConsumer*, since it could do anything with that appointment. I promised we'll simplify types later, so let's just leave it for now. Moving on to *display*:

```
public static Function<
      BiFunction<List<String>, Supplier<Stream<List<String>>>, String>,
      BiFunction<List<String>, Supplier<Stream<List<String>>>, Consumer<IO>>>
          display = view ->
                      (headers, content) ->
                        io ->
                          io.print(content.get().count() == 0 ?
                              "No appointments found\n" :
                              view.apply(headers, content));
```

The first *BiFunction* is the view or presentation function and the second one takes the data and a consumer that, well... consumes it. All those lists and suppliers are nothing

else but the headers and content. Again, I promise we'll encode this whole story in code abstractions. We'll modify one last production namespace that exploded and then we'll fix the compiler errors in the tests. The *Application* namespace gets updated with the normalised types and gets a bit of renaming too:

```
public class Application {

    private final static Console CONSOLE = Console.getInstance();

    private static final String MENU = """
            Menu
            l - list view
            t - tabular view
            a - add new appointment
            x - exit
            """;

    private static final Map<Character, Consumer<? super Console>>
            CHOICE_TO_FUNCTION = new HashMap<>() {
        {
            put('l', display.apply(listFormat).apply(HEADERS, content));
            put('t', display.apply(tabularFormat).apply(HEADERS, content));
            put('a', addNew.apply(save));
            put('x', (_ignore) -> System.exit(0));
        }
    };

    public static void main(String[] args) {
        CONSOLE.choice(MENU)
                .flatMap(c -> ofNullable(CHOICE_TO_FUNCTION.get(c)))
                .ifPresent(f -> f.accept(CONSOLE));
        main(args);
    }
}
```

I omitted the imports, for brevity. The old function table, now *CHOICE_TO_FUNCTION*, still reads well. Go back and read it from a non-programmer point of view. Did you do it? Did it feel natural? What about if Java would have helped us a bit more and we could have written this instead: *display(listFormat)(HEADERS, content)*? Nothing wrong with a little dreaming.

On to the tests, but we won't look into the data access layer tests, as those are boring. We simply have to adjust for the new API in *AppointmentDAO*. The really interesting stuff, where we start to see how using functions reduces boilerplate is the two view tests. A quick reminder of what the list view test used to look like:

```
@Test
public void display() {
  var ds = new DataSource() {
    @Override
    public List<String> entryDetails() {
      return Collections.singletonList("text, other text");
    }

    @Override
    public Stream<List<String>> stream() {
      return Stream.of(
          Arrays.asList("data", "other data"),
          Arrays.asList("x", "y"));
    }
  };
  var expected = "Details (text, other text):\n- data, other data;\n- x, y;\n";

  var actual = ListView.format.apply(ds);

  assertEquals(expected, actual);
}
```

Right? Implement interface, override, curly braces (oops, blasphemy!), all the ceremony. Now let's see what came out of our normalisation refactoring:

```
@Test
public void display() {
  var expected = "Details (text, other text):\n- data, other data;\n- x, y;\n";

  var actual = ListView.listFormat.apply(
        Collections.singletonList("text, other text"),
        () -> Stream.of(
              Arrays.asList("data", "other data"),
              Arrays.asList("x", "y")));

  assertEquals(expected, actual);
}
```

Pretty cool! Just call a function with some arguments. This is what "easy-to-test code" means. We didn't have to kick-start a whole new life form in order to finally test a function call. It now allows us to focus on the domain more. Which reminds us that we've touched the eager/lazy loading mechanism, when we introduced that supplier, remember? In Java (and other languages too), when we try to reuse a stream we get unpredictable results at best, but most likely we get en exception. So, we'll write a test that simply proves we can safely do multiple calls. We just do the multiple calls, without any assertion, because an exception would make the test fail and given its name, we'd know why it failed:

```
@Test
public void calling_format_twice() {
  var headers = Collections.singletonList("text, other text");
  Supplier<Stream<List<String>>> content = () -> Stream.of(
          Arrays.asList("data", "other data"),
          Arrays.asList("x", "y"));

  ListView.listFormat.apply(headers, content);
  ListView.listFormat.apply(headers, content);
}
```

We could extract the headers and content, since they duplicate the ones in the previous test. Whatever you think works for you. Some people like to see the test data inside the test, others say code is code and shouldn't be duplicated. It's a matter of perspective and trade-offs, I think, so you get to make that decision based on your own circumstances. I decided to leave the duplication in.

We can simplify the main function a bit further:

```
public static void main(String[] args) {
    CONSOLE.choice(MENU)
            .map(CHOICE_TO_FUNCTION::get)
            .ifPresent(f -> f.accept(CONSOLE));
    main(args);
}
```

Represent data as data

Before we move on to the type vindication I mentioned, we're going to make a slight detour. The model is now a class. This is a language limitation, because the model is data, not a class of something. Lots of programming languages have specific concepts that come closer to the notion of data and lately, Java has introduced such a concept too. It's called a *record*. This comes with a variety of nice, built-in helpers, that remove the need for boilerplate code. We are therefore going to use this *record* concept with our model:

```
public record
Appointment(LocalDate date, String doctor, String patient, String comments) {
    public boolean isExpired() {
        return date.isBefore(LocalDate.now()) &&
                    Period.between(date, LocalDate.now()).getMonths() > 6;
    }
}
```

Please note that at the time of writing this, some features are still in preview mode in the language and might be removed in future versions (this actually happened a couple of times while evolving the project). We can see that the new model is constructed together with all its properties. It's also somewhat immutable (docs say its shallowly immutable), meaning that the properties of a *record* can contain mutable properties of their own, but still it's a good step forward.

We'll construct new instances like this:

```
public static Function<Consumer<Appointment>, Consumer<TypedIO>> addNew =
      dao ->
            io -> {
               var appointment = new Appointment(
                     io.readDate("Enter date: ", "invalid date"),
                     io.readString("Enter doctor: ", "").orElse(""),
                     io.readString("Enter patient: ", "").orElse(""),
                     io.readString("Enter comments (if any): ", "").orElse(""));

               ...
            };
```

And we'll have convenience methods for each property, which we can use like so:

```
public static Supplier<Stream<List<String>>> content = () ->
  DB.stream()
       .map(a ->
             Arrays.asList(
                   a.isExpired() ? "EXPIRED" : a.date().format(TypedIO.DF),
                   a.doctor(),
                   a.patient(),
                   a.comments()));
```

Yes, I know... we kept the *isExpired* in the model, but so what? It's a property of the model too.

Type abstractions

Finally! We've gotten to the point where I cannot digress anymore and I have to deliver what I've promised. Let's take care of those messy type signatures. We'll take them one by one, to make things easier to follow. Java doesn't have type aliases, but we'll construct some out of existing language tools. First, let's make sure we understand what a type alias is. It is simply another way of referring to the same type. Let's refresh our memories on the view format type signature:

```
public static BiFunction<List<String>, Supplier<Stream<List<String>>>, String>
        listFormat =
       (headers, content) -> header(headers).append(data(content)).toString();
```

That long, generic *BiFunction* means that we want to take the headers (a *List<String>*) and content (a *Supplier<Stream<List<String>>>*) as input and transform them into another shape (a *String*). This is basically what we mean by view. We'll create a new namespace, called *Types* and define this view type in there:

```
public interface Types {

   interface View extends
            BiFunction<Collection<String>,
                       Supplier<Stream<Collection<String>>>, String> {}

}
```

Here we go. That wasn't too difficult. Ah and also, we've generalised those lists to collections. Going back to the *ListView*, we'll simply use this new type alias in our *listFormat* type signature:

```
public static View listFormat = (headers, content) ->
      header(headers).append(data(content)).toString();
```

We've changed nothing in functionality, but added a ton of value to our code shape. It's way more readable now, isn't it? We'll do the same to our tabular format:

```
public static View tabularFormat = (headers, content) -> {
   String rowSeparator = rowSeparator(headers);
   return rowSeparator +
          headers(headers) +
          rowSeparator +
          data(content) +
          rowSeparator;
};
```

Obviously, we'll also have to satisfy the compiler when it complains that it expects lists all over the place (remember we've changed them to collections) where view types are being passed around.
Let's look at the old type signature for the function that displays these views:

```
public static Function<
      BiFunction<List<String>, Supplier<Stream<List<String>>>, String>,
      BiFunction<List<String>, Supplier<Stream<List<String>>>, Consumer<IO>>>
      display = ...
```

We recognise the first argument to the returned function as being the *View* alias we've just defined. The return of that function also has a *View* as an argument, but it also takes a *Consumer* as a second argument. Ok, that means we can reuse the *View* alias, but we need to create an alias for the return function. Again, we design the alias by going through the existing type signature. It's a *BiFunction* that takes a view and a consumer, that means it's probably going to do some side effects, like actually writing that view somewhere (we know it does). So it takes a view and it writes it, hmmm... What about *ViewWriter*? Yeah, that sounds good.

```
public interface Types {

   interface View
   extends BiFunction<Collection<String>,
                      Supplier<Stream<Collection<String>>>, String> {}

   interface ViewWriter<W>
   extends BiFunction<Collection<String>,
                      Supplier<Stream<Collection<String>>>,
                     Consumer<W>> {}
}
```

We use a generic type for the consumer, because we don't need to enforce a specific type. Did we improve the readability of the display function?

```
public static Function<View, ViewWriter<IO>> display = ...
```

There's no doubt that we did. Could we have chosen a different way to represent those types? Definitely! I even encourage you to think of alternatives. It's a good exercise for getting better at modelling such things.
We're left with the function that reads appointments and saves them to the data store. Its current type signature is this:

```
public static Function<Consumer<Appointment>, Consumer<TypedIO>>
addNew = ...
```

It takes a consumer and returns a consumer. This means it connects 2 side-effects. And it really does. It reads some input and writes some content, both of which are side effects. We'll create a type alias for it, naming it bluntly, just like we did we the other ones:

```
public interface Types {

    interface View
    extends BiFunction<Collection<String>,
                       Supplier<Stream<Collection<String>>>, String> {}

    interface ViewWriter<W>
    extends BiFunction<Collection<String>,
                       Supplier<Stream<Collection<String>>>,
                      Consumer<W>> {}

    interface ReadWriter<R, W> extends Function<Consumer<R>, Consumer<W>> {}

}
```

ReadWriter. Again, we parameterised the consumed types.

Using this type alias, the *addNew* function's type signature becomes:

```
public static ReadWriter<Appointment, TypedIO> addNew =
      writer -> reader -> {
         var appointment = new Appointment(
               reader.readDate("Enter date: ", "invalid date"),
               reader.readString("Enter doctor: ", "").orElse(""),
               reader.readString("Enter patient: ", "").orElse(""),
               reader.readString("Enter comments (if any): ", "").orElse(""));

         writer.accept(appointment);
      };
```

Also, we now have a good name for the lambda parameters: *writer* and *reader*. Again, we could have done this differently, by splitting up the reading and writing and composing them back in the *addNew* function. It's another thought exercise for the reader (pun intended). While we're at it, why don't we simplify this function even further? We have no need for that intermediate variable. It's pretty clear what a *new Appointment* means, so:

```
public static ReadWriter<Appointment, TypedIO> addNew =
      writer -> reader -> writer.accept(new Appointment(
            reader.readDate("Enter date: ", "invalid date"),
            reader.readString("Enter doctor: ", "").orElse(""),
            reader.readString("Enter patient: ", "").orElse(""),
            reader.readString("Enter comments (if any): ", "").orElse("")));
```

Phew! That was quite a ride in type-land. We've gotten pretty far and the code shape is quite different. We're expressing most things through functions now. I cannot know your preference, but I certainly prefer less boilerplate, straight to the point, easier to compose and test code. I can only hope that you do too. There's one more thing I'd like to change and then we'll see what kind of possibilities this new code shape has offered us.

There is one thing I'd like to do for the *display* function:

```
public static Function<View, ViewWriter<IO>> display = view ->
      (headers, content) -> io ->
            io.print(content.get().count() == 0 ?
                  "No appointments found\n" :
                  view.apply(headers, content));
```

We notice that there is no test for the second case, the one with no appointments. Let's see how easy it is to write one, given our current code shape. First we extract the *TestIO* inner class from *CreateNewAppointmentTest* and make it top level, so that the new test can reuse it.
The newly created test is related to displaying appointments, so *DisplayAppointmentsTest* seems like an appropriate name. We're only going to test the no appointments scenario, so we'll write a single test: *@Test no_appointments*.

```
public class DisplayAppointmentsTest {

   private TestIO io;

   @BeforeEach
   public void setUp() {
      io = new TestIO();
   }

   @Test
   public void no_appointments() {
      var writer = Appointments.display
            .apply(ListView.listFormat)
            .apply(singletonList("header"), Stream::empty);

      writer.accept(io);

      assertTrue(io.printBuffer.contains("No appointments found\n"));
   }

}
```

It was indeed easy to write, wouldn't you agree? If you don't, think about this: we wrote a test for existing code.

Usually, when we have to do this, we find ourselves mocking many dependencies of the class under test and diving deep into their internals to mock specific behaviours (which is a terrible practice). If you think to yourself "well, this is just a toy project, the code is not nearly as vast as you'd find in an enterprise project", you would be looking at the wrong aspect of code: quantity. You should be looking at code shape instead. Who decided that you can't have this shape of code, but in a very large quantity?
I'd like to think that I've given you some tools with which you can reshape the code into something easier to change, maintain and test.

This concludes the code reshaping journey we've been on for a while now. I'd like to show you next how easy it is to take otherwise hard decisions, just because we've applied these techniques to our project.

Add an HTTP interface

We talked about how last century it was to have a console based application nowadays. Well, what's stopping us from slapping an HTTP presentation layer on top of our system? Turns out not much. We can simply write the new presentation layer and hook it directly into our system, without disrupting too much of anything, you'll see.
Let's get on with it, shall we? For adding HTTP support to our application, we need a library that can do all protocol related stuff and hands us some hooks to do application specific things. We find a small, neat library that does exactly this: sparkjava (you can find it at sparkjava.com). Add it to our dependencies

```
dependencies {
    ...
    implementation('com.sparkjava:spark-core:2.9.2')
}
```

and we're already on our way to a merry HTTP UI. The library has a nice starting-up documentation page which we can use to get up to speed with the API and we're ready to write our 2 endpoints, one for creating new appointments and another for reading existing ones. I'm fairly sure that if you've done web development before, you'll have absolutely no trouble following along. A new entry point is in order, so we'll just call it *HttpApplication* in lack of a better term.

```
public class HttpApplication {

    public static void startHttpEndpoints() {
        get("/appointments", (req, res) -> {return "test";});
        get("/appointment", (req, res) -> {return "test";});
    }

}
```

I chose to implement adding as a GET operation, to make it easy on myself to pass parameters along (a POST would have forced me to create an HTML form and I don't need that to illustrate my point). This is a simple mapping of a path to a function. It's all we need really, no fancy filtering or routing for this proof of concept. Spark will run on port 4567 by default, so let's test that by calling this side-effect from our *Application* namespace:

```
public class Application {

    static { HttpApplication.startHttpEndpoints(); }
    ...
}
```

That's right. We simply throw it inside a static block and have it run on *Application* load. It's perfect for now. We fire up the application and open a browser at http://localhost:4567/appointments. Sure enough, we get a page containing the text "test". We now have two parallel applications running, with no real connection between them other than the fact that they run inside the same JVM. We'll keep it like this for a while, until we build an HTML view and an HTTP specific I/O.
Write the test first, like we did before for new things and we end up with a test that looks like this:

```
public class HtmlTableViewTest {
  @Test
  public void display() {
    var expected = "<table><th>text</th><tr><td>data</td></tr></table>";

    var actual = HtmlTableView.htmlTableFormat.apply(
          Collections.singletonList("text"),
          () -> Stream.of(Collections.singletonList("data")));

    assertEquals(expected, actual);
  }
}
```

We invented the namespace *HtmlTableView* and function *htmlTableFormat*, because we know this is exactly what we need. We're going to create them now and run the TDD cycle until we finish the implementation:

```
public class HtmlTableView {

    public static View htmlTableFormat = (headers, content) ->
          String.format("<table>%s</table>",
                header(headers).append(data(content)).toString());

    private static StringBuilder data(
                                  Supplier<Stream<Collection<String>>> content) {
        StringBuilder sb = new StringBuilder();
        content.get().forEach(row -> {
            sb.append("<tr>");
            row.forEach(item -> sb.append(String.format("<td>%s</td>", item)));
            sb.append("</tr>");
        });

        return sb;
    }

    private static StringBuilder header(Collection<String> hs) {
        StringBuilder sb = new StringBuilder();
        hs.forEach(d -> sb.append(String.format("<th>%s</th>", d)));

        return sb;
    }
}
```

Same concept as the other two views we already have. Very straight forward to implement. We have one task down, now we move on to the second one: custom HTTP I/O. I'll

write it and go through it step by step, so we make sure it's clear:

```
public class HttpApplication {

    public static void startHttpEndpoints() {
        get("/appointments", (req, res) -> {
            StringBuilder sb = new StringBuilder();
            display.apply(htmlTableFormat).apply(HEADERS,
                                                    content).accept(httpIo(req, sb));
            return sb.toString();
        });
        get("/appointment", (req, res) -> {
            StringBuilder sb = new StringBuilder();
            addNew.apply(save).accept(httpIo(req, sb));
            return "created by interaction flow:<br/>"+sb.toString();
        });
    }

    private static TypedIO httpIo(Request req, StringBuilder sb) {
        var params = Optional.ofNullable(req.queryString())
                .map(qs -> Arrays.stream(qs.split("&"))
                        .map(p -> p.substring(p.indexOf("=") + 1))
                        .collect(toList())
                        .iterator());
        return new TypedIO() {
            @Override
            public void print(Object o) { sb.append(o); }

            @Override
            public void printLine(Object o) { print(o + "<br/>"); }

            @Override
            public Optional<String> readString(String prompt, String errorMessage) {
                return params.map(p -> {
                    var val = p.next();
                    print(prompt+val+"<br/>"); // not necessary, but added to show I/O
                                               // interaction behind scenes
                    return val;
                });
            }
        };
    }
}
```

The *display.apply(format)* and *addNew.apply(save)* are already familiar, as we've used them exactly like this for the console application. We notice a custom HTTP I/O

implementation which does nothing more than parsing request parameters from the query string and leveraging a *StringBuilder* for passing the information back to the presentation layer. I'll talk about each one of those things next.

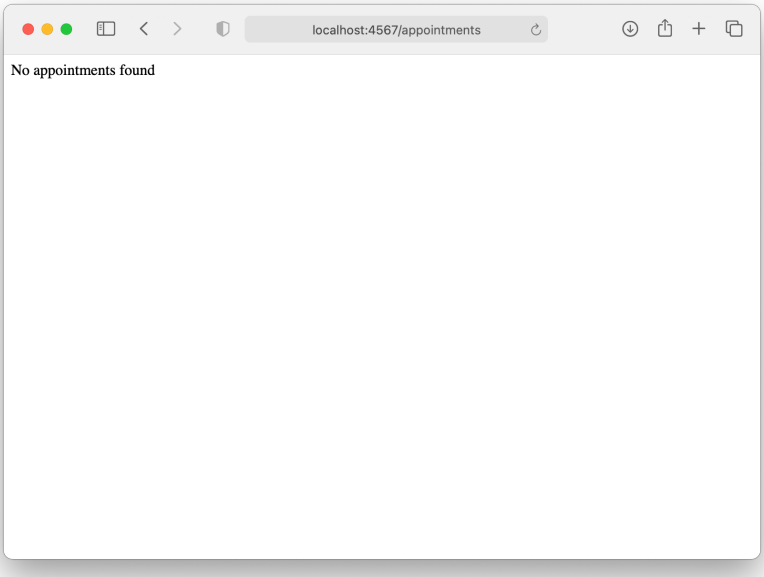
First, we deal with the reading of existing appointments, using the "/appointments" endpoint.
The *httpIo* function takes a new request and a new *StringBuilder* every time that endpoint is called and always returns a new *TypedIO*. The returned *TypedIO* uses the *StringBuilder* as writing medium, by overriding the printing methods, so when the consumer returned by *apply(HEADERS, content)* accepts this I/O object, it will populate the *StringBuilder* with the appointments in the *htmlTableFormat* shape. That was a mouthful, I know, but go back and forth between the code and this paragraph and it will make sense.

Next, we look at adding a new appointment, using the other endpoint "/appointment". This is bit trickier to get in the first go, because I used a convention, as a shortcut to get this done faster (it is a POC after all). The convention is that I know the order in which the appointment fields are read so I chose to send them in the same order in the query string. I parse them and save them in an *Iterator* variable. I override *IO*'s *readString*, to pick the next parameter on each call and that's it. Also, I added a bit of extra logging in the UI, to actually see the operations happening behind the curtain.

I know it's not what you'd write in production (I hope), but it's a good exercise to see which trade-offs I made to get the

POC going as easy and fast as possible. This way I can collect useful feedback from business and maybe even users as early on in the process, to avoid a possibly bad investment, in case they don't like it.

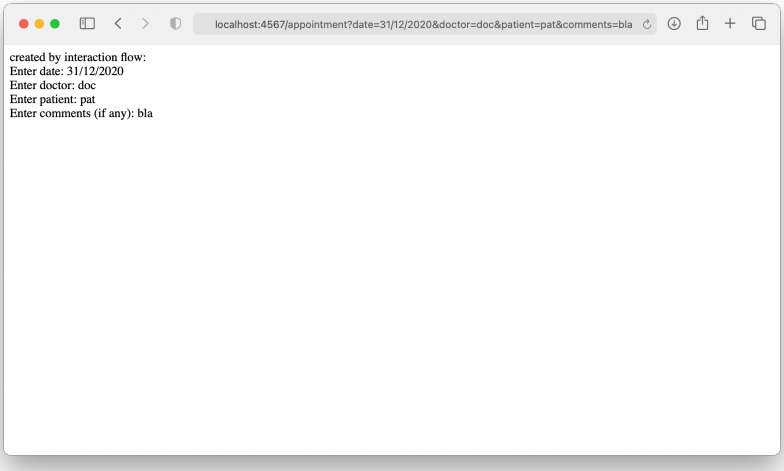
Let's fire it up and give it a try. We launch the *Application*'s main function, open up a browser and type this into the URL bar: http://localhost:4567/appointments



Nice! This is exactly what we would have expected to see when we have no appointments. Let's see if the running console confirms this.

```
Menu
l - list view
t - tabular view
a - add new appointment
x - exit
l
No appointments found
```

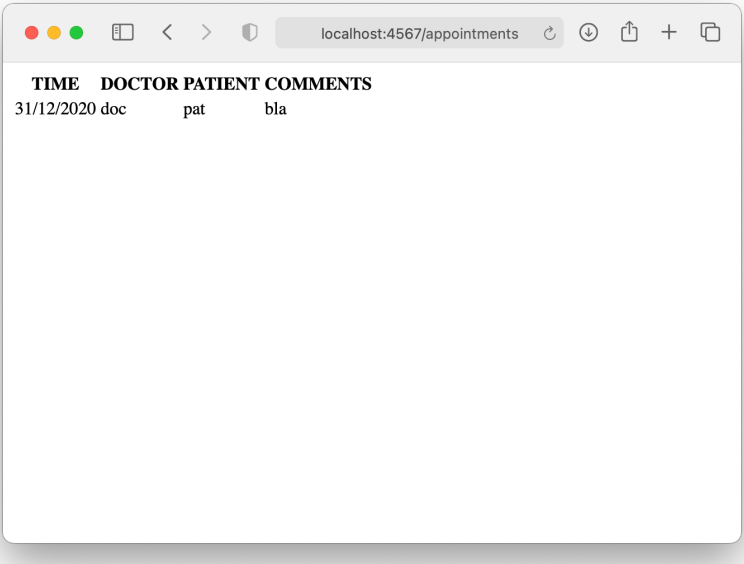
Indeed it does. But the fun part is seeing effects of interaction with one UI in the other. We're going to test adding an appointment through the HTTP UI:



We can see the extra logging we added. Before we check the appointments in the browser, let's check the console.

```
Menu
l - list view
t - tabular view
a - add new appointment
x - exit
t
-------------------------------------------------------------------------------------------------
|        TIME        |        DOCTOR        |        PATIENT        |        COMMENTS        |
-------------------------------------------------------------------------------------------------
      31/12/2020              doc                   pat                  bla
-------------------------------------------------------------------------------------------------
```

Whoa! It's there! Even if it's expected, it's a nice feeling of accomplishment. Check the display in the browser now:



It works as expected. We can test the other way around too and see that it works. One aspect to consider when adding another "terminal" to a system is concurrency. What happens when more than one user puts or reads data from our datastore? I don't know, but we should surely add good system tests to cover that. Meanwhile, we know that our datastore is just an *ArrayList*. Since we know we might get some concurrency during the business pitch (a.k.a. demo), we can just protect it with a synchronisation wrapper:

```
private static List<Appointment> DB =
                        Collections.synchronizedList(new ArrayList<>());
```

One final refactoring before the demo. Currently we have an *Application* and an *HttpApplication*. It's a bit strange, considering the console is not mentioned anywhere in the names. We'll fix that by moving the console code in *ConsoleApplication* and keeping *Application* simply for starting up the other two.

```
public class ConsoleApplication {
    ...
    private static boolean loop = true;
    ...

    private static final Map<Character, Consumer<? super Console>>
            CHOICE_TO_FUNCTION = new HashMap<>() {
        {
            ...
        }
    };

    public static void loop() {
        while (loop)
            CONSOLE.choice(MENU)
                    .map(CHOICE_TO_FUNCTION::get)
                    .ifPresent(f -> f.accept(CONSOLE));
    }
}
```

I also promised that by the end of the book, I'll fix the recursion that will haunt us when scaling, so here we go, it's fixed.

The *Application* class is doing nothing other than starting *ConsoleApplication* and *HttpApplication*:

```
public class Application {
    public static void main(String[] args) {
        HttpApplication.startHttpEndpoints();
        ConsoleApplication.loop();
    }
}
```

We have now completed the POC which is ready to take for a spin with the business. How difficult was this POC to write? Think about it. Imagine you would have had to add an HTTP interface to the original code. How long would that have taken?

Conclusions

I have shown here that confronting an ugly code base is not something to be scared of. It can become something beautiful, if you have the patience and most importantly the dedication for it.

I love good engineering and good engineering means simple solutions that work great. Please don't understand this as "quick hacks for the win". Quick hacks have their own place in engineering, but they should not be the first thing on the menu. What I really hope to have achieved with this book is to have you consider healing old, sick systems when you encounter them, while still being able to serve business requests to a high standard.
I know it's easy and tempting to say "this legacy code is too dirty and is not worth fixing", but this must not be a lightly spoken statement. It needs to be backed up by proper analysis. If the techniques shown in this book work for your system (and in most cases I've encountered they do - hence the book), then you must be fair and do your best to heal it.

The analogy to medicine, in the book's title, is not accidental. It's actually a very good metaphor, describing a doctor-patient relationship, you being the doctor and the system being the patient. You would like your doctors to do all that's possible to salvage your health (and your life if necessary), before they declare you "too dirty and not worth fixing". Some might say that the stakes are not the same and that could be true, but which is higher? Imagine software that flies hundreds of people in the sky. Are the stakes high?
I say good engineers appreciate a tough challenge and encountering a dirty code base is indeed such a challenge. Whenever I get to see new projects with legacy code, I don't get discouraged, but rather feel motivated. "Bring it on!" I say.

The biggest problem of software systems, by far, is over engineering. Just like you were probably amazed by the whole replication and backup system for the console application, the same goes for source code. If you think about it, what was the amazement factor for the two concepts you encountered in the appointments app: infrastructure and code? The code was not so surprising, right? Why? Because this is probably what you have to deal with on a daily basis and anything that becomes familiar also becomes the norm.

I know I have touched some sensitive subjects, one of which is OOP vs. FP. I do not want to make a case of one versus the other, because that is not the point. The point is to make your program as simple and as clean as possible. It so happens that today's programming functions are simpler than today's programming objects and hence make for a

better choice when writing clean code. You might disagree and that's fine.

Tests. Ah, the good old tests and the never-ending flamewars about unit-integration-system-and-what-have-you tests. What is a unit? A unit is a class. No, a unit is a method. No, a unit is a module. As long as the tester returns your feature back to you with a null pointer or reference exception, it's all dust in the wind.
I remember it was ten years ago, I was in Amsterdam's airport, Schiphol, waiting to board a flight to Montreal. We started to grow anxious, as the boarding deadline had passed, but the gates were still closed. Then we heard this on the PA:
"Ladies and gentlemen, there was a technical failure at one of the plane's parts and we needed to replace it. The technical crew replaced the part and now they are running diagnostics. We'll start boarding as soon as possible. Thank you for your patience!"
This was actually the first time in my life I really felt protected by a suite of tests. The story has a happy ending, obviously, as I have safely landed in Montreal and lived to tell the story.
What makes a good suite of tests? First and foremost, a good suite of tests guards against defects sneaking into your system. Second, the feedback should come as fast as possible. I dare say that all other properties of the test suites derive from these two. Whatever level the tests are written for, they should treat the system under test as a black box and only verify the effects of exercising it, not the insides of it.
Test coverage metrics are a lie. Controversial? Maybe. True? Definitely! Let me explain. Let's say you have a program

composed of 100 classes. Each of those classes has 5 fields. Some are composed of primitive fields and some have other classes as fields too. Pretty small, for your neighbourhood friendly production system.

Now a class is really a product type. What does that mean? It means that a class *A* with two fields, a *String* and an *Integer* can have a number of valid instances representing the Cartesian product of those two primitive types: *String* x *Integer*.

Test coverage metrics usually count the lines of production code that were exercised during the test run, not all the possible combinations of all those object values in the system. You see now how the hundred classes system can easily become impossible to really cover properly. What usually happens is you get the happy flow values and a few corner cases covered. If you exercised all the production code paths, the metrics will give you 100% coverage. Lies! We have property based tests, that do a better job at covering values, but even they are no match for all possible values, as that would be totally unpractical with today's computing power. Long story short, please don't obsess over test coverage metrics as it's surely not worth spending too much energy on.

So what should you test for then? You should test the things your system was specifically designed for. Anticipate and plan for a few sensible user mistakes and beyond that, if the user drives the car off a cliff...

"If you design your tests properly, breaking one tiny part in the production code should break an equally tiny part in the tests". Is this true? It should be true, for unit tests (tests that exercise small, independent parts of the system). It should not be true for testing the system as a whole. Why not? Imagine you break the code that connects to your

main database. Most of the system should be dead at this point. So yes, different strategies for different testing angles.
I care a lot about testing (as you might have noticed) and maybe this will make a good subject for a next book, but for now, I'm going to leave it at this.

Refactoring is about changing the shape of code, not its runtime outcome. Be careful how you use this term, especially when talking to business people. By now, they are aware of the term and they've come to hate it. I tend to agree. When you ask for something apparently trivial, but you get the "we need a month of refactoring" response, over and over, you're kind of entitled to feel angry. You've seen how we refactored just enough to fit the new requirement in and we've discussed how we should triangulate decisions for every feature. I've been asked countless times before how a programmer should "sell" refactoring to business. It's actually not hard at all. Engineers know how to estimate change costs, based on the status quo of their system.
Programmers are software engineers and they should be able to do the same. One should simply say how much it costs (time is fine, business knows how to express that in money) and present the reasons in terms the business can understand. For example, you don't want to say things like "this will take longer, because this *HashMap* is not synchronised and when two threads...", because you'll be speaking martian for most business people. Instead, what you should be saying is "for this, we need some time to adapt the system for allowing multiple users to access this feature at the same time, otherwise, they'll get errors and complain". Now that, they can very much understand and

judge the impact of. Don't overestimate to hide refactoring time, instead earn it by properly "selling" it.

The book presents empirical methods for source code transformations. This places it very much into the engineering realm, rather than the theoretical one. I do however, strongly believe, that we could enrich programming techniques with more theoretical concepts. This is being done successfully in lots of other disciplines. We should leverage mathematics more, because it has way better computational mechanisms, that we've been successfully using for millennia.
What we keep doing in programming is constantly reinventing the wheel, by implementing ad-hoc rules. What we should be doing is map the domain to mathematical concepts (e.g., algebraic notions like groups, semigroups, categories, etc.) and leverage that algebra to do the computations for us.
The Haskell community is trying to achieve this goal and their endeavour has echoed in other communities as well. This has generated a plethora of libraries in many languages, but I don't think the industry is yet prepared for a major change like this (even though this started some decades ago already).

I'm not going to make any predictions of the direction the industry is going to take, because I'm almost certainly going to be wrong, but nevertheless, I do think that we can only benefit by reuniting programming with math.

The source code

The appointments project is publicly hosted on GitHub at:

https://github.com/dannicolici/appointments

The chapters in the book are mostly driven by the commit history. Sometimes they might be ahead and sometimes they might leave some small things out, like minor refactoring work. Overall, it pretty much goes hand in hand.

If you've made it this far, I'd like to thank you and wish you all the best in your programming journey!

About the Author

Born in 1981, in Baia Mare, Romania, I am an IT industry professional, with a passion for computers and computer programming since early childhood. I built many software systems over the years, covering a wide range of business domains, such as telecom, finance, sports, etc. My passion extends beyond computers to good engineering and science in general. I also play electric guitar, as a hobby.

You can reach me at dan.nicolici@bitgloss.ro