Mateusz Kupilas

Junior Developer

Wszystko, co trzeba wiedzieć,

by zacząć pracę jako programista

/*Autor bloga JavaDevMatt.pl*/

Mateusz Kupilas
Kupilas Media Mateusz Kupilas
42-600 Tarnowskie Góry, ul. Kaczyniec 12

Projekt okładki:
Visualse.com

Ilustracje:
Openclipart.org

Redakcja i korekta:
Joanna Fiuk

Skład i łamanie:
Marcin Jastrzębowski

ISBN 978-83-947287-0-0

Wydanie I, Tarnowskie Góry 2017

Druk i oprawa:
Print Management
Wojciech Pietrzak
wojtek@printmwp.pl
www.printmwp.pl

Kupując tę książkę, wspierasz finansowo bloga i kanał na YouTube: JavaDevMatt.

Dziękuję. :)

Piratom ściągającym książkę z torrentów lub innych pebów i chomików również dziękuję za wykazane zainteresowanie. Też kiedyś piraciłem i nie mam zamiaru udawać teraz świętoszka. Może będzie kiedyś okazja wypić wspólnie beczkę rumu. Arrrr!

Nikogo za piractwo tej książki ścigać nie zamierzam. Cieszę się z tego, że jest na tyle popularna, by warto było ją kopiować.

Spis treści

1.

Sponsorzy

Niniejsza książka (początkowo wydana tylko w formie e-booka) powstała dzięki akcji na portalu PolakPotrafi.pl: **JuniorDeveloper.pl/pp**. Łącznie projekt wsparło 425 osób. Oto lista tych, którzy wpłacili przynajmniej 75 zł i trafili na listę sponsorów:

- Łukasz „echoman" K.
- Patryk Dolata – 365dw.pl
- Michał Franc – mfranc.com
- Dominik Minta
- Piotr Stefański
- Dominik Kubis
- mlody-inzynier.pl – oferty pracy dla młodych inżynierów

- Organizer leków na Androida – Mobilna Apteczka
- Jan Jakub Jurec
- Marek Pawłowski
- Krzysztof Podkanowicz
- Damian Kolanek „Kolan”
- Paweł Sawicz – pawel.sawicz.eu

2.

Partnerzy projektu

Z magazynem „Programista" współpracuję od stycznia 2015. Ma on duży wkład w promocję mojej osoby i tej książki.

programista

W Szkole Programowania Coders Lab wiedzą, że rozpoczęcie pracy w nowej branży to spore wyzwanie. Co miesiąc pomagają setkom osób odnaleźć swoją ścieżkę w IT. Dlatego też zdecydowali się wesprzeć projekt i promować moją książkę. Absolwenci szkoły, których poznałem, wywarli na mnie pozytywne wrażenie.

Coders Lab
SZKOŁA PROGRAMOWANIA

Radio internetowe KonteStacja w dużym stopniu ukształtowało moją przedsiębiorczą naturę, dzięki której zacząłem blogować i napisałem tę książkę. Szczególny wpływ wywarły na mnie audycje „Wydania Głównego" z lat 2010--2012 i audycje Janka Fijora. Posiadanie własnego podcastu w tym radiu również pomogło mi dotrzeć do szerszego grona odbiorców.

WWW.KONTESTACJA.COM
RADIO INTERNETOWE

3.

O czym i dla kogo jest ta książka?

Sporo osób śledzących mojego bloga Programista na Emigracji (aktualnie Programista po Emigracji) ma dylematy i pytania podobne do tych poniżej:

- Interesuję się programowaniem, ale nie wiem, czy wiązać z nim przyszłość zawodową.

- Jeśli zdecyduję się wykonywać ten zawód, to czy muszę iść na studia?

- Jaki poziom umiejętności muszę posiąść, by starać się o pierwszą pracę jako programista?

- Jeśli się uczyć, to z czego? Jak się uczyć, by nauka była skuteczna? Jest przecież sporo pułapek: poświęcamy na coś czas, a efekty są kiepskie.

- Czego mogę spodziewać się w pierwszej pracy jako programista? Jak wygląda typowy dzień i organizacja pracy? Skoro raz napisany program „już działa", to po

co są zespoły ludzi, którzy pracują nad utrzymaniem danego oprogramowania?

- A co, jeśli jestem kiepski z matematyki?

- Kiedy powinienem się zorientować, że to zajęcie nie dla mnie?

- Co z zarobkami? Tego tematu nie rozwinąłem w książce. W skrócie: każdy specjalista w danej dziedzinie może dobrze zarabiać. Nie uważam, aby wysiłek, który należy włożyć w pracę programisty, był większy/mniejszy niż w innych branżach. **Podążanie w tym kierunku czy jakimkolwiek innym wyłącznie dla pieniędzy to – moim zdaniem – strata czasu i marnowanie sobie życia.**

- Jak wygląda rekrutacja? Jak się do niej przygotować, czego oczekiwać od pracodawcy, co umieścić w CV itp.

- Kiedy mogę pracować w tym zawodzie zdalnie lub jako wolny strzelec mieszkający na Bali?

- Z jakimi przykładowymi zadaniami/problemami spotkam się w pracy?

- Na czym polega nauka programowania, gdy chce się zostać programistą gier? Jak wygląda praca w tej branży? Jak bardzo różni się od innych dziedzin programowania?

- Jakie inne stanowiska (oprócz developera/programisty) mogą mnie zainteresować w tej branży?

- Czym zajmuje się developer, project manager, scrum master?

- Czym jest scrum/agile? Jak to wygląda w praktyce?

Na wszystkie te pytania postaram się odpowiedzieć w tej książce czy też e-booku (jeśli wydrukujesz e-booka i posklejasz kartki, to będzie już jak książka :)).

Jeśli zadajesz sobie pytania podobne do tych, które wymieniłem, to zapraszam do dalszej lektury. Jeśli jednak stwierdzisz, że to nie dla Ciebie, to w ramach poszukiwania swojego „ja" i inspiracji do tego, co ciekawego można robić w życiu, zachęcam do odwiedzenia radia internetowego KonteStacja.com lub Enklawa.net. W końcu w życiu nie chodzi o to, by było łatwo i wygodnie, tylko ciekawie i satysfakcjonująco.

Czytając tę książkę, pamiętaj, że są to moje prywatne opinie, wnioski i przemyślenia. Sporo Czytelników może (i powinno) nie zgodzić z częścią moich poglądów. Chętnie zapoznam się z uwagami/krytyką/opiniami. Wysyłajcie je na matt@javadevmatt.pl.

Miłej lektury!

4.

Skąd pomysł na napisanie książki?

W lipcu 2014 r. nagrałem swój pierwszy filmik na kanał na YT. Był to prosty tutorial o libGDX: bibliotece do multiplatformowego tworzenia gier w Javie (i nie tylko). Ostatecznie na moim kanale JavaDevMatt ukazało się 10 filmików z serii „LibGDX dla Zielonych". Po wypromowaniu tej krótkiej serii tutoriali na Wykop.pl, Warsztat.gd i w paru innych miejscach w sieci zaczęły się pojawiać pytania o to, gdzie pracuję… Gdy zdradziłem, że w niemieckim start-upie tworzącym aplikacje mobilne, posypały się kolejne pytania:

- „Jak wyglądała rekrutacja?"
- „Co zrobić, by kiedyś pracować w podobnym miejscu?"
- „Czy zarabiasz 15k? ;)"

Nagrałem więc filmik o tym, jak przebiegała rekrutacja. Materiał zdobył dużą popularność: dwa dni po dodaniu miał więcej wyświetleń niż cała seria o libGDX razem wzięta.

I tak zacząłem nagrywać materiały, a później również pisać bloga Programista na Emigracji.

Po prawie półtora roku prowadzenia bloga JavaDevMatt.pl oraz związanego z nim kanału YT okazało się, że jest parę pytań, które wciąż się powtarzają. Na wiele z nich odpowiadam w filmach lub we wpisach na blogu, ale potrzebne było jakieś miejsce, w którym ta wiedza byłaby zebrana w całość i logicznie uporządkowana. Książka (początkowo tylko e-book) wydała się dobrym rozwiązaniem. Czytelnik dostaje solidny produkt, w którym znajdzie wyczerpujące wyjaśnienie nurtujących go kwestii, a ja zyskuję sposób finansowania mojego bloga, by nie musieć zaśmiecać go wyskakującymi okienkami z reklamami. Win–win: korzystają wszystkie strony transakcji.

Pierwsze logo serii Programista na Emigracji

Jeśli pobrałeś PDF-a tej książki z innego źródła, to przynajmniej polub mojego bloga na FB i subskrybuj kanał na YouTube. Uważam, że walka z piractwem to walka z wiatrakami – i nie warto tracić na nią czasu. Zresztą sam nie jestem święty w tej kwestii, a nie lubię hipokryzji.
Bawcie się dobrze.

4.1. Co nowego w tym wydaniu?

To, co czytasz, to pierwsze wydanie papierowe książki, ale drugie wydanie e-booka „Junior developer" – e-book pojawił ponad pół roku przed książką, dzięki czemu miałem więcej czasu na uzupełnienie treści. Jakie nowości znalazły się w tym wydaniu?

- Informacje na temat tego, kim jest senior developer.
- Autentyczne pierwsze zadania programistów (podesłane przez czytelników mojego bloga).
- Dlaczego nie polecam zaczynać pracy w zawodzie programisty od bycia wolnym strzelcem.
- Informacje o impostor syndrom (syndrom oszusta), przez który często niesłusznie wątpimy w siebie.
- Poruszyłem problem siedzenia w jaskini, czyli wiecznego kupowania książek zamiast rzeczywistej nauki.
- Więcej o pracy z bugami.
- Od jakiego języka zacząć?
- Inne drobne poprawki i dodatkowe akapity o tym, co warto znać (np. narzędzia do budowania projektu).

5.

Z jakich powodów nie warto zostać programistą?

Zanim zaczniesz poświęcać na coś czas, zastanów się, czy to jest faktycznie to, co chcesz robić. Programista jest obecnie bardzo popularnym zawodem w mediach („dobrze zarobisz/zawsze znajdziesz pracę/masz ciepły stołek w biurze"), ale w życiu można robić tyle innych rzeczy, że nie warto rzucać się na to, co akurat polecają media, znajomi czy społeczeństwo.

Poniżej wymieniłem dwa powody, które nie powinny Cię skłaniać do wyboru ścieżki zawodowej programisty. Trzeci powód jednych odwiedzie od tej pracy, a dla innych będzie zachętą.

5.1. Dla pieniędzy

Pieniądze – obecnie najczęstszy argument przemawiający za wyborem tego zawodu i pierwszy temat pojawiający się w rozmowach o pracy programisty. Polska jest w tym

wypadku ewenementem, ponieważ faktycznie na niektórych szczeblach kariery zarabia się trochę lepiej niż w wielu innych branżach, które wymagają podobnego poziomu zaangażowania. Wynika to głównie z możliwości pracy zdalnej dla zagranicznych klientów czy dużego zapotrzebowania na programistów, dzięki któremu tacy pracownicy bardzo chętnie są podbierani. Jednak na świecie programista, tak jak każdy inny specjalista, **może** zarabiać świetnie, przeciętnie albo po prostu mało (jak na średnią w branży) – wszystko zależy od tego, jak dobry jest w tym, co robi (i jak dobrze negocjuje). Inżynier budownictwa, prawnik, dentysta, doradca podatkowy – jeśli jest dobry w swoim fachu, nie narzeka na brak klientów. Może pozwolić sobie na podniesienie stawki i nadal mieć masę roboty.

Bez lania wody: zostań specjalistą w dziedzinie, która sprawia Ci frajdę albo jest Twoją pasją. Bądź specjalistą programistą, jeśli jest to coś, co chcesz robić, a nie dlatego, że rodzina Cię do tego namawia i pcha na takie studia. Warto czasami poświęcić dwa-trzy lata na szukanie swojego zawodu, próbowanie różnych zajęć, niż obudzić się w wieku 50 lat i stwierdzić, że praca nigdy nie sprawiała nam przyjemności i zmarnowaliśmy kawał życia.

5.2. Bo to „łatwa praca w biurze"

Z tą łatwością bywa różnie. Poziom stresu zależy od tego, w jakiej firmie się pracuje (inaczej jest w małych firmach, w których pracownik jest od wszystkiego, inaczej w średniej wielkości start-upie, jeszcze inaczej w korporacji). Mo-

żesz być jednak pewny, że zdarzy się sytuacja, w której praca nie będzie miała nic wspólnego z przyjemnością: czy to przez niemiłego klienta, zbliżający się deadline, czy po prostu przez mnogość monotonnych i nieciekawych zadań.

Dotyczy to szczególnie pracy w branży gier komputerowych. Idź w tym kierunku tylko wtedy, gdy jest to Twoja pasja albo chcesz nabrać doświadczenia, które później wykorzystasz we własnych projektach. Gamedev to bardzo często nadgodziny, które nie są dodatkowo płatne. Sam nigdy nie pracowałem w firmie tworzącej gry komputerowe, ale w rozmowach z kolegami, którzy próbowali swoich sił w grach, zawsze przewijał się temat nieodpłatnych nadgodzin. Nie zakładaj z góry, że wszyscy stosują takie praktyki (pewnie są przyjemniejsze firmy), **ale na pewno jest to częste zjawisko**.

Jak wygląda praca w branżach „pozagrowych"? Na początku pracy nad projektem zwykle obserwuje się rozluźnienie. Następnie do pierwotnych założeń wprowadzane są zmiany (problematyczny klient, który zmienia zdanie albo nie zdawał sobie sprawy z tego, czego chce – rozwiązaniem tego problemu może być agile, o którym trochę później), pojawia się kłopot z wybraną do projektu technologią (okazuje się, że coś nie jest możliwe do zaimplementowania i należy sporo przebudować, a to wymaga paru tygodni pracy, których nie mamy) i zaczynają się schody. Wówczas cała nadzieja w dobrej kadrze zarządzającej, która rozwiąże problem w inny sposób niż przez zarządzenie nadgodzin dla zespołu. Bywa, że nie mamy na to wpływu. Słowem, nie zawsze jest różowo.

Trzeba jednak przyznać, że ze względu na częste podbieranie sobie programistów przez różne firmy zazwyczaj o pracownika w tej branży się dba. Nie jest to „łatwa praca w biurze", ale jeśli pokażesz, że jesteś coś wart i że konsekwentnie się rozwijasz, to prawdopodobnie zostaniesz doceniony.

5.3. Konieczność ciągłego rozwoju i śledzenia nowości

Ten punkt dla jednych będzie ogromną zaletą, dla innych – największą wadą. Jeśli należysz do tej drugiej grupy, zastanów się, czy nie wolisz jednak pracować w branży, w której nie pojawia się tyle nowości, ile w przypadku programowania. W wielu dziedzinach stały rozwój jest niezbędny, ale w tym wypadku trzeba się liczyć z ogromnym postępem technologicznym.

Dla kogo będzie to zaleta? Istnieje grupa osób, dla których ciągły rozwój technologiczny jest fascynujący. Tacy ludzie męczą się w monotonnej pracy. Żeby być na bieżąco, wystarczy znaleźć np. newsletter (w przypadku programisty Android świetnym źródłem informacji jest Android Weekly), który kumuluje nowości z danej branży. Nie trzeba czytać wszystkiego od deski do deski, ale zapoznanie się z ciekawym artykułem czy napisanie od czasu do czasu prostego programu to cenne doświadczenie. Ważna jest regularność, a nie intensywność. Lepiej w każdym tygodniu poświęcić godzinę, dwie, a nie co dwa miesiące cały weekend.

Dla innych konieczność stałego rozwoju będzie wadą. Oni nie mają ochoty ciągle się uczyć (albo przynajmniej nie tak intensywnie) i szukają czegoś, co mogą robić zawodowo za rozsądną stawkę. Resztę czasu chcą natomiast przeznaczyć na inne hobby albo dla rodziny. Oczywiście da się znaleźć złoty środek. Nie musisz poświęcać wszystkiego i żyć tylko programowaniem: ważna jest ciekawość świata technologicznego, w którym na co dzień się poruszasz. Jeśli nie lubisz/nie potrafisz tego regularnie robić, to powinna Ci się zapalić czerwona lampka.

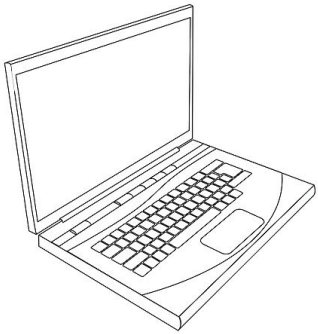
Osoby mające telefon z Androidem (a właściwie wszyscy właściciele smartfonów) pewnie pamiętają telefony z roku 2010 – jak bardzo różnią się od tych z 2015 r. Pięć lat to cała epoka. Ktoś, kto bez problemu dostał pracę jako Android developer w roku 2010 i od tego czasu w ogóle się nie rozwijał, w roku 2015 miałby problem. By utrzymać najlepszych klientów/dostać awans w pracy, trzeba być na bieżąco z technologią. Naturalnie w granicach rozsądku: wystarczy po prostu się nią interesować.

5.4. A dlaczego warto?

Największy plus, jaki widzę, to **łatwy dostęp do materiałów do nauki**. Ich ilość może być wręcz problemem (sporo teorii i praktycznych projektów z otwartym kodem źródłowym jest na wyciągnięcie ręki) i łatwo wpaść w spiralę uczenia się nie tego, co rzeczywiście potrzebne (o tym, jak i z czego się uczyć, w dalszej części).

Trudno mi wyobrazić sobie drugą taką branżę, w której można równie łatwo stworzyć solidne punkty do CV. Nie potrzebujesz drogich maszyn, licencji ani specjalistycznego oprogramowania. Oczywiście istnieją płatne środowiska programistyczne i specjalistyczne narzędzia, ale za pomocą darmowych narzędzi można zdziałać TAK DUŻO jak w żadnej innej branży. Nie jest potrzebny ani klient, ani materiały. Wystarczy najprostszy komputer, trochę prądu i internet (chociaż można i bez internetu) – to całe wyposażenie niezbędne do tego, by się rozwijać. I najważniejsze… trzeba mieć chęci.

Fajni ludzie. Serio. Jeśli pasuje Ci ta praca, to poznasz mnóstwo interesujących i wartościowych ludzi. Z moich obserwacji wynika, że wśród programistów jest wyjątkowo duży odsetek osób z wszelkimi pasjami i ambicjami życiowymi. Być może wynika to z faktu, że jest to praca dla ciekawych świata i chcących się rozwijać – nie tylko w zawodzie, ale również w innych dziedzinach.

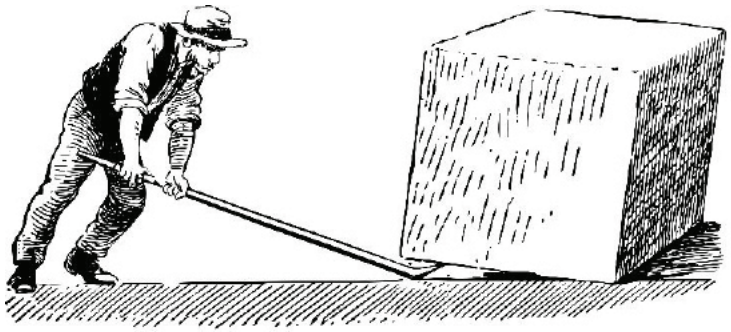


Mój główny powód: ogrom możliwości usamodzielnienia się. Po paru latach doświadczenia i pracy nad sobą zaczyna otwierać się wiele dróg, o które trudno w innych zawodach: praca zdalna dla bogatego klienta (gdy mieszka się w kraju z niskimi kosztami życia) czy stosunkowo łatwe tworzenie własnych produktów (nie ma ogromnych kosztów materialnych, jak w innych branżach). Przy okazji niskie ryzyko zawodowe. Jakaś odpowiedzialność jest, ale nie zabijemy kogoś tak łatwo jak lekarz, nie wsadzimy tak szybko do więzienia jak prawnik ani nie narazimy na karę jak doradca podatkowy. Chociaż – jeśli się postaramy – to możemy sporo na świecie namieszać – i to jest właśnie fajne!

6.

Co powinieneś potrafić, by zacząć pracę jako programista?

Jak i z jakich źródeł się uczyć, omówiłem w dalszej części. O rzeczach opisanych w tym rozdziale można pomyśleć jak o mnożniku. Zawsze będziesz w stanie wykonać pewną ilość pracy, poznać pewną ilość nowinek ze średnią prędkością pomnożoną przez A, B i C (kiedyś wyczytałem, że mądrze jest nazywać to dźwignią, więc również będę się wymądrzał i używał tego słowa).



Prędkość, z jaką się uczysz, czyli wdrażasz w projekt, przyswajasz technologie wykorzystane w projekcie czy robisz research technologii, które warto zastosować, zależy od paru punktów/dźwigni. Poniżej wymieniłem te punkty i objaśniłem, dlaczego są ważne. Co konkretnie możesz zrobić, by szlifować swoje umiejętności, znajdziesz w rozdziale „Praktyczna ścieżka rozwoju – konkrety".
Zaczynamy.

6.1. Język angielski

Jeśli chcesz robić coś więcej niż tworzenie stron, wizytówek w HTML-u i instalowanie wordpressów, musisz znać angielski. Programiści pracują z dokumentacją, która zazwyczaj jest po angielsku. Najwięcej materiałów w sieci (w książkach w sumie też; co więcej, w recenzjach często można wyczytać, że do polskich tłumaczeń wkradają się błędy rzeczowe) dostępnych jest po angielsku: czy to artykuły, filmiki na YouTube, czy materiały wideo na innych stronach (Lynda.com, Udemy.com, Codecademy.com itd.).

Angielski to zdecydowanie największa dźwignia, którą należy szlifować. Jeśli na co dzień pracujesz/uczysz się wolniej niż inni, przyczyną tego jest prawdopodobnie słaba znajomość języka angielskiego. W przypadku programowania większość czasu poświęconego na projekt zajmuje czytanie kodu, a nie jego pisanie (dotyczy to oczywiście projektów, które trwają dłużej niż prace zaliczeniowe na dwa wieczory). Tak jak mniej czytelny kod jest głównym powodem wolnych postępów w pracy, tak powolny rozwój młodego programisty wynika często ze słabej znajomości angielskiego.

Oprócz szybszej i efektywniejszej nauki angielski przynosi także inne korzyści:

- większą wartość na rynku pracy;
- lepszą współpracę w przypadku pracy zdalnej (gdy każdy pracuje w innym miejscu na świecie, cała komunikacja odbywa się po angielsku);
- możliwość pisania bloga, który dobrze wygląda w CV.

6.2. Umiejętność szukania informacji

Druga dźwignia na liście – ściśle powiązana z poprzednim punktem. Gdy szuka się rozwiązania zadań typowo szkolnych/uczelnianych, wyszukiwanie w języku polskim może wystarczyć, natomiast w tej pracy większość błędów wklejonych do wyszukiwarki zwróci wyniki w języku angielskim.

Wiedzieć, jak (i czego) szukać, to w przypadku wielu zadań już połowa sukcesu. Na umiejętność szukania składają się w 90% doświadczenie (którego nabywa się wraz z udziałem w projektach trwających dłużej niż tydzień czy miesiąc), a w 10% – proste wskazówki. Pokazałem to w rozdziale „Praktyczna ścieżka rozwoju – konkrety" na paru przykładach.

6.3. Rozbijanie zadań na mniejsze i praca nad jedną rzeczą naraz

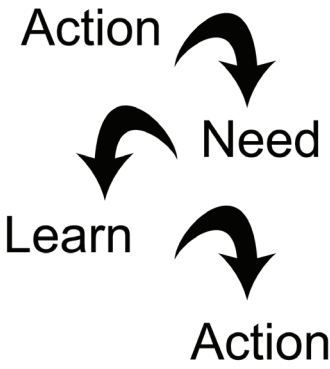
W poradnikach na temat efektywnej pracy/nauki pojawia się czasami pojęcie monkey mind. Jest to metafora oznaczająca małpkę, która siedzi w ludzkiej głowie i domaga się natychmiastowej gratyfikacji. Skutkiem ulegania małpce jest znana wszystkim prokrastynacja (przekładanie czegoś na później).

Tej małpy nie da się wygonić z umysłu, można jedynie wyrobić sobie nawyki, które ułatwią negocjacje z nią. Takim nawykiem jest rozbijanie większego zadań na kilka mniejszych, które można wykonać jedno po drugim. Mentalny obraz dużego zadania może zniechęcać. Jeśli jednak skupimy się tylko na jednym małym zadaniu, to łatwiej będzie przekonać nasz małpi umysł, by wykonał następny produktywny krok, a nie oddał się beztroskiemu przeglądaniu Reddita czy przekopiowanych z Reddita śmiesznych obrazków na Wykopie.

6.4. Napisanie (nawet bardzo małego) projektu od początku do końca

Złą praktyką w nauce programowania jest ślepe przeklepywanie tego, co pokazano w tutorialu. Pracując nad projektem, powinieneś stosować metodę just in time learning, czyli uczyć się czegoś dopiero w momencie, **gdy tego faktycznie potrzebujesz**.

Najlepszym sposobem, by zacząć tak się uczyć, jest napisanie nawet bardzo prostego projektu od początku do końca. Załóż sobie konto na GitHubie (protip: nie zakładaj konta pod własnym nazwiskiem czy oficjalnym nickiem – będziesz się czuł bardziej komfortowo, wrzucając pierwszy kod) i realizuj małe (może czasem średnie) projekty.



Dodatkowa korzyść z tego jest taka, że jeśli projekt okaże się w miarę fajny, będziesz mógł go umieścić w CV. To szczególnie cenne w przypadku braku pierwszego doświadczenia zawodowego.

Przy okazji – **nie popełniaj błędu zwanego siedzeniem w jaskini**. Jeśli masz chociaż jeden z tych objawów:

- zbierasz mnóstwo zakładek w przeglądarce z tutorialami z przeznaczeniem na później;

- kupujesz książki informatyczne, które przerabiasz bardzo pobieżnie (albo wcale);

– to prawdopodobnie dopadł Cię problem siedzenia w jaskini. To łudzenie się, że kiedyś przerobimy te wszystkie zachomikowane zakładki i książki. Zamkniemy się w jaskini na miesiąc, będziemy się uczyć 12 godzin dziennie i po miesiącu wyjdziemy z niej jako wszystkowiedzący nadczłowiek. Też miałem ten problem, ale na szczęście się go pozbyłem. To oszukiwanie samego siebie: jeśli nie jesteś w stanie wygospodarować w tygodniu 2-3 godzin na naukę, to nie zmusisz się, by kiedyś w końcu przysiąść i wszystko przerobić. O ile nie masz w piwnicy kilograma koksu, z pewnością będzie to niewykonalne dla Twojego mózgu – a nawet z koksem takie podejście na dłuższą metę się nie sprawdzi.

Siedzenie w jaskini opisałem szerzej na blogu, nagrałem również film na ten temat (link: **JuniorDeveloper.pl/jaskinia**).

6.5. Umiejętność skonfigurowania środowiska pod projekt

Na tym spędzisz pierwszy dzień (a może nawet parę dni) w pierwszej pracy… i prawdopodobnie w każdej kolejnej. Konfiguracja małych projektów, pisanych na zaliczenie, przeważnie jest prosta, natomiast projekty bardziej złożone (nad którymi zespół pracuje parę lat) sprawiają już trudności. By zbudować projekt, często potrzebne jest zapewnienie odpowiednich praw dostępu – to normalne. Nie przejmuj się, ludziom w firmie będzie zależało, byś mógł zacząć sprawnie pracować. Ktoś Ci pomoże, ewentualnie będziesz musiał parę razy poprosić.

Myślę, że początkowa konfiguracja środowiska jest jednym z głównych powodów, przez które wiele osób zniechęca się na starcie: „Jak to: 1-3 godziny ustawiałem wszystko, by wpisać w konsoli dwa słowa albo wyświetlić puste okienko?”.

Warto zatem wcześniej poćwiczyć na projektach np. z GitHuba. Z każdym samodzielnie skonfigurowanym środowi-

skiem do projektu będziesz radził sobie coraz lepiej. Zanim stanie się to intuicyjne, czeka Cię frustracja. Pamiętaj, że nie jesteś z tym sam. Wykonanie zdania często wiąże się z instalacją lokalnego serwera, założeniem lokalnej bazy danych, by zyskać środowisko testowe (nie zawsze mamy serwer testowy) itp. Poradzisz sobie.

6.6. Podstawy baz danych (zapisywanie i czytanie danych), technologii webowych, działania sieci, programowania obiektowego, pisania testów i webserwisów

Podkreślam, że chodzi o solidne **podstawy**, a nie wiedzę ekspercką. Zdziwisz się, ilu rzeczy uczyłeś się z obawy, że będą ich od Ciebie wymagać, tymczasem okazały się niepotrzebne. Dlatego ważne jest **just in time learning**, czyli zgłębianie wiedzy wtedy, gdy jest faktycznie potrzebna.

Jeśli chodzi o umiejętności z zakresu **baz danych,** powinieneś potrafić założyć nową tabelę, wykonać zapytania, by wyciągnąć z niej dane/dodać nowe dane. Na detale przyjdzie czas, gdy będziesz na co dzień pracował z bazą danych. Moja wiedza na temat baz danych była mocno podstawowa i miałem w związku z tym obawy. Okazało się, że musiałem wykonywać jedynie proste zapytania, by sprawdzić jakieś dane (współcześnie są dostępne narzędzia, które znacznie to ułatwiają i usprawniają pracę), a ważniejsza okazała się sprawna komunikacja z osobami zajmującymi się webserwisami. W dalszej części książki znajdziesz parę praktycznych wskazówek, jak nabrać pewności siebie i sprawdzić, czego trzeba się jeszcze douczyć.

Dlaczego **technologie webowe**? Nawet jeśli całe dnie zajmujesz się serwerem i tworzysz zaplecze (back-end), powinieneś też umieć przygotować lekki webowy front-end. Tym bardziej że opanowanie tej umiejętności jest proste: nie wymaga wiele pracy, a korzyści są spore.

Działanie sieci? Na początek doczytanie o HTTP/HTTPS, FTP albo POP wystarczy. Ważne, by podejść do tego od strony programistycznej i wykorzystać któryś z protokołów w prostej aplikacji (język programowania nie ma znaczenia: znasz głównie Ruby, to działaj w Ruby).

Programowanie obiektowe: jak w każdej innej dziedzinie umiejętności przyjdą wraz z doświadczeniem. Na początek powinieneś znać podstawowe pojęcia, takie jak:

- obiekt,
- klasa,
- klasa abstrakcyjna,
- interfejs,
- dziedziczenie,
- kompozycja,
- metoda,

- wzorzec projektowy („Jakie wzorce stosujecie w projekcie? Chciałbym o nich doczytać" – to dobre pytanie na pierwsze dni w pracy; nie bój się pytać!).

Webserwisy. Jeśli nigdy ich nie stosowałeś, na początek zainteresuj się, czym jest REST.

Pisanie testów jednostkowych. Powinieneś wiedzieć, po co się je pisze i jakie technologie są popularne w Twoim ulubionym języku programowania. Na przykład w przypadku Javy warto pobieżnie zorientować się, czym są JUnit i Mockito. Nie wnikaj w szczegóły – jeśli dopiero zaczynasz, tylko się zniechęcisz. Na początek musisz wiedzieć tylko tyle, że takie coś istnieje i jest przydatne w przypadku projektów rozwijanych przez dłuższy czas. Im dłużej ma być rozwijany projekt, tym bardziej wartościowe są testy jednostkowe. Kiedy je pisać i ile ich pisać – to temat na osobną książkę. W tej chcę jedynie wprowadzić juniora i pokazać mu, jak ogólnie wygląda praca programisty. MÓWIĘ DO CIEBIE, doświadczony programisto, który czyta tego PDF-a, zamiast zająć się pracą (wiem, wiem: prokrastynacja to suka).

6.7. Parsowanie danych JSON i XML

Dane, z którymi będziesz pracował, często występują w jednym z tych dwóch formatów (istnieją też inne, bardziej zaawansowane formaty, np. protobuf, ale na razie nie zawracaj sobie nimi głowy). Przetworzenie danych, by móc z nimi pracować z poziomu kodu, to popularne **parsowa-**

nie. Nie jestem pewny, czy takie słowo istnieje w języku polskim, ale powinieneś je znać: programiści stosują je na co dzień w pracy.

Początkowo parsowanie wydaje się dziwnie trudne, ale z czasem staje się banalną rutyną.

6.8. Praca z narzędziem kontroli wersji, np. Gitem

Zaprzyjaźnij się z Gitem. Jest niezbędny do pracy zespołowej. Jeśli zaczniesz go stosować, nie będziesz mógł się bez niego obyć. Serio. Jeśli zawsze z tym zwlekałeś, odstaw wszystko inne i pobierz jakieś narzędzie wizualne (np. SourceTree).

NIE MUSISZ BYĆ MISTRZEM STOSOWANIA GITA W KONSOLI. Ja znam tylko podstawy konsoli Gita i na co dzień używam narzędzi wizualnych. Jeśli jestem zmuszony do konsoli, na bieżąco googlam to, czego potrzebuję.

Powinieneś wiedzieć, czym są: branch, checkout, clone, merg „mergowanie", commit, push, pull, GitFlow.

Te pojęcia rozjaśnią się, gdy dojdziesz do praktycznej części książki: „Praktyczna ścieżka rozwoju – konkrety". Na tym etapie wypisałem jedynie, co mniej więcej powinieneś potrafić. Później możesz wrócić do tego rozdziału i odhaczać kolejno punkty, z których musisz się dokształcić.

6.9. Usprawnianie pracy w IDE lub innym środowisku

Pewne czynności w pracy wykonuje się regularnie, jednak zajmują one dziwnie dużo czasu, np. zmiana nazwy metody/funkcji. IDE ma wbudowane różne ułatwienia, które usprawniają pracę. Powszechną pułapką jest próba wyuczenia się i wprowadzenia wszystkich tych ułatwień jednocześnie.

Just in time learning: naucz się jednej praktycznej sztuczki (np. przeskoku do następnego erroru albo usunięcia danej linii kodu skrótem klawiszowym) i postaraj się ją stosować. Zapisz skrót klawiszowy na karteczce i przyklej ją pod monitorem. Jeśli po paru dniach uznasz ten skrót za bezużyteczny, olej go. Widocznie nie jest Ci w tej chwili potrzebny. Po paru miesiącach uzbierasz cały arsenał sztuczek, które faktycznie usprawniają pracę. Czasami kolega podpowie coś, co uznasz za przydatne. Zero pamięciówki. Przydatne informacje same zaczynają wchodzić do głowy, gdy widzisz, że faktycznie ułatwiają codzienną pracę.

Oto komentarz mojego kolegi Marka (senior developera) dotyczący sprawnej pracy w IDE:

„Sam miałem pytanie o to, w jakim IDE pracuję i jakie znam z niego skróty, jak miałem rozmowę do projektu dla klienta w Warszawie. Może to wydawać się głupie, ale jednak coś tam sprawdza".

6.10. Dla zielonych w programowaniu – od jakiego języka zacząć?

Oczywiście od angielskiego. ;) A serio, najlepiej zacząć od języka, w którym programuje znajomy. Będziesz miał od kogo się uczyć. Jeśli znajomego brak, dobrym wyborem będzie Python lub JavaScript. Te języki nie wymagają dużo pracy przy konfiguracji środowiska programistycznego – a taka konfiguracja często bywa upierdliwa dla początkującego i zniechęca do programowania. Jeśli wybierzesz np. JavaScript, możesz zacząć od dowolnego edytora tekstu i przeglądarki internetowej.

6.11. Już to potrafię – czy jestem senior developerem?

Niniejsza książka jest o tym, co powinien umieć junior developer, ale sporo osób pyta, co cechuje seniora. Gdy zapytamy trzech senior developerów o różnice między seniorem a juniorem, otrzymamy najprawdopodobniej trzy różne odpowiedzi – w końcu każdy ma inne doświadczenia. Jednak istnieją cechy, z którymi większość powinna się zgodzić. Oto, co wyróżnia senior developera:

Ma doświadczenie w wielu technologiach, które pomaga w podejmowaniu decyzji

By wybrać właściwą technologię dla projektu, wymagane jest doświadczenie w wielu technologiach (no shit, Sherlock). W przypadku niektórych technologii pierwsze efek-

ty widoczne są szybciej, w innych trzeba na nie poczekać – mamy za to większą kontrolę nad szczegółowymi modyfikacjami albo projekt jest w dłuższej perspektywie łatwiejszy w utrzymaniu. Taki balans plusów i minusów.

Jeżeli klientowi zależy na szybkim prototypowaniu pomysłu, to technologie z najszybciej widocznymi efektami będą dobrym wyborem. Podobnych decyzji trzeba podjąć wiele. Z jakimi systemami projekt ma być zintegrowany? Czy istnieje już jakiś gotowy kod źródłowy, z którego można skorzystać? Jeśli tak, to jakie technologie to wymusza? Czy w dłuższej perspektywie zaoszczędzimy czas, jeśli zmienimy istniejącą technologię?

Seniora z taką wiedzą często potrzebuje project manager, aby się z nim konsultować i na tej podstawie podejmować odpowiednie decyzje. Dzięki doświadczonemu programiście, który pomaga dokonać dobrego wyboru, firma może zaoszczędzić sporo pieniędzy (i to bez napisania nawet jednej linii kodu w projekcie). Pamiętaj: najwięcej czasu zaoszczędzamy na tym, czego świadomie decydujemy się nie robić. Podejmowanie takich decyzji (oraz właściwe ich argumentowanie) to jedna z ważniejszych umiejętności seniora. Do tego jednak wymagane jest doświadczenie

z większą liczbą technologii – a to oznacza lata pracy, które każdy senior musi mieć na koncie.

BTW: rekruterzy właśnie na podstawie lat doświadczenia określają, czy kandydat do pracy jest senior developerem – oczywiście to nie najistotniejsze kryterium i wiele osób ma pretensje do działów HR o stosowanie takiego podejścia. Jednak spójrzmy na to z boku: jaką inną metodą ktoś, kto nie pracował w branży (nie mówiąc już o konkretnej technologii, w której szuka się pracownika), może to zweryfikować? Nie ma więc sensu frustrować się z powodu takich praktyk.

Umie się komunikować i współpracować z innymi

Senior potrafi także umiejętnie mówić o problemach występujących w projekcie (junior przez długi czas siedzi cicho, gdy są kłopoty – informuje o nich trochę za późno) i skutecznie współpracować z innymi. Oto przykład dobrej technologicznie osoby, której słaba umiejętność komunikacji szkodzi projektowi: programista koniecznie chce zastosować w projekcie technologię **A**, chociaż nikt w zespole jej nie zna. Gdyby dać mu rok na samodzielną pracę, to może efekt byłby dobry, jednak w przypadku 10-osobowego zespołu taka decyzja **nie zawsze** będzie słuszna. Dziewięć pozostałych osób musiałoby poświęcić mnóstwo czasu, by zapoznać się z nową technologią – a to spory koszt dla firmy. Dlatego umiejętność komunikacji i wybieranie ścieżek optymalnych dla całego zespołu (nie tylko dla jednostki) to bardzo doceniane umiejętności u seniora.

Świetne projekty są pisane przez zespoły – nie przez jednostki. **Dobrą analogię stanowi żonglowanie.** Każdy człowiek jest w stanie żonglować dwiema czy (po paru dniach ćwiczeń) trzema piłkami. Aktualny rekordzista żongluje bodajże 11 piłkami, i to bardzo krótko. Jeden doświadczony wymiatacz może lepiej żonglować dziewięcioma piłkami niż dwóch-trzech świeżaków… Jednak w przypadku projektu, w którym żongluje się 30 piłkami (i to przez parę lat), potrzebny jest świetnie współpracujący zespół.

Łatwo (i szybko) przyznaje, że czegoś nie wie

Jako juniorzy często bierzemy na siebie zadania oparte na technologii, którą znamy bardzo słabo (lub znamy tylko jej nazwę i przeczytaliśmy o niej wpis w Wikipedii), i źle szacujemy czas potrzebny na ich realizację. Mówimy sobie: „jakoś to będzie" i dopiero po dwóch-trzech dniach stwierdzamy, że praca zajmie nam jednak dłużej, bo musimy lepiej ogarnąć daną technologię.

Tymczasem senior bez trudu przyznaje, że czegoś nie wie. Jest świadomy ogromu technologii i od razu uprzedza, że

musiałby poświęcić X czasu na research albo że potrzebuje konsultacji z kimś doświadczonym w danej technologii. Lepsza świadomość tego, czego nie wiemy, bardzo pomaga w estymacji (szacowaniu) pracy.

Wie, jak zabrać się do TDD

Test-driven development oznacza pisanie testów przed pisaniem funkcjonalności. Senior nie zawsze pracuje w TDD, ale ma w tym doświadczenie i wie, jak to ugryźć. Jako junior musisz wiedzieć jedynie, czym jest TDD – i tak masz sporo na głowie.

Walczy o czysty, czytelny kod i wprowadzanie standardów

Każdemu seniorowi, jakiego poznałem, zależało na większej czytelności kodu w projekcie. Jeśli w zespole/firmie spotykają się różne konwencje pisania kodu, to senior często wychodzi z inicjatywą ustalenia wspólnego standardu. Na przykład tak podstawową rzecz jak nawiasy należałoby ujednolicić:

```
function(){
  // pseudokod
}
```

czy

```
function()
{
  // pseudokod
}
```

Co oczywiste, seniora charakteryzuje pewien standard czystości kodu i stosuje on większość konwencji opisanych w książkach typu clean code.

Orientuje się w nowym (cudzym) kodzie źródłowym

To umiejętność, którą można nabyć tylko dzięki wieloletniej praktyce. Nie da się tego etapu przeskoczyć. Podczas rekrutacji na stanowisko senior developera siada się czasami z kandydatem do pair programmingu i śledzi jego tok myślowy, zwraca uwagę na to, jakie pytania zadaje, czego szuka itd. Chodzi o pewną intuicję związaną z kodem, która pomaga odnaleźć się w czymś, co jest już jakiś czas rozwijane.

Umie konstruktywnie krytykować współpracowników

Umiejętność krytykowania to sztuka. Pewnych błędów współpracowników nie można przemilczeć – trzeba na nie odpowiednio zareagować. Dobry senior potrafi skrytykować w taki sposób, by ktoś zrozumiał błąd (a nie myślał, że to zwykłe czepialstwo) i – to już wyższa szkoła jazdy – miał motywację, by w przyszłości bardziej się postarać. Na pewnym etapie ta umiejętność staje się bardzo istotna, ponieważ pozwala wspierać w rozwoju mniej doświadczonych współpracowników.

Bez żalu porzuca rozwiązania, które się nie sprawdzają

Junior niechętnie rezygnuje z rozwiązania, w które zainwestował sporo czasu. Próbuje je ratować i w konse-

kwencji traci jeszcze więcej czasu, brnąc w coś, co kiepsko się sprawdza. Natomiast senior – w imię ograniczenia strat – nie boi się zejść z raz obranej ścieżki.

Istnieje taki gatunek małp, które łapie się, umieszczając jedzenie wewnątrz pojemnika, do którego mieści się dłoń, ale nie zaciśnięta pięść. Małpa próbuje wyciągnąć jedzenie i w efekcie sama staje się pożywieniem dla łowcy. Taką małpą bywa junior, który z uporem trwa przy nieefektywnym rozwiązaniu. Wprawdzie nie kończy jako posiłek dla plemienia łowców małp (do tej pory z taką formą kary się nie spotkałem, niezbadane są jednak techniki motywacyjne wielkich korporacji), ale traci w firmie sporo zasobów czasowych.

Niektóre umiejętności są zależne od konkretnej działki technicznej… jednak znając powyższe punkty, już powinieneś z grubsza wiedzieć, jak przebiega rozwój po zdobyciu stanowiska junior developera.

7.

Szkoła, studia i odwieczne pytanie o matematykę

Uwaga! W żadnym wypadku nie przyjmujcie tego rozdziału za prawdę objawioną. Potraktujcie go jako jedną z wielu opinii, które pomagają podjąć decyzję o edukacji.

Jestem typem człowieka, który woli uczyć się samodzielnie, dlatego też mam poglądy, jakie mam – i nie jestem fanem formalnych studiów. W tym rozdziale opisałem, jakie decyzje związane ze studiami podjąłem, dlaczego postąpiłem tak, a nie inaczej. A na koniec: wnioski.

7.1. Bez matematyki ani rusz?

Zacznijmy od pytania o matematykę, które nieustannie zadajecie. Nie, nie musiałeś być dobry w szkole z matematyki, by zostać programistą. Jeśli pracując, natkniesz się na zadanie wymagające konkretnej wiedzy matematycznej, po prostu się douczysz.

I nie przejmuj się, jeśli nauczyciel matematyki mówi: „Jak chcesz być informatykiem/programistą, to MUSISZ być dobry z matematyki". Chcesz zostać murarzem, to pytaj murarza, jak zostać murarzem. Chcesz zostać programistą, to pytaj programistę, a nie nauczyciela matematyki. Nie przejmuj się jego opinią i nie trać czasu na bezproduktywną dyskusję. Warto nauczyć się ignorować niektórych ludzi i czasami przytaknąć dla świętego spokoju, a później i tak robić swoje.

Na tym można zakończyć temat matematyki. Serio.

Ewentualnie warto wspomnieć, że matematykę wykorzystują częściej osoby piszące np. silniki do tworzenia gier, rzeczy związane z fizyką itp. Jednak w wielu przypadkach nie jest ona dla programisty **aż tak istotna**. To, czego nauczyłeś się w szkole podstawowej i gimnazjum (czy w ośmiu klasach szkoły podstawowej, jeśli jesteś z rocznika starszego niż '86), w większości przypadków wystarczy aż nadto.

Liczy się umiejętność logicznego myślenia, którą ćwiczysz z każdym kolejnym projektem i każdym rozwiązanym problemem.

7.2. Jak podchodziłem do kwestii szkoły/studiów

Dosyć specyficznie. Pod koniec gimnazjum/na początku liceum zacząłem tworzyć strony WWW i zarabiać dzięki temu pierwsze pieniądze. Studia postrzegałem jako etap,

który pomoże mi w przyszłości znaleźć sensowną pracę. Skoro jednak sam organizowałem sobie pracę (w czasach LO miałem pierwszą mikrofirmę pod skrzydłami Akademickich Inkubatorów Przedsiębiorczości), to po co miałem w takim razie iść na studia?

Tak myślałem jako licealista. Nie zamierzałem studiować, ale rodzice mnie do tego namawiali. Poszedłem więc na kompromis i zdecydowałem się zrobić inżyniera informatyki zaocznie. Dziś żałuję, że dałem się przekonać. Poznałem paru ciekawych ludzi (jedyny plus), ale ogólnie uważam, że tylko zmarnowałem czas i pieniądze.

Gdybym dzisiaj ponownie miał 19 lat, to prawdopodobnie poszedłbym na pierwszy rok (albo pierwszy semestr) dziennych studiów, by zobaczyć, czy to dla mnie. Po pierwszym semestrze zdecydowałbym, co robić dalej. Myślę, że zrezygnowałbym z uczelni i spróbował załapać się gdzieś na praktyki, by podpatrzeć, jak w rzeczywistości wygląda praca osób zajmujących się tym, czego chcę się nauczyć.

Jest to jednak opinia osoby, która w dłuższej perspektywie chce prowadzić własną firmę, a nie pracować większość

życia na etacie. Studia mają swoje zalety i w wielu przypadkach warto na nie iść.

7.3. Dlaczego warto iść na informatykę?

Rówieśnicy o podobnych zainteresowaniach. Nauka przychodzi łatwiej, jeśli otaczamy się osobami, które mają ten sam cel. Dzięki studiom poznasz wielu ciekawych ludzi, którzy mogą Cię zainspirować do poszukiwań własnej ścieżki w życiu. Dlatego warto wybrać uczelnię, na którą większość osób chce się dostać, a nie taką, na którą się dostają, bo gdzie indziej ich nie chcą. Unikałbym też uczelni, na której wielu studentów studiuje, ale nie do końca wie dlaczego: „Nie wiem, co robić ze swoim życiem, więc na razie robię studia". ;)

Podsumowując: jeśli już studia, to jakaś konkretna uczelnia (na której można poznać ambitnych i fajnych ludzi), a nie kolejna pseudoszkoła kręcąca biznes na nieporadnych dzieciach, których rodzice płacą za czesne, by ich syn/córka mieli za parę lat papierek magistra.

Zależy Ci na pracy na etacie. To, że ja nie zamierzam pracować całe życie na bezpiecznym etacie, wcale nie znaczy, że etat to zło. Jeśli jesteś typem człowieka, który lubi regularne godziny pracy, liczy się dla niego bezpieczeństwo i dobrze się czuje w świecie korporacyjnym, to studia mogą być bardzo przydatne. W wielu firmach formalnym wymogiem awansu jest wyższe wykształcenie. Oczywiście absolwenci studiów informatycznych zakładają firmy i mają się bardzo dobrze. Nie przejmujcie się moją opinią, że studia są głównie dla etatowców.

Jeśli zadbasz o to, by tytuł z uczelni szedł w parze z umiejętnościami (nabytymi dzięki projektom realizowanym na uczelni czy projektom hobbistycznym), to wybierasz bezpieczną i dla wielu osób właściwą drogę. Nie daj sobie wmówić, że rzucenie studiów i samodzielna nauka w IT jest **zawsze lepszym rozwiązaniem**. Gdybyśmy wszyscy działali i myśleli tak samo, świat byłby strasznie nudny.

Potrzebujesz bodźca, który pobudzi Cię do nauki. Nawet jeśli masz naturę samouka, wiesz, że nie zawsze łatwo się zmobilizować do działania. Dla wielu osób zbliżające się egzaminy to niezbędna motywacja. Potrzeba regularnej nauki do egzaminów może też pomóc w wyrobieniu sobie dobrych nawyków.

Studia to nie szkoła zawodowa. Błędne jest założenie, że studia informatyczne mają głównie przygotować do wejścia na rynek pracy. Jeśli chcesz się po prostu rozwinąć jako człowiek i nie zależy Ci na szybkim starcie w życie zawodowe, to studia mogą być świetnym okresem w życiu. Każdy ma inne priorytety i inaczej ukierunkowane ambicje – należy o tym pamiętać, zanim zaczniemy kogoś pouczać.

7.4. Dlaczego nie warto iść na informatykę?

Sporo wiedzy nie przyda Ci się w przyszłości. Jeśli jesteś nastawiony na wiedzę praktyczną i szybko chcesz zacząć pracę w zawodzie, studia mogą nie być dobrym wyborem. Część materiału może się przydać, jednak większość uznasz pewnie za zbędną: wyższą matematykę, elektrotechnikę i wiele innych przedmiotów. Jeśli zależy Ci na ogólnym

poszerzaniu horyzontów, skorzystasz na tych przedmiotach, ale jeśli już mniej więcej wiesz, czym chcesz się zająć w życiu, będzie to duże obciążenie. Odniesiesz wrażenie, że lepiej spożytkowałbyś czas, realizując się w jakimś projekcie pobocznym.

Takie projekty poboczne to kolejny powód: **koszt alternatywny**, którego nie widać. Kosztem alternatywnym jest wiele projektów, które nigdy nie powstaną, ponieważ poświęciłeś czas na studia. Nie nauczysz się wielu rzeczy, których mógłbyś się nauczyć, gdybyś inaczej gospodarował czasem.

Chcesz w przyszłości prowadzić swój biznes. Jeśli widzisz się w przyszłości jako przedsiębiorca, na studiach często będziesz czuł się sfrustrowany. Praktyki czy praca (nawet za bardzo niską stawkę) w średniej lub dużej firmie mogą się okazać dużo bardziej wartościowe. Dzięki nim nie tylko opanujesz wiele kwestii czysto technicznych, ale również dowiesz się, jak wygląda sprawna (lub wręcz przeciwnie) organizacja pracy.

Nie potrzebujesz specjalistycznego sprzętu. Mało jest takich kierunków jak informatyka (a w szczególności programowanie), na których możesz tak wiele nauczyć się samodzielnie i bez specjalistycznego sprzętu. Przykładowo lekarz nie może (legalnie) ot tak pokroić zwłok i zobaczyć, jak wygląda człowiek od środka.

7.5. Krótko o belfrach

Można odnieść wrażenie, że mam uraz do nauczycieli i dlatego wyżej stawiam samodzielną naukę od tradycyjnej szkoły/uczelni. Nauczyciel, który czyta moje wypociny, może poczuć się urażony. Dlatego podkreślam: uważam za beznadziejny aktualny **system nauczania**, natomiast sami nauczyciele są zazwyczaj w porządku. Często mają związane ręce przez system. Nauczyciele, których wspominam najlepiej, realizowali tylko niezbędne minimum oficjalnego programu, a później uczyli tego, co faktycznie było ciekawe (i czasami nawet przydatne!). Po oficjalnych godzinach lekcyjnych nauczyciel bardzo często okazuje się znacznie sympatyczniejszy, co tłumaczy też popularność korepetycji.

8.

CV, rozmowy kwalifikacyjne i szukanie pracy

Co warto napisać w CV, czego spodziewać się na rozmowach kwalifikacyjnych (jak do nich podchodzić) i na co zwrócić uwagę w samym procesie szukania pracy – tego dowiesz się z niniejszego rozdziału.

8.1. Moje pierwsze CV na stanowisko programisty – dlaczego było do bani?

Co zawierało moje CV:

- Sporo informacji o tym, że byłem freelancerem, który tworzył strony WWW i pozycjonował je w wyszukiwarkach.

- Listę stron WWW, które wykonałem. Cześć z nich była już niedostępna w sieci, a te, które zostały, nie były zbyt ambitne od strony technicznej.

- Miejsca, w których moje projekty/strony zostały wspomniane (np. „Dziennik Zachodni" czy różne strony WWW), napisałem też, kto kupił strony.

- Szczegóły dotyczące ruchu na stronach WWW, które stworzyłem.

- Umiejętności. Te, które wymieniłem, były bardzo ogólne: tworzenie stron WWW z wykorzystaniem HTML/CSS oraz PHP/MySQL, programowanie obiektowe w Javie i C# (to w Javie da się programować nieobiektowo?), znajomość baz danych i oczywiście szybkie uczenie się (idealny punkt do wpisania, jeśli nie mamy się czym pochwalić).

- Języki obce, wykształcenie (standard – nie ma czego tu omawiać).

Co było w nim do bani? Na tyle dużo, że mogę to ładnie wypunktować. :)

- Gdy aplikujesz na dane stanowisko, które wymaga X, Y i Z, to, co wpisujesz w CV, powinno być powiązane z tym stanowiskiem. Po cholerę wymieniałem strony WWW, skoro aplikowałem na stanowisko Java developera – sam się sobie dziwię. Dużo lepiej byłoby umieścić link do webserwisu napisanego w ramach nauki.

- Wspomniałem już o nieistotnych informacjach typu: ile odwiedzin miały moje strony WWW (co byłoby w su-

mie dobrą informacją… gdybym ubiegał się na stanowisko w dziale marketingu). Ważniejsza jest przejrzystość, którą przez takie zbędne dane traciłem. Szczegóły zawsze można dołączyć w osobnym dokumencie lub dopowiedzieć w trakcie rozmowy. Samo CV powinno być krótkie i na temat.

- Było widać, że w przypadku stanowiska, na które aplikowałem, nie miałem za bardzo czym się pochwalić – tylko zaśmiecałem CV zbędnymi informacjami.

- Zamiast lać wodę o umiejętnościach, powinienem wypisać technologie, które znam dobrze, średnio i słabo. Krótko i do rzeczy.

Co powinienem był umieścić w CV jako osoba z małym doświadczeniem? Wymienić kilka technologii, których się uczyłem; podać linki do projektów, które wykonałem w ramach nauki; i zaznaczyć, w jakim kierunku planuję się rozwijać.

Typowe CV studenta świeżaka zawiera wszystkie technologie, które ten student przerobił na studiach albo o których przeczytał na Wikipedii. To, że miałeś laborkę z asemblera, nie oznacza, że należy to wpisać w doświadczenie. Im programista bardziej doświadczony, tym mniej informacji umieszcza w CV – obserwuje się taką tendencję.

8.2. Drugie CV – lepsze, ale wciąż nieidealne

Z drugim CV poradziłem sobie ciut lepiej:

- Wypisałem krótko, gdzie pracowałem i na jakim stanowisku.
- Wymieniłem moje główne techniczne umiejętności. Zamiast lać wodę: „Tworzyłem strony WWW, które odwiedzały trzy osoby dziennie" napisałem: „Web Development: xHTML, CSS, JavaScript, Servlets/JSP".
- Podałem główne narzędzia/IDE, z których korzystałem w pracy.
- Krótko i na temat.
- Jedyne, czego zabrakło, to **konto na GitHubie z projektami do wglądu**.

Mogłem też podzielić umiejętności techniczne na główne (które znam najlepiej) i takie, które znam średnio czy dopiero w nich raczkuję. Szczera samoocena wygląda znacznie lepiej od przydługiej listy mądrych słówek.

8.3. Gdybym dzisiaj przygotowywał CV

Nie jestem ekspertem od pisania CV i pewnie w wielu poradnikach internetowych znalazłyby się inne wskazówki od moich. Jednak na podstawie własnych doświadczeń

w aplikowaniu do pracy jako programista i jako przyszły potencjalny pracodawca radziłbym:

- Wypisać jedną-trzy główne technologie (unikamy przydługawej listy typowej dla CV studenta świeżaka).

- Przygotować konto na GitHubie z solidnie napisanym kodem źródłowym. Najważniejszy jest nie poziom skomplikowania projektu, tylko jakość napisanego kodu. Testy jednostkowe (sensowne testy!) też zawsze dobrze wyglądają.

- Oprócz firm, w których pracowaliśmy, wymienić stanowisko i krótko opisać zakres obowiązków.

- W przypadku projektu, który był wykonywany poza pracą zawodową i nie znajduje się na GitHubie, podać link do np. Google Play, Apple Store czy strony projektu.

- Pisać na tyle zwięźle, by zmieścić się na jednej stronie A4, dwie strony to maksimum.

- Wymienić najciekawsze informacje na początku.

Filmik dołączony do CV?

Gdy aplikowałem do drugiej pracy jako programista, nagrałem krótki, 10-minutowy filmik, w którym raz po angielsku, a raz po niemiecku mówiłem szerzej o tym, co napisałem w CV. W zwykłej czarnej koszulce i jeansach…

Ot luźno omówiłem CV i przy okazji pokazałem, że znam inne języki niż ojczysty. Prawie każda osoba, która zadzwoniła, pochwaliła ten pomysł.

Warto się przełamać i spróbować czegoś podobnego. Filmik był zwykłym niepublicznym materiałem na YT (czyli takim, który można uruchomić tylko wtedy, gdy ma się bezpośredni link – nie można go wyszukać w wyszukiwarce).

8.4. Proces szukania pracy

Gdy masz przygotowane CV pod stanowisko, które Cię interesuje (np. Android developer), zaczyna się zabawa w zbieranie adresów e-mail. Postaraj się wygooglać parędziesiąt firm i ofert w Twojej okolicy zajmujących się programowaniem, następnie wyślij im CV z informacją, że szukasz pracy.

Nawet jeśli dana firma obecnie nikogo nie szuka, to bardzo prawdopodobne, że zna kogoś, kto akurat poszukuje pracowników, i przekaże Twoje CV (o ile jest sensowne).

Warto również (na początku) aplikować do firm z miast, w których nie zamierzasz pracować. Dostaniesz parę telefonów więcej, odbędziesz kolejne rozmowy na Skypie i oswoisz się z całym procesem. Szybko zauważysz, że rozmowy o pracę są do siebie podobne, i nabierzesz w nich wprawy. Możesz odmówić osobistego spotkania – zawsze to kolejna rozmowa telefoniczna więcej, z której czegoś się nauczysz.

8.5. Rozmowy kwalifikacyjne

Rozmowy kwalifikacyjne są schematyczne.

Na pewno zapytają Cię, **dlaczego chcesz u nich pracować i jak ich znalazłeś**. Odpowiadasz, że jesteś zainteresowany rozwojem w kierunku X/Y i sądzisz, że akurat ta firma Ci to umożliwi.

Spytają też, co wiesz o firmie. **Warto przeczytać, w jakich projektach brał udział potencjalny pracodawca.** Jeśli tego wcześniej nie sprawdziłeś, nie ściemniaj. Powiedz, że wygooglałeś ofertę pracy i wysłałeś CV. Tyle.

Przejdźmy do sedna:

- Naucz się wypowiadać na temat projektów, nad którymi pracowałeś: jakie pojawiły się największe problemy, jakie technologie stosowałeś, wspomnij o narzędziach pracy.

- Dostaniesz pytanie związane z którąś z technologii wymienionych w CV. Dlatego warto podzielić je na te, które zna się dobrze i słabo. Jeśli zadeklarowałeś „dobrą znajomość" danej technologii, a w rzeczywistości kiepsko się w niej orientujesz, to szybko się spalisz.

- Staraj się być szczery i zaciekawiony stanowiskiem pracy. Jeśli czegoś nie wiesz, przyznaj się do tego: „Nie stosowałem tego, nie mogę nic na temat tego powiedzieć". Jeśli pracodawca/rekruter powie coś więcej o firmie, wykaż zainteresowanie i zadawaj pytania: **pytaj o wszystko, co Cię ciekawi** (o ile ma to związek ze stanowiskiem, o które się ubiegasz). Ty starasz się o pracę, ale to przede wszystkim firma stara się o dobrego pracownika.

Jeśli idziesz na rozmowę osobistą, nie bój się zapytać o dress code. Zazwyczaj można przyjść luźno ubranym, ale warto się doinformować.

8.6. Zbieraj doświadczenie w aplikowaniu i wyciągaj wnioski

Im więcej rozmów odbędziesz, tym lepiej. Poznasz swoje braki, które możesz przecież nadrobić. Nikt Ci nie zabroni aplikować do tej samej firmy ponownie za parę miesięcy czy rok.

Aplikowanie to bardzo dobry test. Zamiast bezcelowo uczyć się czegoś nowego w domu, warto pozgłaszać się do różnych firm i przekonać, co nam idzie dobrze, a nad czym należy popracować.

8.7. Zadanie praktyczne

Podczas rekrutacji często będziesz dostawał do rozwiązania zadanie praktyczne lub test – kolejny argument za tym, by aplikować częściej. Testy są bowiem do siebie podobne i po paru razach będziesz już wiedział, czego się spodziewać.

Ja również musiałem wykonać kilka takich zadań praktycznych. Raz był to przydługawy test z pytaniami zamkniętymi i otwartymi. Pracodawca oszczędza w ten sposób czas, bo nie musi poświęcać go na długą rozmowę sprawdzającą kandydata. W testach często liczyła się nie idealna odpowiedź, tylko fakt, że umie się kombinować i rozpracować dany problem.

Warto zrobić kilka testów praktycznych na Codility.com. Taki test nie jest w żadnym wypadku odzwierciedleniem wiedzy: albo potrafisz rozwiązać problem i masz 100%, albo nie umiesz z ruszyć z miejsca i masz 0%… To kwestia szczęścia, a jednak część pracodawców wysyła taki test. W ramach praktyki na Codility.com można wykonać parę testów za darmo.

Część z nich jest bardzo podobna do zadań z https://www.reddit.com/r/dailyprogrammer – również polecam tam zerkać i ćwiczyć szare komórki.

Innym razem dostałem do wykonania zadanie: mail przyszedł o 9 rano, dano mi czas do 15-16. Dostałem link do

webserwisu, miałem pobrać dane i wyświetlić je w aplikacji na telefonie, dodatkowo zaimplementować część danych na Google Maps Api.

WAŻNE: liczyło się nie to, **czy** wykonam zadanie w 100%, tylko **jak** je wykonam. Dużo **lepiej jest napisać mniej, ale lepszej jakości przemyślanego kodu**, niż na siłę próbować zmieścić się ze wszystkim w czasie, byle działało.

Czasami w ramach rekrutacji będziesz musiał pobrać projekt do modyfikacji, więc ważna jest umiejętność pracy z Gitem. Znowu ten Git… Jeśli masz zapamiętać jedną wskazówkę z tej książki, niech będzie to: **NAUCZ SIĘ STOSOWAĆ GITA.** To niesamowite, jak bardzo usprawnia on codzienną pracę. Chyba wolałbym zrezygnować z kawy niż z Gita. Zdecydowanie narzędzie numer jeden.

8.8. Dlaczego nie polecam zaczynać od pracy zdalnej/freelancingu

Często słyszy się, że programiści mogą pracować zdalnie dla zagranicznych klientów. To oczywiście prawda, jednak nikomu – szczególnie junior developerom – nie radzę od tego zaczynać.

Pracując stacjonarnie, nauczysz się sporo o pracy w zespole programistów, szybciej się rozwiniesz, lepiej zorientujesz w stawkach, ogólnie dowiesz się, z czym to się je. Przejście na pracę zdalną można rozważyć po paru latach pracy stacjonarnej, ale na początek to kiepski pomysł.

Bez konkretnego doświadczenia trudniej o zdobycie zleceń, nie wiadomo, jak wycenić taką pracę, jak to w ogóle raportować… Efekt? Frustracja i niepotrzebny stres.

8.9. Przydatne portale

Jeśli nie słyszałeś wcześniej o serwisach typu **GoldenLine.pl, Xing.com i LinkedIn.com**, to czas nadrobić braki. Dlaczego? Gdy założysz na nich konto i w profilu umieścisz informację, że szukasz pracy jako programista, to ludzie z działu HR sami Cię znajdą. Wielokrotnie spotkałem się z praktyką, że za ściągnięcie do pracy programisty dostawało się od 1000 zł (w małych firmach) do 5000 euro (taka była nagroda za znalezienie programisty do firmy, w której pracowałem) i więcej.

Poświęć godzinkę i załóż konto. Przyda się podczas szukania pracy i późniejszej ewentualnej zmiany.

8.10. Wątpisz w siebie, chociaż sporo wiesz? Prawdopodobnie masz syndrom oszusta

„I don't know what I'm doing. Everything I've done in my life I've just made up as I went along. That's what everyone is doing. That's how life works. That's how evolution works" – ten cytat

pochodzi z bloga programisty gry Super Meat Boy. Dotyczy on powszechnego problemu (nie tylko w IT) zwanego **syndromem oszusta** (impostor syndrome).

Mimo że mamy wiedzę i jej potwierdzenie (np. skończone projekty w danej technologii), to czasami czujemy, że po prostu dopisało nam szczęście, a tak naprawdę jesteśmy oszustami, którzy tylko dobrze udają. Znasz to? Nie martw się. Z różnych ankiet wynika, że w niektórych grupach nawet 70% osób doświadczyło takich myśli.

Dlaczego wspominam o tym w rozdziale o CV? Nie chcę, żebyś się niepotrzebnie martwił i z powodu głupich wątpliwości opóźniał aplikowanie do firm.

Gdy jesteśmy nowi w zespole, to normalne, że dopytujemy innych, jak coś jest zbudowane, i że na początku jesteśmy mniej wydajni. Ja nawet po roku pracy nadal czułem, że jestem jakimś przebierańcem, a pozostali to „prawdziwi" pracownicy. Dopiero gdy zerknąłem na ostatnie sprinty i zauważyłem, że przez ostatnich parę tygodni wykonałem około 60% tasków androidowych (na dwóch programistów Android w zespole), dotarło do mnie, że może faktycznie jestem pełnoprawnym pracownikiem. Zerknięcie na suche fakty pomaga w walce z problemem.

Jak radzić sobie z syndromem oszusta? Wypisać listę rzeczy, które się zrobiło. Dzięki faktom łatwiej przekonać umysł, że jednak wiemy, co robimy. W moim przypadku taki cel spełniają podsumowania na blogu. Pozwalają mi uporządkować myśli, radzić sobie z syndromem oszusta, a przy okazji są chętnie czytane.

Przykładowe podsumowanie: **JuniorDeveloper.pl/podsumowanie**.

Pamiętaj, że dużo osób ma ten problem. Gdy syndrom się nasila, to prawdopodobnie dlatego, że się rozwijamy. **To coś dobrego.** Będąc całe życie w strefie komfortu, tylko stoimy w miejscu.

8.11. Podsumowanie

- Przygotuj krótkie i zgrabne CV, uwzględniając rady, które wymieniłem.

- Jeśli chcesz się wyróżnić z tłumu, nagraj krótki filmik, w którym rozwiniesz poszczególne punkty CV.

- Załóż konto na GitHubie i wrzuć na nie dwa–trzy w miarę czysto napisane projekty (liczy się jakość, nie ilość: lepiej wrzuć jeden zamiast kilku śmieciowych).

- Nie bój się aplikować. Aplikuj masowo, nabieraj doświadczenia w rozmowach kwalifikacyjnych i testach praktycznych.

Dla wielu ten rozdział może być odkrywaniem Ameryki, ale zadziwiająco dużo osób nie radzi sobie z szukaniem pracy…

9.

Dzień, organizacja i role w pracy. Agile, scrum i inne mądre słówka

Praca programistów nie jest tania, dlatego powstają różne procesy/metody pracy, by ich czas był wykorzystywany efektywnie.

Gdy pierwszy raz usłyszałem o efektywnej organizacji pracy zespołu programistów, pomyślałem, że chodzi o to, by pracowali długo lub intensywnie (a najlepiej jedno i drugie). Jednak takie podejście prowadzi tylko do wypalenia i na dłuższą metę się nie sprawdza.

Współczesna organizacja pracy w firmach IT polega głównie na tym, że programiści pracują nad zadaniami o najwyższym priorytecie. Podejście takie skupia się na tym czego lepiej **nie robić**, a nie robić możliwie dużo w krótkim czasie.

By system działał sprawnie, potrzebne są różne role w zespole. Opisałem je poniżej, skupiając się naturalnie na roli programisty.

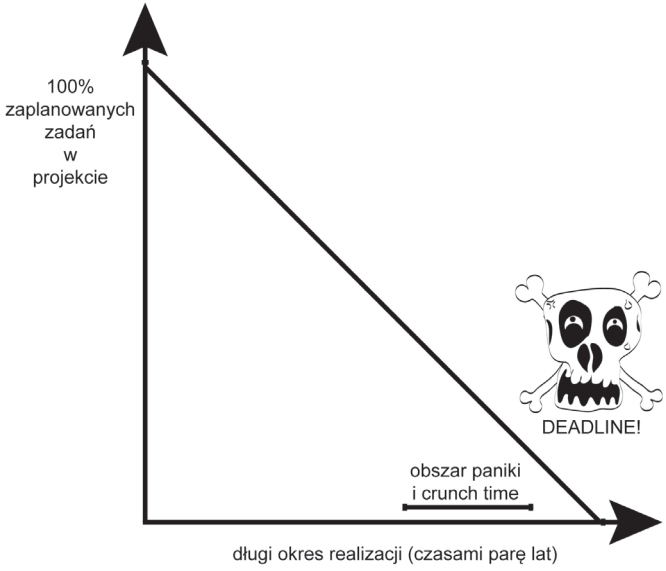
9.1. Agile i scrum – z czym to się je?

To nie jest podręcznik o scrumie, a ja nie jestem w nim ekspertem. Chcę tylko przybliżyć, jak rozumiem tę metodę pracy:

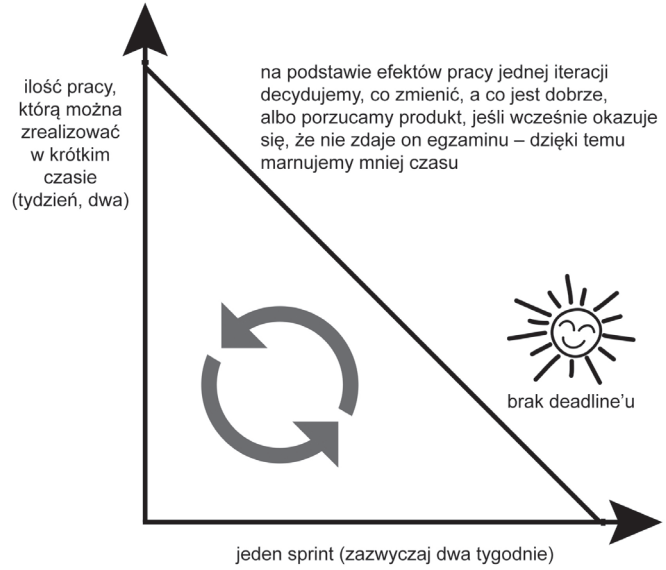
- jak pracowałem według niej prawie trzy lata na etacie w różnych zespołach;
- jak stosuję agile w codziennym życiu (wykorzystałem scrum m.in. podczas pisania pracy inżynierskiej czy nauki do certyfikatu oracle'owego z Javy);
- jak zmieniło to moje podejście do własnych projektów i pracy na własną rękę.

Big picture

Obraz mówi więcej niż tysiąc słów, dlatego wstawiłem dwa obrazki (hm… to tak, jakbym napisał dwa tysiące słów: czas na przerwę w pisaniu :)).



Nieagile

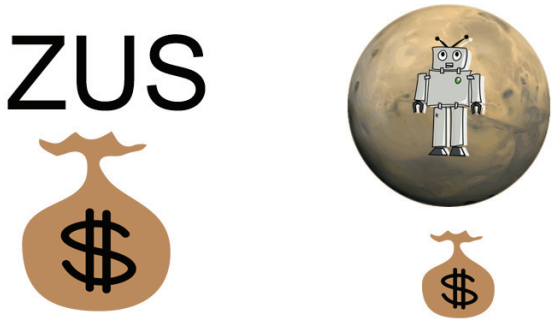


Agile

Bardzo to uprościłem, ale na potrzeby książki wystarczy. Ważne są różnice między tymi metodami pracy: pierwsza polega na tym, że planuje się cały projekt według sztywnej specyfikacji i wyznacza deadline, czyli termin, kiedy projekt ma być skończony. W drugiej metodzie staramy się nakreślić, jakie cele dla klienta ma spełniać produkt, i wstępnie w krótkich iteracjach budujemy to, co w danym momencie uważamy za sensowne. Następnie efekty pracy konsultujemy z klientem i decydujemy, czy wprowadzamy zmiany, czy trzymamy się obranego kierunku.

9.2. Lepiej pracować według podejścia agile?

W przypadku sztywnych wytycznych projektowych, gdy produkt definiuje urzędnik, nie liczy się to, czy nasza praca jest sensowna (przykład: informatyzacja ZUS-u, która nie działa i kosztowała więcej niż wysłanie łazika na Marsa). Często mamy związane ręce i wymuszany jest pierwszy sposób pracy.



Natomiast w przypadku projektów, które muszą się sprawdzić na rynku, zdecydowanie lepsze jest drugie podejście:

- wielu rzeczy nie da się przewidzieć i decyzję można podjąć dopiero wtedy, gdy produkt nabierze kształtu, po paru iteracjach;

- nie zmarnujemy tygodni/miesięcy/lat pracy zespołu, jeśli wcześniej zauważymy, że dana funkcja jest niepotrzebna lub za mało istotna – w tym samym czasie możemy zrobić coś, co przyniesie lepsze efekty;

- szybciej się uczymy na błędach i wyciągamy wnioski.

O ile nie chcesz zostać scrum masterem, to taka wiedza na razie Ci wystarczy. Przejdźmy teraz do tego, co dla Ciebie istotne jako developera.

9.3. Sprint…

Moment… STOP!

Dlaczego opisuję metodykę pracy, a nie samo programowanie czy technologie? Przecież ta książka ma pomóc w ukierunkowaniu się, by znaleźć pierwszą pracę jako programista.

Znajomość podstaw procesów pracy jest niesamowicie przydatna podczas rozmowy kwalifikacyjnej. Pracodawca docenia osoby, które wiedzą, jak pracować efektywnie.

Dodatkowo pomoże Ci to w organizowaniu procesu nauki czy rozwoju własnych pierwszych projektów.

Sprint

To powtarzająca się iteracja. Najczęściej trwa dwa tygodnie. Na początku planujemy, co chcemy zrobić w tej iteracji, i wyciągamy z **backlogu** (taka piwnica, w której nasz product owner Fritzl przechowuje różne user story/taski – więcej o rolach takich jak product owner w dalszej części) różne zadania, które możemy zrealizować w tej iteracji.

Następnie zaczyna się wesołe tworzenie. W trakcie sprintu jest parę stałych meetingów, np. backlog grooming, podczas którego z product ownerem omawiamy planowane zadania i staramy się oszacować, ile pracy wymaga każde z nich.

Pod koniec sprintu prezentujemy efekty pracy klientowi czy szefowi. Nazywa się to **review** i pozwala w krótkim czasie zebrać wartościowy feedback, który być może zaowocuje już w następnym sprincie. Dobrze wiedzieć, że coś klientowi się nie podoba, i zmienić to wcześnie, zamiast podążać w złym kierunku, prawda?

Na koniec iteracji siadamy z zespołem do retrospektywy i decydujemy, co można poprawić w następnym sprincie.

9.4. Role w organizacji pracy

Product Owner

Czyli manager. On decyduje o kryteriach odbioru zadań: musi odebrać każdy ficzer, zanim zostanie on uznany za gotowy. Pewnie słyszeliście powiedzenie: gdzie kucharek sześć… No właśnie: musi być jedna główna osoba, która decyduje o kierunku i kształcie produktu. Inaczej całość nieuchronnie dryfuje ku nijakości.

Product owner prowadzi główne rozmowy z zarządem/szefem na temat produktu. Dzięki temu developer ma wolną głowę i może koncentrować się na kolejnym zadaniu. O całokształt i finalny produkt martwi się product owner. On decyduje, w jakiej kolejności zostaną wykonane zadania, co aktualnie jest najważniejsze dla produktu, a co może zostać odłożone na później.

Product owner to też pierwsza osoba, do której można się zwrócić, jeśli coś nie działa lub są jakieś pytania. Dzięki temu developerzy mają spokój – nikt nie przeszkadza im w pracy. Osoba z działu marketingu nie wpadnie nagle do pokoju z pomysłem i nie zaburzy pracy całego zespołowi programistów: product owner ją przechwyci i z nią porozmawia.

Product owner to osoba do kontaktu na zewnątrz, czy to z szefem firmy, czy z inwestorami mającymi pytania o produkt. Często jest on również odpowiedzialny za

wszelkie kalkulacje potencjalnych zysków/kosztów danego działania. Dlatego dużą zaletą takiej osoby jest sprawne poruszanie się w bazach danych. Niby powinna mieć odpowiednie narzędzia BI (business intelligence), które ją w tym wyręczą… w praktyce zdarzało się jednak, że product owner musiał pisać zapytania SQL-owe, by przygotować odpowiednie dane do analizy. W takich wypadkach zazwyczaj bierze on do pomocy osobę z zespołu, która najlepiej orientuje się w bazie danych projektu.

Scrum master

Dba o to, żeby wszystko sprawnie działało. Żeby meetingi niepotrzebnie się nie przedłużały i żeby brały w nich udział osoby, które są rzeczywiście potrzebne: po co mają w nich uczestniczyć zbędne osoby? Każda godzina to pieniądze. Scrum master obserwuje, jak działają procesy między zespołami/osobami, i stara się je usprawnić/wprowadzić nowe. Moderuje retrospektywę po sprincie, analizuje, co można by poprawić lub zmienić w aktualnie istniejących procesach.

Spotkałem się z eksperymentem, w którym rolę scrum mastera przejął developer z zespołu. Był on w ⅔ developerem, a w ⅓ scrum masterem. Na dłuższą metę pomysł się nie sprawdził, ale przypomina o ważnej regule: nic nie jest sztywno narzucone. Należy eksperymentować i śledzić wyniki, a później po prostu pracować według zasad, które się sprawdzają. Jeśli coś nie działa, zmieniamy to.

Developer

Czyli osoba, która pracuje nad kodem źródłowym. Czasami w zespole znajduje się również grafik/designer, ewentualnie grafika jest dostarczana z zewnątrz. Mój ostatni zespół składał się w 100% z programistów plus jednego testera (QA engineer), a designer pracował jednocześnie dla kilku zespołów w firmie. Dąży się do tego, by jeden programista mógł docelowo pracować nad każdym elementem projektu: nanieść poprawkę w webserwisie, coś pogrzebać w bazie danych, aplikacji mobilnej czy front-endzie strony WWW. W praktyce każdy ma swoją specjalizację, np. ktoś w zespole jest głównie od Androida, ale gdy zdarzy się sprint, w którym 90% pracy polega na przebudowie czegoś serwerowego, taka osoba też stara się w miarę możliwości pomóc. Każdy w czymś się specjalizuje, ale dąży się do tego, by wszyscy mieli przynajmniej podstawową wiedzę o innych elementach projektu.

Pozostałe role?

Trzy opisane wyżej role składają się na standard scrumowy. Czasami tworzy się dodatkowo role małe i niestandardowe, np. wprowadziliśmy kiedyś w zespole funkcję… **cookie commissionera**. Była to osoba, która podczas każdego sprintu robiła coś miłego dla zespołu: kupowała ciastka, organizowała wspólny wypad integracyjny. Mała rzecz, a znacznie poprawiła atmosferę w zespole. To tylko ciekawostka, która przypomina, że system jest elastyczny i można trochę pokombinować. Ważne, by wiedzieć, na czym polegają trzy główne role w scrumie.

9.5. Podsumowanie

- Znasz już powody, dla których nie warto zostać programistą. Od programisty, tak jak od każdego innego specjalisty, wymaga się konsekwentnej pracy. Nie jest to „łatwa droga na skróty", na której przy okazji dobrze zarobisz. Czasami warto odpuścić i spróbować czegoś innego.

- Jeśli postanowiłeś dasz sobie szansę próbować albo masz już pewność, że to zajęcie dla Ciebie, to z kolejnego rozdziału dowiesz się, nad czym popracować, by znaleźć pierwszą pracę w tym zawodzie.

- Dowiedziałeś się, że matematyka nie jest niezbędna, a studia przydają się tylko czasami.

- Potrafisz przygotować odpowiednie CV.

- Wiesz już sporo o organizacji pracy typu agile/scrum.

Świetnie. Jedziemy dalej… :)

10.

Praktyczna ścieżka rozwoju - konkrety

W rozdziale „Do rzeczy: co powinieneś potrafić, by zacząć pracę jako programista?" opisałem, co musi wiedzieć i umieć osoba myśląca o tym zawodzie. Obiecałem również, że sposoby i źródła nauki omówię w dalszej części. Wspólnie dotarliśmy do tego punktu. Zaczynamy.

10.1. Język angielski

Znajomość języka angielskiego znacząco wpływa na szybkość robienia postępów w nauce programowania. Większość materiałów/dokumentacji jest w tym języku. Nie namawiam do intensywnych kursów językowych, według mnie najważniejszy jest bowiem kontakt z językiem i korzystanie z niego. Angielskiego radzę się uczyć na tej samej zasadzie co programowania: robić projekty i uzupełniać braki w wiedzy, gdy pojawi się taka potrzeba. Tak samo jest w przypadku wszystkiego innego.

Kontakt z językiem

Jak zapewnić sobie częstszy kontakt z językiem? Większość z nas nie ma okazji codziennie rozmawiać po angielsku, ale istnieje parę banalnych sztuczek:

- **Zmiana języka w telefonie**. Jak długo korzystamy z telefonu każdego dnia? Parędziesiąt minut. Czasami dobijamy do godziny. W skali miesiąca/roku uzbiera się sporo godzin. O ile nie jesteś ratownikiem medycznym albo nie pracujesz na innym podobnym stanowisku, w przypadku którego wolniejsze posługiwanie się telefonem może zdecydować o czyimś życiu lub śmierci, zmiana języka chyba nie jest dużym utrudnieniem. Przestaw język z wygodnego polskiego na angielski. Gdy nie będziesz wiedział, co oznacza dana funkcja, przetłumaczysz ją albo na chwilę zmienisz język na polski. Dzięki temu po paru tygodniach nabijesz sporo godzin kontaktu z językiem.

- **Zmień język, gdzie tylko się da**: w komputerze, ustawieniach konta Google, przeglądarce, każdej grze (nawet w Gothicu!). Zyskujesz dzięki temu kolejne godziny kontaktu z językiem. Korzystasz z niego na co dzień i sam wchodzi do głowy, bez konieczności poświęcania dodatkowego czasu na lekcje. Początki mogą być trudne – będziesz trochę wolniej posługiwał się urządzeniami, ale długofalowo to duży zysk.

- **Staraj się zastąpić lektora napisami.** Kolejne godziny w miesiącu, które można jakoś wykorzystać. Gdy tylko

masz wybór, sięgaj po napisy, a nie wersję z lektorem/dubbingiem. Najlepiej, gdybyś od razu oglądał film po angielsku, ale nic na siłę. Wybieranie wersji z napisami to dobry pierwszy krok. Rób tak systematycznie, a za parę lat będziesz bardziej osłuchany. To zresztą ogólna zasada: robić coś regularnie przez długi okres, a nie intensywnie przez tydzień, a potem wracać do złych starych nawyków.

- **Podcasty.** Koniecznie na temat, który Cię interesuje. Ja na przykład kocham grę Morrowind, a istnieje parę angielskojęzycznych podcastów prezentujących świat gry czy sztuczki, jak można dostać szybciej jakiś zajebisty jednoręczny topór. Podczas drogi do pracy/szkoły pobierz jakiś podcast i posłuchaj go zamiast muzyki. To nic, że wszystkiego nie zrozumiesz. Wybranie interesującego tematu powinno ułatwić skupienie się.

- **W rytuał czytania nowinek w sieci wpleć treść angielską. Na dobry początek można wybrać** coś z Reddita. Polecam: reddit.com/r/todayilearned/.

To tyle. Nigdy nie potrafiłem uczyć się języka przez naukę słówek. Kontakt z językiem przebija wszystko. Najważniejsze, by robić te rzeczy regularnie – za parę lat będą widoczne efekty.

Aaaa… i jeszcze jedno. **Gramatyka.** W przypadku angielskiego potrzebnego do nauki programowania gramatykę możesz olać. Nigdy nie przejmowałem się gramatyką i radziłem sobie dzięki intuicji nabytej dzięki kontaktowi

z językiem. Czy myślisz o polskiej gramatyce? No właśnie… Pewnie nie przywiązujesz do niej wagi, o ile nie jesteś polonistą (pozdrawiam polonistów, którzy to czytają! :)), którego irytuje masa błędów popełnionych przeze mnie w tej książce.

10.2. Umiejętność szukania informacji

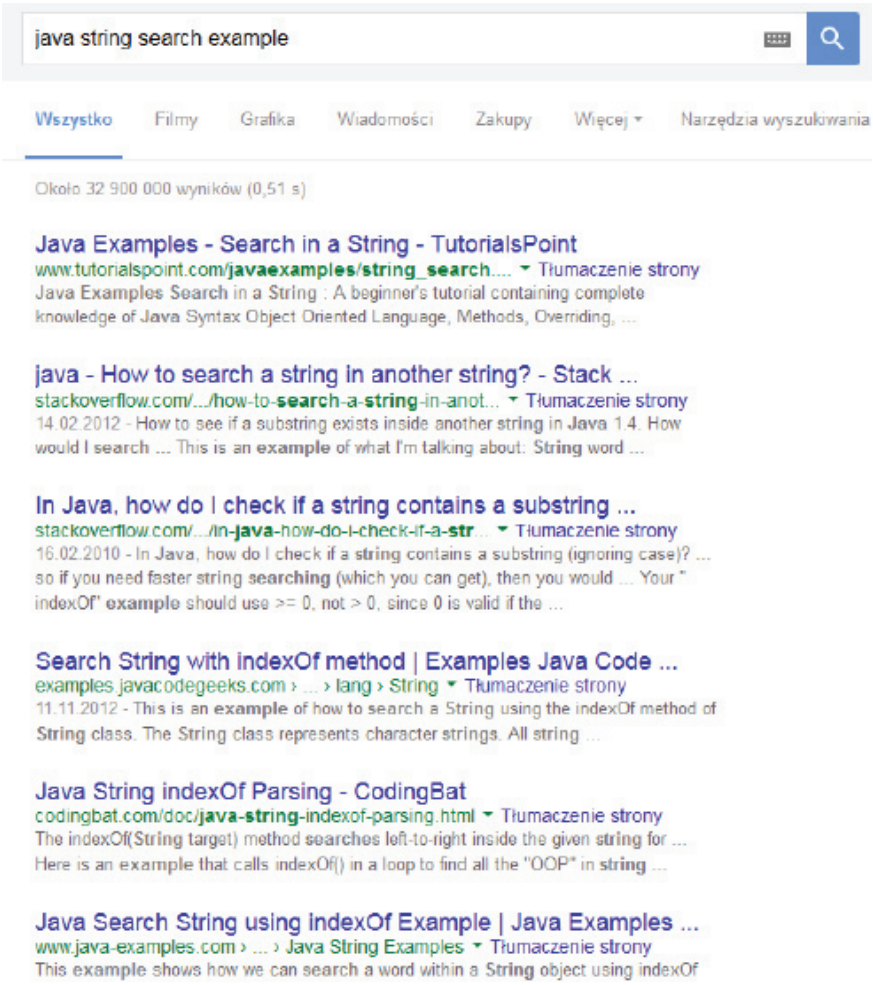
Po pierwsze: szukaj po angielsku. Albo przynajmniej się staraj i ucz na podstawie tego, jak poprawia Cię wyszukiwarka.

Po drugie: warto znać parę przykładów, by mieć lepszy obraz tego, jak szukać na potrzeby programowania.

10.3. Dopisywanie „example" do szukanej frazy

Załóżmy, że chcemy w programie (pisanym w Javie) zastosować szukanie w tekście. Wiemy, że tekst to „string", więc dobrze jest poszukać praktycznego przykładu. Czyli: „**java string search example**" – zaraz pojawia się fajna lista.

<screenshot wyników w Google na kolejnej stronie>


java string search example
Wszystko
Filmy
Grafika
Wiadomości
Zakupy
Więcej
Narzędzia wyszukiwania
Około 32 900 000 wyników (0,51 s)
Java Examples - Search in a String - TutorialsPoint
www.tutorialspoint.com/javaexamples/string_search.... Tłumaczenie strony
Java Examples Search in a String : A beginner's tutorial containing complete knowledge of Java Syntax Object Oriented Language, Methods, Overriding, ...
java - How to search a string in another string? - Stack ...
stackoverflow.com/.../how-to-search-a-string-in-anot... Tłumaczenie strony
14.02.2012 - How to see if a substring exists inside another string in Java 1.4. How would I search ... This is an example of what I'm talking about: String word ...
In Java, how do I check if a string contains a substring ...
stackoverflow.com/.../in-java-how-do-i-check-if-a-str... Tłumaczenie strony
16.02.2010 - In Java, how do I check if a string contains a substring (ignoring case)? ... so if you need faster string searching (which you can get), then you would ... Your " indexOf" example should use >= 0, not > 0, since 0 is valid if the ...
Search String with indexOf method | Examples Java Code ...
examples.javacodegeeks.com › ... › lang › String Tłumaczenie strony
11.11.2012 - This is an example of how to search a String using the indexOf method of String class. The String class represents character strings. All string ...
Java String indexOf Parsing - CodingBat
codingbat.com/doc/java-string-indexof-parsing.html Tłumaczenie strony
The indexOf(String target) method searches left-to-right inside the given string for ... Here is an example that calls indexOf() in a loop to find all the "OOP" in string ...
Java Search String using IndexOf Example | Java Examples ...
www.java-examples.com › ... › Java String Examples Tłumaczenie strony
This example shows how we can search a word within a String object using indexOf

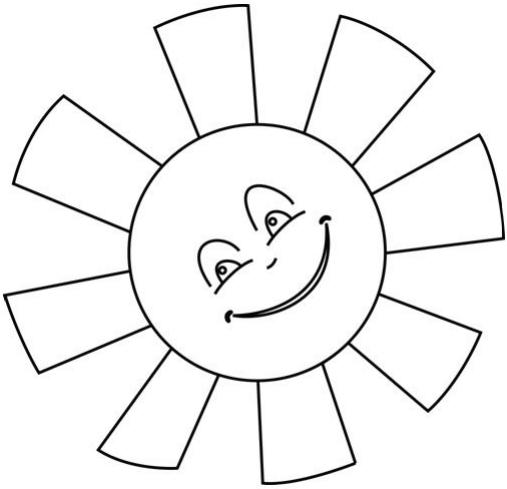
Dlaczego warto dodawać „example"? Może to kwestia osobistych upodobań, ale zawsze wolałem szukać w miarę krótkich i przejrzystych przykładów, jak coś zrobić. Zobaczyć, jak to działa, a teorię ewentualnie poznać później. Myślę, że wiele osób – podobnie jak ja – uczy się sprawniej na przykładach niż przez samą teorię.

10.4. Nie bój się szukać (nawet prostych rzeczy)

Kiedyś… dawno, dawno temu, gdy byłem młody (piękny) i głupi… myślałem, że dobry programista pisze wszystko z głowy. Że nie musi szukać w sieci rzeczy, którą już raz kiedyś zrobił. Można odnieść takie wrażenie, gdy widzi się starego wyjadacza, który siedzi w jakimś projekcie parę lat i wiele zadań wykonał już milion razy. Prawda jest jednak inna. Najważniejsze to wiedzieć, że coś istnieje i że da się to w danym języku zrobić, a szczegóły implementacji można zawsze wyszukać. Na przykład zapisywanie tekstu do pliku tekstowego: mniej więcej wiem, co i jak, ale nie napisałbym tego teraz z palca. W parę minut wyszukałbym jednak potrzebne informacje i sklecił coś na swoje potrzeby.

Strzelam, że winę za nieumiejętność uczenia się ponosi kiepski system edukacji. Nie pokazuje się nam, jak się uczyć, jak szukać informacji i wykorzystywać je w praktyce (nie mówiąc już o wykorzystaniu wiedzy, by stworzyć jakiś produkt, który sprawdzi się na rynku), tylko idzie na łatwiznę. Uczymy się na pamięć. Dzięki temu łatwo ma nauczyciel, który najpierw czyta na głos coś, co jest zapisane w książce (albo najlepiej dyktuje, by uczeń mógł to

zapisać – szczyt marnotrawienia czasu), a następnie wymaga, by uczeń na sprawdzianie przelał tę wiedzę na kartkę. Jaki kretyn wymyślił taki system nauki… Pamiętaj: nie jesteś już w szkole. Masz wykorzystywać wiedzę w praktyce, a nie uczyć się formułek po to, by jakaś stara baba mogła narysować uśmiechnięte słoneczko obok Twojego nazwiska.



10.5. Szukaj i nabieraj doświadczenia

Szukanie to zazwyczaj kwestia doświadczenia. Kopiujesz część komunikatu błędu i widzisz, że wyniki nie są tym, czego szukasz. Kopiujesz więc część błędu i dodajesz jakąś frazę, np. „error bla 123 + Android + network problems", pojawia się coś więcej. Następnym razem próbujesz jeszcze inaczej… Nie bój się sam szukać rozwiązania, zanim zwrócisz się o pomoc do kolegi (o ile masz kogo zapytać, np. będąc na stażu).

Umiejętność sprawnego szukania to w ponad 90% kwestia doświadczenia i nabytej w ten sposób intuicji. Warto jednak znać parę sztuczek szukania w Google.

10.6. Krótki kurs zaawansowanego szukania w Google

Istnieje parę sztuczek, które można wykorzystać w wyszukiwarce Google. Postanowiłem podzielić się **trzema najważniejszymi**, które stosuję na co dzień.

- Szukanie konkretnego zlepku wyrazów za pomocą cudzysłowu: **„”**. Jeśli wpiszemy w wyszukiwarkę frazę: java error, pojawią się inne wyniki niż dla frazy „java error” (czyli ujętej w cudzysłów). Szukane są wtedy wystąpienia całej połączonej frazy, a nie tylko wyniki z dwiema pojedynczymi frazami. Przydatne podczas szukania konkretnych komunikatów błędu.

- Site search, czyli szukanie na konkretnej stronie WWW/domenie. Szukanie w całej zawartości internetu może dawać śmieciowe wyniki. Warto wtedy ograniczyć szukanie do konkretnej strony, np. Stackoverflow.com, czy konkretnego forum, np. php-forum.com. W tym celu należy wpisać: **„site:www.strona-www.com szukana fraza”**, np. „site:stackoverflow.com Android arrayadapter”.

- Wyrzucanie wyników zawierających słowo, którego nie chcemy. Wystarczy wpisać to słowo, poprzedzając je myślnikiem/minusem: **-** Przykład: „game develop-

ment -java" – pojawią się wtedy wyniki dotyczące tworzenia gier, ale tylko na stronach, na których nie występuje słowo „java". Może to być przydatne, gdy np. szukamy informacji o JavaScripcie i wszędzie pojawiają się podpowiedzi, by wykorzystać jQuery, którego akurat w tym momencie nie chcemy. Wystarczy dodać do zapytania ciąg znaków „-jquery" i problem z głowy.

10.7. Rozbijanie zadań na mniejsze i praca nad jedną rzeczą naraz

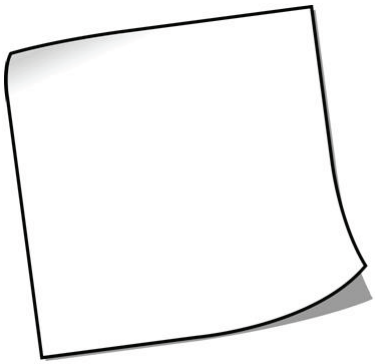
Czas, gdy możemy w skupieniu nad czymś pracować, jest dobrem ograniczonym. Umysł ludzki szybko się męczy i zniechęca, jeśli ma przed sobą zadanie, które wymaga wielu godzin/dni/miesięcy pracy. Dlatego należy nauczyć się dzielić duże zadania na kilka mniejszych i wykonywać je jedno po drugim.

Istnieje parę prostych sztuczek, którymi możemy się wspomóc w tej trudnej walce.

10.8. Karteczki samoprzylepne – popularne sticky notes

Po co one? Można przecież zrobić listę zadań i kolejno je rozpracowywać… niby tak. Jednak odkąd zacząłem stosować kartki samoprzylepne, praca stała się jakby lżejsza. Listę zadań często miałem przed sobą i za każdym razem, gdy na nią zerkałem, czułem przygnębienie, że jeszcze tak dużo muszę zrobić. W przypadku karteczek wygląda to trochę inaczej. Zadania można wypisać na karteczkach

i mieć na biurku tylko JEDNĄ taką kartkę. Skupić się, wykonać to jedno zadanie, następnie z satysfakcją zgnieść karteczkę i spróbować trafić do kosza. Widzimy efekt naszej pracy, a to działa bardzo pozytywnie na motywację.



Oto przykład zadania, które można sprawnie rozbić i nanieść na pojedyncze karteczki. Chcemy **poćwiczyć robienie aplikacji na Androida i planujemy program, która pobiera dane z webserwisu**. Zadanie można podzielić tak (planowanie przez rozpisywanie zadań na karteczki jest swoją drogą świetnym sposobem, by przemyśleć to, co chcemy zrobić):

- założenie projektu na GitHubie (by móc go w przyszłości pokazać albo po prostu mieć zapisany gdzieś kod – za parę miesięcy można do niego wrócić i zerknąć na przebieg pracy);

- stworzenie pustego projektu w IDE + ewentualnie skonfigurowanie środowiska pracy;

- wybór webserwisu, z którego chcemy pobrać dane (sama decyzja to też praca umysłowa, którą musimy wykonać; praca programisty to nie zawsze klepanie nowego kodu źródłowego);

- stworzenie pierwszego, prostego UI w ramach prototypu (w tym momencie jego wygląd jest kwestią drugorzędną – dopracowanie go to osobna karteczka);

- napisanie kodu, który potrafi wysłać pierwsze zapytanie do webserwisu;

- prezentacja wyników zapytania w ramach UI;

- obsługa błędów (co się stanie, jeśli np. webserwis zwróci błąd);

- pierwszy refactor kodu;

- poprawienie UI aplikacji;

- …cokolwiek innego, co chcemy dalej zaimplementować.

Dysponując taką listą zadań, łatwiej zabierzemy się do pracy: „Oo, nie mam teraz godziny, ale mam 20 minut, a to wystarczy, by założyć projekt na GitHubie i stworzyć pusty projekt w Android Studio" – zawsze to 20 minut, które później masz zaoszczędzone. Gdybyś nie rozbił zadania na części, pewnie byś nie zaczął – w końcu to tylko 20 minut do zagospodarowania. Gdy zada-

nie jest podzielone na etapy, całość nie wygląda już tak strasznie. Każdy etap powinien trwać od 10 minut do 2 godzin (jeśli zajmuje więcej czasu, to powinieneś **postarać** się go rozbić – o ile to oczywiście możliwe).

10.9. Trello – proste w obsłudze narzędzie do rozbijania zadań

Istnieją różne narzędzia do organizacji pracy. Ja prawie codziennie korzystam z Trello. Dla małych zastosowań jest darmowe i pozwala fajnie porządkować małe i średnie projekty. Przecież nie chcesz zatonąć w morzu karteczek…

Nie będę opisywał, jak korzystać z Trello, bo narzędzie jest intuicyjne i po paru minutach klikania powinieneś sobie poradzić. Warto założyć konto i popróbować z testowymi projektami. To, że zarządzałeś projektem w tego typu narzędziu, jest dobrym doświadczeniem, o którym możesz wspomnieć na rozmowie kwalifikacyjnej.

10.10. Ćwiczenie skupiania się/wyciszania – krótko o medytacji (możesz olać ten punkt)

Wszystko da się ćwiczyć, także umysł, by myśli nie uciekały ciągle w 100 tys. kierunków jednocześnie. Od niedawna poświęcam 20 minut dziennie na to, by starać się „nie myśleć". Ot, próbować przez jakiś czas się wyciszyć i nie dopuszczać do siebie żadnej myśli. Nie wiem, czy to efekt placebo, ale zauważyłem, że pracuję wydajniej i łatwiej jest mi się na czymś skupić.

Istnieje mnóstwo poradników poświęconych sztuce medytowania. Oto moje trzy reguły:

- **Nie możesz leżeć podczas medytacji** (bo zaśniesz), musisz siedzieć (obojętnie jak) lub stać.

- **Staraj się nie myśleć…** Łatwo powiedzieć. Medytacja to ćwiczenie skupiania umysłu i sesja, podczas której udaje się nie myśleć przez 5-20 sekund, też jest udana. Ćwiczymy ten „mięsień" i akceptujemy fakt, że nie wytrzymamy za długo.

- **Rób to regularnie.** Codziennie. Ten proces jest długofalowy. Lepiej próbować codziennie przez 5–10 minut (ustaw sobie timer) niż raz w tygodniu dłużej.

Jeśli spojrzeć na to z boku, medytacja jest najprostszą czynnością, jaką może robić człowiek: siedzieć w bezruchu, „nie myśleć" i ćwiczyć uspokajanie umysłu.

Możesz olać ten punkt, jeśli uważasz go za bezsensowny. Mnie te ćwiczenia pomagają, łatwiej jest mi się skoncentrować. Polecam.

10.11. Napisanie (nawet bardzo małego) projektu od początku do końca

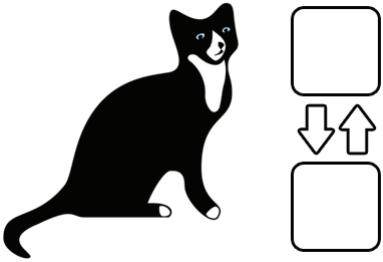
Twoja ścieżka rozwoju powinna być znaczona małymi projektami. Gdy napiszesz projekt od początku do końca, zyskasz świadomość, jak szybko może się rozrosnąć i że lepiej planować rozwój metodą małych kroków. Nie popadaj w pułapkę: nie twórz 100 pojedynczych projektów, w których nie będzie nic oprócz przepisanego na ślepo kodu z 10-minutowego tutorialu.

Dla osób, którym z trudem przychodzi wymyślanie projektów, przygotowałem przykłady. Zanim jednak do nich przejdziemy, po raz kolejny przypominam o konieczności założenia konta w serwisach GitHub/Bitbucket lub innych podobnych. Ułatwia to organizację pracy i daje łatwy wgląd do starych projektów.

Poniżej znajdziesz dwie propozycje prostych projektów, obydwa w pięciu wariantach, od najprostszego po bardziej skomplikowane. Pomysły są neutralne technologicznie, można je wykonać w prawie każdym języku programowania.

10.11.1. Manager schroniska dla zwierząt

Program pozwala na dodawanie/usuwanie z bazy zwierząt oraz sprawdzanie stanu schroniska (pełne, przepełnione, puste itd.)



Wariant I – bardzo prosty:

- Program konsolowy.
- Schronisko ma mieć określoną liczbę miejsc.
- Możliwość dodania/usunięcia zwierzaka do/z listy zwierząt w schronisku (na tym etapie lista nie musi być zapisywana do żadnej bazy danych ani pliku tekstowego).
- Po wpisaniu „status" program powinien wyświetlić aktualną listę zwierząt i wyrzucić komunikat, co oznacza ta liczba zwierząt: czy schronisko ma jeszcze miejsca, jest pełne czy przepełnione.

- Zaimplementowanie logiki, która uniemożliwia dodanie zwierzaka, gdy schronisko jest pełne.

Wariant II – to, co wyżej, plus:

- Zamiast programu konsolowego jest proste UI.
- Lista zwierząt jest zapisywana do pliku tekstowego lub bazy danych.
- Po wyłączeniu programu i ponownym odpaleniu ma zostać załadowany ostatni stan z bazy/pliku.

Wariant III – to, co wyżej, plus:

- Gdy w schronisku jest mniej niż pięć wolnych miejsc, wysyłany jest e-mail informacyjny do pracowników.
- Możliwość edycji poszczególnych zwierząt. Dodanie pól takich jak np. stan zdrowia, płeć itp.

Wariant IV – to, co wyżej, plus:

- Udostępnienie webserwisu, w którym można poznać aktualny status schroniska.

Wariant V – to, co wyżej, plus:

- Aplikacja mobilna (na dowolny system operacyjny), która korzysta z powyższego webserwisu.

- Zamiast aplikacji mobilnej może być też strona WWW. Ważne, by pobierała dane, korzystając z napisanego webserwisu.

Co jeszcze można dodać?

- Testy jednostkowe, które pokrywają najważniejszą funkcjonalność (nie martw się, jeśli nie znasz jeszcze niektórych pojęć, takich jak np. testy jednostkowe; zostaną one później krótko omówione).

- Eksport statusu schroniska do pliku .csv.

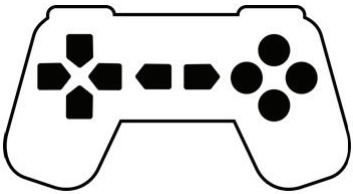
- Generowanie dziennego raportu ze stanu schroniska do pliku .pdf, który zostanie wysłany na określonego maila oraz dodany do jakiegoś archiwum.

- Możliwość dodawania zdjęć określonego zwierzaka przez aplikację mobilną (wymaga rozszerzenia webserwisu oraz aplikacji mobilnej).

Jak widać, nawet tak prosty program jak manager schroniska można poważnie rozbudować. Zachęcam Cię, abyś wymyślił coś własnego, zgodnie ze swoimi zainteresowaniami. Być może ten przykład będzie dla Ciebie inspiracją.

10.11.2. Pierwsza gra

Jeśli chcesz uczyć się programowania pod kątem tworzenia gier, oto propozycja: prosty przykład na pierwszą grę.



Wariant I – bardzo prosty:

- Okno z dowolnym napisem (np. tytuł gry).
- Gracz (jakiś prosty kwadrat), którym porusza się za pomocą strzałek w lewo, prawo, górę, dół. Prosta gra 2D z widokiem z góry.
- Przeszkody, z którymi gracz może kolidować.

Wariant II – to, co wyżej, plus:

- Przeciwnik, który porusza się w losowym kierunku, odbija od ścian i krawędzi ekranu.
- Licznik liczący, ile czasu spędziliśmy w grze.

- Pasek życia, który po kolizji z przeciwnikiem się zmniejsza.

- Efekty dźwiękowe po kolizji z przeszkodą/przeciwnikiem.

- Ekran „Game over", gdy pasek życia spadnie do zera.

Wariant III – to, co wyżej, plus:

- Prosta sztuczna inteligencja przeciwnika. Nie zmierza on już w losowym kierunku, tylko szuka gracza.

- Możliwość strzelania w przeciwnika.

Wariant IV – to, co wyżej, plus:

- Zapisywanie wyniku stanu gry do lokalnego pliku.

- Możliwość grania przez dwóch graczy na jednym komputerze (mogą się zwalczać lub walczyć razem z przeciwnikami).

- Urozmaicenie sztucznej inteligencji komputerowego przeciwnika.

Wariant V – to, co wyżej, plus:

- Opcja dwóch graczy przez sieć.

Inne pomysły?

Warto zaglądać na www.reddit.com/r/dailyprogrammer. Są tam dostępne różne zagadki programistyczne (od łatwych po trudne) i można poćwiczyć ich rozwiązywanie. Następnie metodą podobnej do tej powyżej da się je rozbudowywać. Zaczynamy od prostej aplikacji konsolowej, później dodajemy UI, zapisywanie danych, podpięcie webserwisu… możliwości jest sporo.

10.12. Skonfigurowanie środowiska pod projekt

Konfiguracja środowiska jest jednym z głównych czynników zniechęcających na starcie do programowania. W przypadku tworzenia stron WWW czy nauki JavaScriptu (dlatego też .js jest często polecany jako pierwszy język – bardzo łatwo zacząć) konfiguracja to jedynie przeglądarka i prosty Notatnik (czy Notepad++), natomiast w przypadku czegoś bardziej rozbudowanego, jak np. Android, Ruby on Rails czy Java EE, konfiguracja może wydawać się przytłaczająca. Warto ją więc przećwiczyć, by nabrać wprawy.

Na konfigurację środowiska składają się zazwyczaj:

- Instalacja IDE.
- Doinstalowanie technologii wymaganych przez projekt (np. możliwość kompilowania przez maszynę Ruby, Javy, Pythona itd.).
- Konfiguracja i podpięcie pod projekt lokalnej bazy danych.
- W przypadku pracy w firmie: udostępnienie różnych praw, by komputer mógł łączyć się z zabezpieczonym serwerem testowym.
- Wiele różnych drobnostek, na które na początku trudno wpaść.

Oto parę przykładowych rzeczy, które możesz wyszukać w sieci i spróbować lokalnie skonfigurować:

- Aplikacja na Androida, która korzysta z Google Maps.
- „Hello world" webserwisu z wykorzystaniem Javy, Eclipse lub IntelliJ, Tomcata i Springa. Podpowiedź: szukaj pod frazą „java Tomcat webservice Spring example". Lub spróbuj odpalić prosty webserwis w dowolnej innej technologii, którą wyszukasz.

- Skompiluj najprostszą możliwą grę z wykorzystaniem różnych możliwych technologii: Cocos2d-x, libGDX, Ogre3D, Unity pod Androida, Unreal Engine, Allegro (biblioteka, nie serwis aukcyjny) i C++, Allegro.js, cokolwiek innego, co wygooglasz z technologii do tworzenia gier.

- Aplikacja okienkowa napisana w dowolnym języku, która będzie korzystała z lokalnej bazy danych.

- Odpal przykładowe „Hello world" z wykorzystaniem dowolnego frameworka do PHP i postaraj się podpiąć lokalną bazę danych.

…mam nadzieję, że łapiesz. Przywykniesz do tego, że ciągle pojawiają się różne problemy wymagające częstych poszukiwań w Google, ale z każdym kolejnym znalezionym rozwiązaniem będziesz miał większe doświadczenie.

Problemy z konfiguracją występują PRAWIE ZAWSZE. Ciągle wyskakują mi błędy, gdy podpinam AdMoba (reklamy mobilne) do iOS-owej wersji mojej gry napisanej w libGDX. Trzeba do tego przywyknąć i nie dołować się niepotrzebnie, gdy konieczne będzie poświęcenie kolejnej godziny na naprawę jakiegoś głupiego błędu z IDE zamiast na programowanie. Wszyscy przez to przechodzą.

10.13. Zagadnienia, w których powinieneś się orientować

Są to:

- podstawy baz danych (ogólnie zapisywanie i czytanie danych),
- technologie webowe,
- protokoły komunikacyjne,
- programowanie obiektowe,
- testy i webserwisy (tworzenie i używanie).

Zazwyczaj konieczne będzie zgłębienie konkretnego obszaru wiedzy. Wystarczy jednak orientować się w podstawach, by umieć szukać i wiedzieć, o czym mówi kolega obok, specjalizujący się w innej działce.

Przy okazji przypominam, że ta książka jest drogowskazem dla osób pragnących zacząć przygodę z programowaniem. Opisałem w niej, co trzeba umieć, by aplikować do pierwszej pracy, i jak odnaleźć się na takim stanowisku. Natomiast szczegółów dotyczących samej implementacji tu nie znajdziesz. Odpowiedź, w jaki sposób wykonać dane zadanie, wyszukasz bez problemu w sieci.

10.13.1. Pliki z danymi, zapisywanie i czytanie danych

CSV

Bardzo często spotykam się z danymi zapisanymi w formacie .csv, znanym jako excelowy czy tabelkowy.

```
ID, Imię, Nazwisko
1, Jan, Nowak
2, Tomasz, Tomecki
```

Łatwo sobie wyobrazić, jak z takiego pliku powstaje tabelka w Excelu.

ID	Imię	Nazwisko
1	Jan	Nowak
2	Tomasz	Tomecki

Spróbuj zrobić prosty projekt, w którym odczytasz dane z takiego pliku, edytujesz je albo stworzysz całkowicie nowy plik z jakimś stanem programu.

*Jak to zrobić? Pomocna fraza: **„reading writing csv example” + technologia**: Java, Python, PHP itd.*

TXT

Zwykły plik tekstowy. Możesz spróbować odczytać z niego dane i zapisać stan programu do takiego pliku. Nie jest on zbyt często wykorzystywany w większych projektach (przynajmniej ja się z tym nie spotkałem), ale to najprostszy przykład pliku, z którym warto poćwiczyć.

PDF

Do PDF-a eksportujemy dane, by mieć je dostępne w czytelnej formie, gotowe do wysłania jako np. ładny załącznik do maila. Mało kiedy czyta się z niego dane (nie spotkałem się z tym, ale właśnie doczytałem, że jest to możliwe).

Bez problemu wyszukasz narzędzia/biblioteki, które ułatwiają eksportowanie danych jako PDF w Twojej technologii.

HTML

To forma strony WWW, którą można odczytać za pomocą dowolnej przeglądarki internetowej. Dobrym przykładem tak wygenerowanych plików są wszelkiego rodzaju raporty.

Umiejętność czytania (parsowania) takich plików daje ogromne możliwości. Pozwala wykorzystać w programie dane pobrane z różnych stron WWW.

Jak to zrobić? Pomocna fraza: „parsing HTML" + Twój język programowania: Java/Python/Ruby/PHP + „example"

10.13.2. Bazy danych

Z bazami danych wiąże się wiele zawodów, jednak Tobie wystarczy na początek opanowanie podstaw. Powinieneś umieć:

- założyć bazę danych z różnymi tabelami;
- dodawać dane do bazy;
- edytować dane w bazie;
- usuwać dane z bazy;
- podpiąć bazę danych pod technologię, z której korzystasz.

Jeśli raz uda Ci się to zrobić z DOWOLNĄ bazą danych w DOWOLNEJ technologii, powinieneś sobie poradzić również w każdej innej. To kwestia doczytania szczegółów i poświęcenia kilku godzin na próby i błędy.

Powinieneś już wiedzieć, jak wyszukać taką informację… „Java/Python/Ruby database example".

Istnieje milion rzeczy, których możesz się jeszcze nauczyć o bazach danych, ale zacznij od tego, co wyżej wymieniłem. Ważne jest, żebyś dodawanie, edytowanie, usuwanie potrafił wykonać z poziomu programu, który właśnie piszesz, a nie tylko bezpośrednimi zapytaniami na bazie.

10.13.3. Podstawowe technologie webowe

HTML, CSS i JavaScript (z ewentualnym wsparciem ze strony jQuery lub AngularJS) są proste w nauce i bardzo pomocne. Nie wyobrażam sobie, by programista nie wiedział, jak stworzyć najprostszą stronę WWW za pomocą HTML-a, CSS-a i jakiejś małej sztuczki w JavaScripcie.

Przydaje się to szczególnie do testowania jakiegoś prostego front-endu oprogramowania.

HTML

W jakim stopniu musisz go opanować? Wystarczy, że wiesz, czym są sekcje HEAD, BODY, jak tworzy się linki do innych stron, znasz podstawy prezentacyjne (pogrubienie tekstu, podkreślenie), wiesz, co to paragraf, div, jak zrobić tabelkę do przedstawienia danych… Resztę doczytasz, gdy pojawi się taka potrzeba.

W jeden dzień ogarniesz, o co w tym chodzi, i stworzysz pierwszą prostą stronę WWW. Serio, to bardzo proste.

CSS

HTML definiuje strukturę dokumentu, a CSS opisuje, jak ten dokument ma wyglądać. Dokument HTML zyskuje dzięki temu na przejrzystości, bo całość (albo przynajmniej większość) informacji na temat wyglądu dokumentu znajduje się w osobnym pliku .css.

Szukaj pod frazą „css crash course", by zorientować się, jak CSS działa w połączeniu z HTML-em.

Na bieżąco szukaj tego, czego potrzebujesz. Na razie postaraj się tylko zrozumieć, jak to działa – intensywna nauka CSS-a mija się z celem, jeśli nie zamierzasz profesjonalnie tworzyć stron WWW albo być ekspertem od front-endu.

JavaScript / jQuery

Do dynamicznych zmian na stronie WWW służy JavaScript. Nie ma on nic wspólnego z Javą – zbieżność nazw wynika z tego, że powstały w podobnym okresie i w celach marketingowych chciano podpiąć się pod hype Javy…

Jest to potężne narzędzie, w którym można tworzyć zaawansowane gry. Jako początkujący powinieneś:

- wiedzieć, jak zamieścić .js skrypt na stronie WWW;
- zerknąć na dowolny „JavaScript crash course”;
- spróbować wykorzystać jakiś istniejący webserwis za pomocą jQuery – naprawdę fajne doświadczenie, pomaga ogarnąć big picture wokół tego wszystkiego.

10.13.4. Protokoły komunikacyjne

Prawdopodobnie najwięcej będziesz miał do czynienia z **HTTP i HTTPS**.

Możesz obejrzeć kilka filmików na YouTube na ich temat, poznać teorię, jeśli chcesz… Jednak najważniejsza jest próba wykorzystania ich w praktyce:

Jak to zrobić? Pomocna fraza: „**http request example**" + Java/C#/Python/Android itd.

Spróbuj zaimplementować kilka przykładowych requestów (zapytań) w swoim programie. Jeśli jesteś ambitny, to możesz skierować te zapytania na własny serwer, na którym zamieściłeś np. swój webserwis.

FTP

Protokół do transferu plików z Twojego komputera (czy komputera klienta) na serwer i odwrotnie. Zazwyczaj użytkownik potrzebuje loginu i hasła, by mieć dostęp do takiego serwera.

W pracy jako programista wykorzystywałem przez 95% czasu komunikację po HTTP/HTTPS, więc jeśli masz ograniczony czas, to głównie na nim powinieneś się skupić. Warto jednak przerobić przykład z protokołem FTP, by dowiedzieć się, jakie oferuje możliwości.

Pomocna fraza: Java/C#/PHP + „FTP example".

Przykładem programu korzystającego z takiego protokołu jest popularny Total Commander.

POP i SMTP

POP (Post Office Protocol) i SMTP (Simple Mail Transfer Protocol) to protokoły do poczty elektronicznej.

Za ich pomocą można pisać programy pocztowe (no shit, Sherlock) i stworzyć wszelką funkcjonalność dotyczą wysyłania/odbierania poczty elektronicznej.

Tym razem nie podpowiem, jak szukać materiałów. Powinieneś to już wiedzieć :)

W ramach nauki możesz napisać program do odbierania maili. **To świetny przykład do nauki programowania aplikacji mobilnych.** Aplikacja do odbierania maili to świetny kandydat do projektu do CV na młodszego Android developera.

10.13.5. Programowanie obiektowe

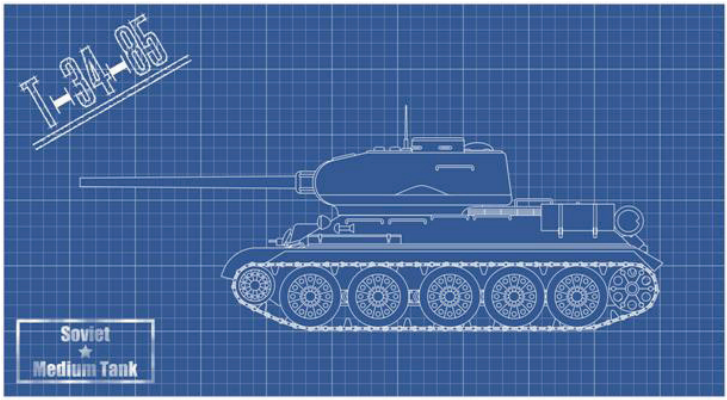
Programowanie obiektowe to dziwny stwór… na początku kompletnie mnie od niego odrzucało i nauka wydawała się bardzo męcząca. Po jakimś czasie przeskoczył mi w głowie

jakiś przełącznik i nagle stało się to „fajne i oczywiste" i nie wyobrażam sobie już programowania nieobiektowego.

Postaram się odtworzyć proces myślowy, który sprawił, że wreszcie zrozumiałem, z czym to się je. **Często irytowało mnie, że mylę obiekt z klasą.** Nie mogłem tych dwóch pojęć logicznie w głowie z niczym połączyć. Gdy przeczytałem, czym się różnią, pamiętałem to przez chwilę, a za parę dni znowu myliłem pojęcia.

Przełom nastąpił, gdy przeczytałem (czy usłyszałem, nie pamiętam) o porównaniu z czołgiem. Wtedy zaświeciła mi się w głowie jakaś lampka.

Widzisz ten plan czołgu? Nie jest to faktyczny **OBIEKT**, ale plan stworzenia takiego obiektu! Ten plan to **klasa**. Pokazuje, jak stworzyć obiekt, ale nie jest to fizyczny obiekt.



Booooooom! To porównanie zostało w mojej głowie już na zawsze i przestałem mylić pojęcia.

To, co piszemy (klasy), to plany obiektów, które później można w jakiś sposób zbudować w programie. Nie zamierzam tworzyć kolejnego kursu programowania obiektowego – kluczem do nauki jest praktyka. Możesz poznać teorię, ale najlepiej uczyć się na przykładach. **Nie ma magicznego sposobu. Niestety.** Rób własne projekty, pokazuj kod, czytaj kod innych osób, pisz kolejny projekt. Wszystko to kwestia doświadczenia.

W rozdziale 6 wymieniłem, jakie pojęcia związane z programowaniem obiektowym trzeba dobrze znać, by móc komunikować się z innymi programistami. Przypominam je:

- obiekt,
- klasa,
- klasa abstrakcyjna,
- interfejs,
- konstruktor,
- dziedziczenie,
- kompozycja,

- metoda,
- wzorzec projektowy.

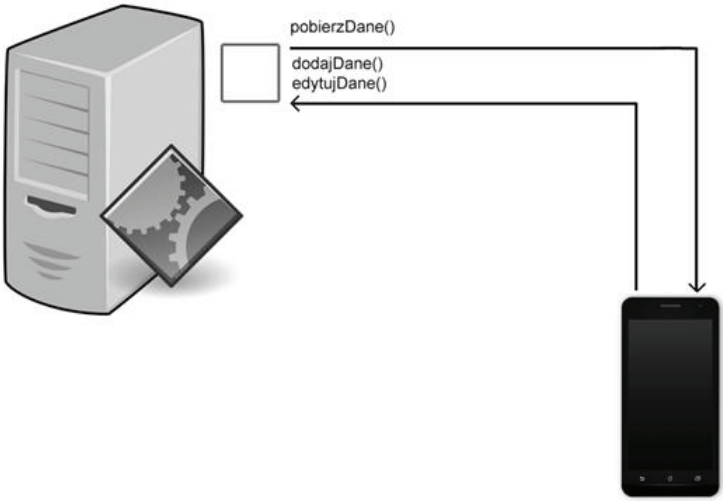
Powinieneś potrafić zastosować je praktyce, a nie tylko wyjaśnić. Jeśli jeszcze tego nie opanowałeś, to już wiesz, do czego się zabrać. Musisz intuicyjnie wiedzieć coś o każdym z tych punktów.

10.13.6. Web serwisy

Gdy zaczynałem w pierwszej pracy jako programista, nigdy wcześniej o czymś takim jak webserwis nie słyszałem. Głupi ja. To znaczy wiedziałem, że istnieje coś z taką funkcjonalnością, jaką ma webserwis, ale nie umiałem przyporządkować tej nazwy do funkcjonalności.

Dlatego zacznę od prostego wyjaśnienia, czym jest webserwis. To **usługa sieciowa**, tyle że **usługa dla programisty**, napisanego przez niego oprogramowania, a nie dla użytkownika. Usługą dla użytkownika byłoby wejście na stronę WWW, graficzne wyświetlenie zawartości i zapewnienie możliwości korzystania z portalu. W przypadku usługi sieciowej (webserwisu) jest to pewien adres (URL) w sieci, za pomocą którego można korzystać z danej usługi (np. pobrać, edytować, usunąć dane).

Najlepiej pokazać to za pomocą ilustracji.



Na co zwrócić uwagę: aplikacja w telefonie, która korzysta z usługi sieciowej (webserwisu) na serwerze, **otrzymuje dane w formacie neutralnym technologicznie** (zazwyczaj JSON lub XML) i może je w dowolny sposób przetworzyć.

Zamiast aplikacji w telefonie **można wstawić cokolwiek**: aplikację konsolową, lodówkę, maszynkę do kawy z wi-fi – mieliśmy taką w pracy, ale odłączyli nam tę funkcjonalność, bo bali się, że za dużo namieszamy. :(

Co powinieneś umieć i czego spodziewać się w pracy?

- Korzystanie z webserwisów (czyli pisanie **aplikacji klienckiej** – jak ten telefon na powyższej grafice. Jest on klientem serwera, który zapewnia usługę sieciową).

- Pisanie usługi, czyli budowanie funkcjonalności na serwerze, **z której będą mogli korzystać klienci**: telefony, lodówki i wszystko, co można zaprogramować.

Oto podpowiedź, co możesz wyszukać, by poćwiczyć na przykładzie:

- „rest service **client** Java/C#/Android/Python example" lub „web service **client** Java/C#/Android/Python example" – by napisać prostą przykładową aplikację klienta, która korzysta z usługi sieciowej. Szukaj przykładu napisania **klienta (client)** usługi sieciowej, a nie pisania samej usługi, która moim zdaniem jest trudniejsza do zrozumienia.

- Gdy pojmiesz, jak działa klient, spróbuj napisać samą usługę. To bardzo rozległy temat, ale na początek wyszukaj „rest service example" + coś z technologii, którą chcesz poćwiczyć „Spring, Java, C#…".

Ważne, by zacząć na maksymalnie prostych przykładach, które można w miarę samodzielnie modyfikować. Na przykład w przypadku **klienta** wystarczy, że napisze aplikację, która pobiera aktualną pogodę. W przypadku pisania **usługi** wystarczy zwrócenie twardo zakodowanej liczby, później może jakichś prostych danych pobranych z bazy.

Istnieje świetny kanał na YouTube, który tłumaczy usługi sieciowe: **Java Brains**. Więcej wartościowych materiałów

znajdziesz w rozdziale „Z jakich źródeł się uczyć? Polecane strony/materiały".

10.13.7. Testy jednostkowe

Pewnie obiło Ci się o uszy, że istnieje coś takiego jak pisanie testów. Na początku nie zawracałbym sobie nimi głowy – i tak będziesz przytłoczony nadmiarem informacji. Na tym etapie musisz wiedzieć jedynie to:

- Projekt zawierający testy jest uważany za projekt z lepszej jakości kodem.

- Testy pokazują swoją wartość **długofalowo** – informują, że popsuła się jakaś stara funkcjonalność, bez konieczności jej ręcznego przetestowania.

- Ich pisanie **często zajmuje tyle samo czasu, ile pisanie właściwego kodu**, przez co wydaje się, że wolniej robimy postępy. Jednak w przypadku dużego (wieloletniego) projektu warto poświęcić na to czas.

- Istnieje coś takiego jak **TDD** (test-driven development), czyli proces tworzenia oprogramowania, w którym najpierw pisze się testy, a później implementację. Nie polecam tego początkującym – jako junior szybko się zniechęcisz. Zainteresuj się tym, gdy nabędziesz już trochę doświadczenia. Na razie musisz tylko wiedzieć, że coś takiego istnieje.

- Warto znać pojęcie **code coverage**, które określa, ile procent kodu ma pokrycie w testach.

- Pisanie testów jest pracochłonne, więc jeśli się na to decydujemy, powinniśmy **świadomie określić, gdzie testy są najbardziej potrzebne**. W idealnym świecie z nieskończoną ilością czasu wszystko byłoby pokryte testami, jednak w rzeczywistości budżety projektów są ograniczone i nie można wiecznie rozwijać projektu w piwnicy.

Jak szukać prostych przykładów do nauki pisania testów?

„Unit test example ” + technologia: Java, C#, JavaScript, Android.

10.13.8. Parsowanie danych JSON i XML

Parsowanie to przetworzenie danych z neutralnego technologicznie formatu (zazwyczaj JSON lub XML) na lokalny format danych. Za przykład niech posłuży prosty XML:

```
<klienci>
	<klient id=”1”>
		<imie>Jan</imie>
		<nazwisko>Kowalski</nazwisko>
	</klient>
	<klient id=”2”>
		<imie>Tomasz</imie>
		<nazwisko>Tomecki</nazwisko>
	</klient>
</klienci>
```

Parsowanie takiego XML-a polega na pobraniu go przez program i przekształceniu danych w lokalne obiekty. W tym przypadku będzie to lista klientów, na której znajdują się obiekty „Klient" z polami: id, imię i nazwisko. Proste, prawda?

By spróbować przykładów z konkretną wybraną technologią, poszukaj:

XML parsing example + technologia

JSON parsing example + technologia

Na początku wyda się to męczące, ale z czasem przywykniesz. Parsowanie danych jest standardowym zadaniem, wykonywanym bardzo często.

0.13.9. Dodawanie bibliotek i budowanie projektu

W przypadku Javy dodanie pliku .jar z biblioteką, dodanie biblioteki z wykorzystaniem Maven, Gradle, ewentualnie Ant… to niby banał, ale warto zgłębić temat.

Jeśli używasz innych technologii, szukaj pod frazą:
adding external library + technologia.

Na czym polega budowanie projektu? Gdy projekt ma dużo zewnętrznych zależności (jak np. biblioteki), to budowanie („odpalenie”/„włączenie”) może się niepotrzebnie skomplikować. Warto poczytać o budowaniu projektu w takich technologiach jak Maven czy Gradle.

Pomocna fraza: building project with + technologia (np. Maven).

10.13.10. Praca z bugami i proces debugowania

O radzeniu sobie z bugami można by napisać osobną książkę (i parę zostało napisanych), skupmy się jednak na tym, co powinieneś wiedzieć jako początkujący. Bugi w programach są nieuniknione. Jak do nich podchodzić, jak są raportowane i jak skutecznie szukać ich przyczyny (debugowanie)?

Zależnie od technologii, w której pracujesz, może to wyglądać trochę inaczej, ale podstawy są wszędzie prawie takie same. Jeśli korzystasz z debuggera (pomocne narzędzie w IDE do szukania błędów), powinieneś znać następujące pojęcia:

- Breakpoint – miejsce, w którym zatrzyma się wykonywanie programu. Sam możesz zdecydować, w którym miejscu program się zatrzyma, by np. sprawdzić aktualne wartości w zmiennych.

- Step Over – pojęcie związane z nawigacją w debuggerze. Pozwala przeskoczyć o jedną linijkę kodu niżej.

- Step Into – kolejne pojęcie związane z nawigacją – wskakujemy wewnątrz jakiejś metody.

- Step Out/Return – odwrotność powyższego – wychodzimy z metody.

Znaczenie tych pojęć najlepiej byłoby poznać w praktyce. Poszukaj dowolnego tutorialu o debugowaniu w wybranej technologii i spróbuj swoich sił. Szukania błędów (debugowania) uczymy się w praktyce, dlatego pisz, jeszcze raz pisz, popełniaj błędy i szukaj ich źródeł.

W przypadku programowania webowego należałoby znać najpopularniejsze sposoby debugowania z poziomu przeglądarki, jak np. Firebug czy Chrome Developer Tools.

Więcej o procesie raportowania i codziennej pracy z bugami znajdziesz w podrozdziale 12.4. „Bugfixing/debugging – szukanie i naprawianie błędów".

10.13.11. Praca z narzędziem kontroli wersji, np. Gitem

Żebyś sprawnie komunikował się w zespole i poznał z grubsza pojęcia związane z Gitem, przygotowałem poniższą listę. Znalazły się na niej najważniejsze pojęcia

wytłumaczone w bardzo uproszczony sposób (szczegóły doczytasz w dokumentacji i zrozumiesz, na czym polega stosowanie Gita w praktyce – w żadnym wypadku nie traktuj poniższych wyjaśnień jak oficjalnych definicji!).

- **Clone** – pobranie/sklonowanie projektu na dysk do jakiegoś folderu.

- **Branch** – gałąź projektu. Projekt ma różne rozgałęzienia, co pozwala swobodnie pracować nad nową funkcjonalnością w zespole (lub lepiej zorganizować pracę samemu sobie).

- **Checkout** – zmiana branchu, gałęzi projektu.

- **Plik .gitignore** – plik z podanymi typami plików lub konkretnymi plikami, które Git ignoruje. Dzięki temu **repozytorium** (miejsce, w którym przechowywane są pliki) jest bardziej przejrzyste.

- **Commit** – wprowadzenie, „zapisanie" zmiany w lokalnym repozytorium (ważne: na zewnętrznym serwerze zmiana będzie widoczna dopiero po wykonaniu **pusha**).

- **Push** – wysłanie „zakommitowanych" zmian na serwer.

- **Pull** – pobranie zmian z serwera (np. gdy kolega coś wrzucił i chcemy to pobrać, robimy to przez **pull**).

- **merg** „mergowanie" – rozwiązywanie konfliktów w plikach: taki konflikt powstaje, gdy np. dwóch programistów wykonało zmianę w tej samej linii kodu i Git nie jest w stanie sam rozstrzygnąć, co powinno się znajdować w danym miejscu.

W książce nieustannie powtarzam, by wrzucać swoje projekty/postępy na GitHuba/Bitbucket czy w inne podobne miejsce.

To jest właśnie nauka Gita przez praktykę. Polecam zaopatrzyć się w **SourceTree** – świetny program do wizualnej pracy z Gitem. Oprócz SourceTree warto doinstalować **DiffMerge, wspaniałe** narzędzie do rozwiązywania konfliktów po mergowaniu.

A oto przykład tego, co powinieneś zrobić, by poczuć się pewniej w pracy z Gitem:

- Na GitHubie wybierz dowolny stworzony przez siebie projekt (jeśli jeszcze go nie masz, to do dzieła!) i pobierz go na drugi komputer (ostatecznie w inne miejsce na dysku, jeśli nie masz dostępu do drugiego komputera).

- Stwórz na drugim komputerze osobny branch pobranego projektu.

Nie wiesz, jak to zrobić? Oto pomocne frazy: „git making a branch" lub „git source tree making a branch".

- Wprowadź drobne zmiany w branchu. Następnie wykonaj commit i push.

- Na pierwszym komputerze zrób checkout nowego branchu i spróbuj go „wmergować" w ostatni stan projektu.

- Wspomagaj się internetem, jeśli nie potrafisz wykonać któregoś z tych zadań.

Jeśli uda Ci się zrealizować wszystkie kroki, to świetnie! Masz już przedsmak tego, jak wygląda praca w grupie korzystającej z Gita.

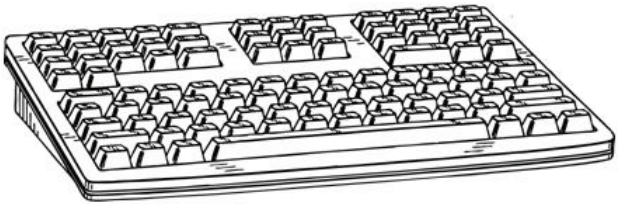
Byłoby idealnie, gdybyś robił z kimś projekt i korzystał z Gita. Rozwiązując napotkane problemy, sporo się nauczycie.

10.14. Usprawnianie pracy w środowisku programistycznym

Od osób rekrutujących programistów często słyszałem, że bardzo dobre wrażenie robiło na nich to, jak sprawnie kandydat korzystał z IDE (lub dowolnego innego narzędzia używanego w codziennej pracy). Nie chodziło o ślepo wyuczone skróty klawiszowe, dzięki którym potencjalny pracownik umiał obyć się bez myszki, ale o płynność i efektywność pracy takiego człowieka.

Jak posiąść tę umiejętność? Po pierwsze, jest to proces długofalowy, kurs czy artykuł nie załatwią sprawy. Polega na wyłapywaniu czynności, które często wykonujemy, i wprowadzaniu drobnych usprawnień, dzięki którym zyskujemy parę sekund.

Przykład? Piszę tego e-booka/książkę w przeglądarce w WordPressie. W ostatnich dniach non stop korzystam z przeglądarki, pisząc kolejne zdania i robiąc mniejszy/większy research w sieci. Bardzo często zmieniam tab (zakładkę) z WordPressem na inny (Ctrl + Tab) i wpisuję coś w pasek adresu (przejście do sekcji wpisywania adresu to Ctrl + L). Dzięki skrótom nie odrywam dwóch rąk od klawiatury (nie muszę sięgać po myszkę), a co za tym idzie – pracuję sprawniej. Dodatkowo kupiłem sobie klawiaturę, która po prawej stronie ma wbudowany touchpad. Jeśli potrzebuję coś szybko przesunąć/zaznaczyć, korzystam z touchpada. Po myszkę sięgam dopiero wtedy, gdy potrzebuję większej precyzji, np. przy obróbce graficznej obrazka. To kwestia przyzwyczajenia i pewnie wiele osób dziwi się, że lubię używać touchpada (kocham touchpada na macu – w ogóle nie potrzebuję wtedy myszki). **Ja** dzięki niemu sprawniej pracuję, a co okaże się skuteczne w **Twoim** przypadku – musisz sam odkryć.



Zawsze powinieneś szukać drobnych usprawnień w sposobie korzystania z narzędzi. Załóżmy, że w każdym miesiącu wprowadzisz jedną zmianę, dzięki której będziesz pracować o 1% wydajniej. Po roku Twoja wydajność wzrośnie na tyle, że w ciągu miesiąca będziesz wykonywał o 10% pracy więcej.

Oto parę przykładowych usprawnień, które uważam za sensowne podczas pracy w IDE (skrót klawiszowy danej czynności w swoim IDE wyszukasz bez problemu sam):

- **Call Hierarchy** – gdzie metoda jest wywoływana.
- **Quick Outline** – pokazuje listę metod/pól w klasie, w których można szukać.
- **Refactor metody/method refactor** – zmiana nazwy metody/argumentów we wszystkich miejscach w projekcie.
- **Open Resource** – wyszukanie, którą klasę lub który zasób (np. plik .xml) otworzyć.
- **Quick fix** – popularne np. w Eclipse „Ctrl + 1", które pozwala przykładowo na szybkie przypisanie wyniku wyrażenia do zmiennej lokalnej lub zapewnia szybki dostęp do paru innych popularnych akcji.
- Przesunięcie linii kodu w górę/dół.

- Szybkie usunięcie linii kodu.

- Zakomentowanie linii kodu itd.

Usprawnienia dobrze jest wprowadzać po jednym. Jeśli zechcesz korzystać ze wszystkich jednocześnie, w praktyce nie będziesz stosował żadnego. Zawsze możesz podpytać kolegów, jakimi skrótami/usprawnieniami oni najczęściej się posługują.

10.15. Podsumowanie

Za Tobą dużo materiału. Jeśli przeczytałeś całość ze zrozumieniem i wykonałeś parę ćwiczeń, to powinien Ci się powoli rysować przed oczami obraz pracy programisty. Mam nadzieję, że znasz już zarówno swoje braki, jak i mocne strony.

W kolejnym rozdziale podałem kilka źródeł, z których można się uczyć programowania.

11.

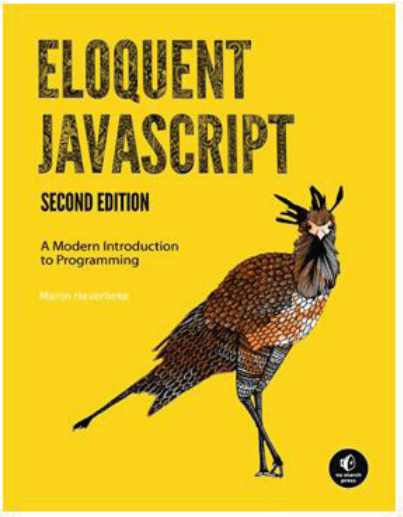
Z jakich źródeł się uczyć? Polecane strony/materiały

W sieci można znaleźć mnóstwo materiałów, które obszernie opisują różne języki programowania. Trudno jednak znaleźć takie, w których wiedza podana jest w przystępny sposób. Zawsze pozostaje sięgnięcie do oficjalnej dokumentacji, będącej dobrym źródłem… dla osoby z doświadczeniem. Zieloni w programowaniu tylko się zniechęcą.

Ten rozdział zawiera propozycje przystępnych materiałów do samodzielnej nauki. Są to źródła, z którymi miałem do czynienia i które polecam (nie zaproponowałem materiałów do wszystkich języków programowania, a jedynie te, z których korzystałem).

11.1. Najlepszy start w programowanie – świetna (i darmowa!) książka

Istnieje język programowania, do którego nie trzeba specjalnie nic konfigurować. Wystarczy prosty edytor tekstowy. Oto JavaScript.



„Eloquent JavaScript", Marijn Haverbeke

„Eloquent JavaScript" jest w mojej opinii jednym z najlepszych możliwych wstępów do programowania. Tym bardziej cieszy fakt, że książkę można czytać w sieci za darmo: http://eloquentjavascript.net/.

Dlaczego dostęp jest bezpłatny? Wszystko kosztuje: książka powstała dzięki liście sponsorów (http://eloquentjava-

script.net/backers.html). Sama **Fundacja Mozilla dała na ten cel 10 000 $** – potrzeba lepszej rekomendacji?

11.2. Początki w Javie

Cave of Programming – John Purcell

Uważam, że zaczynanie od książki jest jednak trochę męczące. Zawsze wolałem zobaczyć konkretną czynność w IDE i samodzielnie ją powtórzyć. Dlatego na start polecam mój ulubiony kanał YouTube uczący programowania: Cave of Programming – John Purcell, youtube.com/caveofprogramming.

Podziwiam, ile pracy autor włożył w ten kanał. Purcell oferuje również płatne kursy na Udemy – warte swojej ceny. A co można znaleźć w darmowych materiałach?

- Java dla początkujących.
- Java Collections Framework (kolekcje)
- Wielowątkowość w Javie.

- Wzorce projektowe w Javie.
- C++ dla początkujących (tego kursu nie oglądałem).
- Wstępy do płatnych kursów (Android, Swing, Spring).

Co zawierają płatne materiały premium?

- Kurs Androida.
- Kurs Java Swing (aplikacje desktopowe w Javie).
- Kurs Java z wykorzystaniem Springa.
- Java z Servletami i JSP (tworzenie stron w Javie).
- Kurs Perl 5.

Derek Banas

Czas na mój drugi ulubiony kanał na YT: Derek Banas, https://www.youtube.com/derekbanas. Kanał równie dobry jak Cave of Programming – może ciut bardziej chaotyczny. Brakuje mi usystematyzowania takiego jak w przypadku kanału Johna, jednak jest to zdecydowanie jeden z topowych kanałów do nauki programowania.

https://www.youtube.com/user/derekbanas

Oprócz programowania Derek uczy również, jak gotować niskokaloryczne potrawy. ;)

Na kanale znajdziecie następujące materiały dotyczące programowania:

- Kurs Javy.
- Android.
- Objective-C.
- PHP.
- XML.
- JavaScript.
- Python.
- SQL/SQL Lite/Mongo DB (**bazy danych**).
- WordPress.

- **Programowanie obiektowe.**
- Ajax.
- NodeJS.
- CSS.
- Ruby on Rails.
- jQuery.
- **Webserwisy w PHP.**
- C.
- Git.
- …I pewnie trochę pominąłem.

To skarbnica wiedzy, jednak należy traktować te kursy jako wstęp do samodzielnej nauki. Programowanie to przede wszystkim pisanie kodu źródłowego i samodzielne rozwiązywanie problemów.

Żeby pisać, trzeba pisać – nie popadaj w pułapkę ślepego przepisywania tutoriali. Po paru tutorialach postaraj się zastosować nabytą wiedzę w prostym projekcie! KONIECZNIE! Nie popełniaj błędu siedzenia w jaskini.

PHP Academy

Kolejnym świetnym kanał na YT jest PHP Academy, https://www.youtube.com/phpacademy. Wiedzie od podstaw PHP po naukę frameworków na wielu praktycznych przykładach. Bardzo wartościowe źródło.

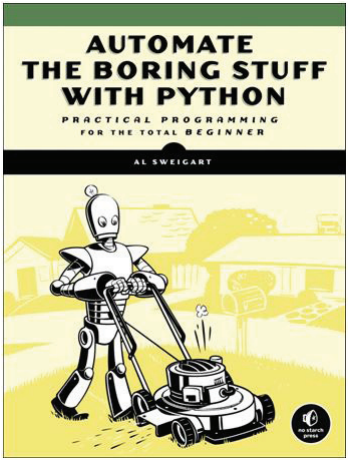
11.3. Krótkie i przyjemne wprowadzenie do Ruby (po polsku)

Jeśli zamierzasz zainteresować się Ruby, to ten materiał jest świetny. Ruby w 20 minut: https://ruby-lang.org/pl/documentation/quickstart/.

11.4. Dobra książka do C++

Przerabiałem częściowo książkę **„C++ Primer Plus”** Stephena Praty, którą szczerze polecam jako solidne usystematyzowanie wiedzy. Jak już wspomniałem, wolę zaczynać naukę nie od książek, tylko od filmów: zobaczyć, jak wszystko działa w akcji, a dopiero później przejść do książki. W moim przypadku się to sprawdza.

11.5. Źródła do nauki Pythona



„Automate the Boring Stuff with Python”, Al Sweigart

Książka wydaje się przystępna, jej autor postawił na praktykę. Jeśli zawsze chciałeś spróbować swoich sił w Pythonie, to poświęć jej trochę czasu. Wersję elektroniczną można za darmo czytać tutaj: https://automatetheboringstuff.com.

Sentdex – kanał na YouTube

Autor tego kanału bardzo się przykłada, by przystępnie wyłożyć wiele zagadnień związanych z Pythonem. Link do kanału: https://www.youtube.com/sentdex.

Na poszczególnych playlistach można znaleźć konkretne tematy, np. jak korzystać z baz danych w Pythonie – nie

tylko kod źródłowy, ale również wszystkie narzędzia związane z otoczką danej technologii.

Blog MiguelGrinberg.com – praktyczna nauka Flaska

Blog zawiera megatutorial Flaska (framework webowy), w którym zostało wyjaśnione, jak po kolei wszystko konfigurować: **www.juniordeveloper.pl/flask-confi**g.

Drugi polecany tutorial na tym blogu dotyczy webserwisów RESTO-wych. Zdziwisz się, jak mało kodu trzeba napisać, aby powstał funkcjonalny webserwis: **www.juniordeveloper.pl/flask-rest**.

The Hitchhiker's Guide to Python!

Gdy znasz już podstawy Pythona i zastanawiasz się, co dalej, ta strona będzie dobrym przewodnikiem: http://docs.python-guide.org/. Dowiesz się z niej, jak strukturować kod, co wykorzystywać do aplikacji webowych, do GUI, baz danych, parsowania… każdy znajdzie coś dla siebie. Tę stronę należy traktować jako źródło, do którego warto zajrzeć, gdy pojawi się taka potrzeba, a nie łudzić się, że przerobimy wszystko od deski do deski.

11.6. Najlepsze źródło informacji o nowinkach w Androidzie – Android Weekly

Android Weekly, http://androidweekly.net/, czytałem swego czasu regularnie. To świetny newsletter z linkami do najlepszych artykułów na temat Androida.

To wszystko?

Istnieje wiele innych źródeł, ale postanowiłem wymienić tylko te, do których sięgałem i które mogę z czystym sumieniem polecić. Mógłbym rozszerzyć listę przez proste googlanie najbardziej polecanych książek, ale po co? **Lepiej wymienić mniej źródeł, które faktycznie są wartościowe**, niż chwalić się długą listą, którą większość i tak by zignorowała.

Prawie wszystko jest po angielsku…

Wspominałem, że język angielski to najważniejsza umiejętność programisty. Bez niego uczysz się wolniej, mniej efektywnie i sporo **tracisz na każdym kroku**. Im wcześniej sięgniesz po książki po angielsku, tym lepiej dla Ciebie.

12.

Jak wygląda typowy tydzień pracy programisty?

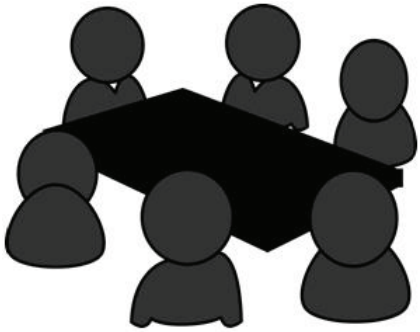
Początkowo chciałem zatytułować ten rozdział: „Jak wygląda typowy dzień pracy programisty", ale stwierdziłem, że opisanie typowego tygodnia będzie lepszym pomysłem. Poznasz realia takiej pracy, dowiesz się, jakie problemy się pojawiają i jak się je rozwiązuje oraz dlaczego nie na wszystko ma się wpływ.

12.1. Planowanie, estymacja, meetingi

Planowanie i estymacja (szacowanie, jaki nakład pracy jest potrzebny na wykonanie danego zadania) to stałe elementy tej pracy. Zależnie od tego, gdzie pracujesz i z jakim projektem masz do czynienia, mogą wyglądać trochę inaczej. Przykładowo w scrumie planowanie następnego sprintu (jeden, dwa lub trzy kolejne tygodnie pracy) odbywa się pierwszego dnia sprintu, więc bardzo prawdopodobne, że co drugi poniedziałek trzeba będzie

poświęcić pół dnia na planowanie pracy. Pół dnia to może za dużo powiedziane: czasami planowanie trwa godzinę, dwie, wszystko zależy od zaawansowania projektu. Jako programista będziesz rozbijał zadania na mniejsze części, które można dzień po dniu wykonywać.

W tygodniu odbywają się również meetingi, podczas których omawia się techniczną stronę różnych zadań. Przyjmuje się, że jeśli programista sześć z ośmiu godzin może pracować nad kodem, to jest dobrze… bardzo dobrze!



Praktycznie nigdy nie spędza się na programowaniu 100% tygodnia roboczego.

W innych systemach organizacji pracy również jest planowanie. Nie da się go ominąć. Śmiało można założyć, że planowanie/meetingi i inne sprawy organizacyjne zajmują jeden dzień z tygodnia pracy programisty. Dobrze, jeśli jest to tylko jeden dzień… Codzienność bywa frustrująca.

12.2. Coś nagle nie działa i nie masz na to wpływu

Gdy pracujesz w zespole osób, który współpracuje z innymi zespołami, to zaczyna się budować wiele zależności. Wówczas nietrudno, by coś przestało działać prawidłowo – i nie możesz nic zrobić. Zaczyna się wtedy chodzenie po zespołach i pytanie o sytuację. Zazwyczaj u wszystkich „wszystko działa" i sprawa nie jest prosta.

Za przykład niech posłuży **mock** (atrapa danych/funkcjonalności, z której się korzysta, dopóki coś nie jest do końca zaimplementowane), którego dostarczył nam kiedyś zespół BI. Tym mockiem były symulowane dane, które miał zwrócić webserwis.

Przy okazji: dowiedz się, czym w praktyce jest mock/mockowanie. Pomocne frazy: „mocking a web service", „java/C# mocking" itd.

Pod dostarczoną strukturę danych przygotowaliśmy dynamiczne UI. Po sprincie wszystko było cacy. Po tygodniu okazało się, że jeden z developerów zespołu BI nie dogadał się z drugim i zaczęliśmy dostawać dane o innej strukturze niż ta w mocku. W efekcie musieliśmy ponownie przysiąść do funkcjonalności, która była właściwie gotowa i odebrana przez product ownera (odebrana funkcjonalność to taka, która ma już otwartą drogę

do nowej publicznej wersji produktu). Irytująca sytuacja, ale z czasem da się przywyknąć.

Na pewno zdarzy się tak, że coś przestanie działać nie z Twojej winy – i nie będzie można nic na to poradzić. To tylko kwestia czasu. Zdarzy się też, że komuś coś nie będzie działało przez Twój błąd. To również tylko kwestia czasu. Każdy człowiek popełnia błędy. Pamiętaj, by zawsze szukać rozwiązania, a nie winnego – **pewnego dnia Ty będziesz tym, który popsuł**.

12.3. Zależności ciąg dalszy

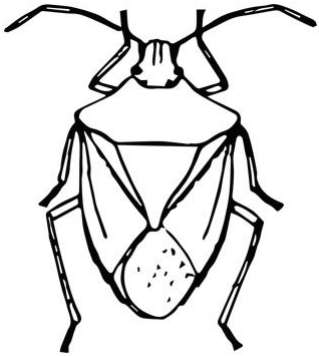
Zależności występują nie tylko wewnątrz zespołu/firmy. Czasami (właściwie to aż za często…) korzysta się w projekcie z różnych zewnętrznych bibliotek. Wszystko jest super: mamy bibliotekę, która rozwiązuje jakiś problem. Oszczędzamy dzięki niej wiele czasu, bo nie musimy czegoś sami implementować… Problemy zaczynają się wraz ze zmianami technologii (np. Android bardzo szybko się rozwija); w nowych wersjach systemu pojawiają się błędy w działaniu oprogramowania. Miałem taki przypadek, gdy biblioteka do automatyzacji testów – Robolectric – zaczęła sprawiać problemy na serwerze testowym z nową wersją Androida. Bibliotekę wymieniliśmy na inną, co pociągnęło za sobą sporo innych zmian. Musieliśmy jednak tak zrobić, ponieważ nie było pewności, jak długo ta biblioteka będzie wspierana przez twórców. Co się stanie, gdy wejdzie kolejna nowa wersja, a nie będzie wsparcia? No właśnie… :)

W każdym typowym tygodniu pracy może się pojawić podobny problem. Czasami rozwiązanie okazuje się proste i traci się tylko parę godzin, ale znam przypadki, w których brak dalszego wsparcia jakiejś zewnętrznej biblioteki kosztował tydzień pracy programisty.

Warto zapamiętać: jeśli korzystasz z zewnętrznych bibliotek, dbaj o to, by były to zależności z solidnego źródła. Na przykład w przypadku Androida porządne i godne zaufania są biblioteki firmy Square.

12.4. Bugfixing/debugging – szukanie i naprawianie błędów

Istnieje takie pojęcie jak **bug sprint**, oznaczające czas spędzony wyłącznie na łataniu błędów. Mam nadzieję, że nie będziesz musiał za często w czymś takim uczestniczyć, bo jest to bardzo męczące.



Naprawieniem błędów będziesz zajmował się **często**. Trudno o tydzień bez takiego zadania. Jak zazwyczaj wygląda praca nad bugami?

- Prawdopodobnie firma będzie miała system do zarządzania ticketami (np. JIRA) i będziesz mógł sam wybrać błąd do naprawy lub ktoś Ci go przydzieli.

- W tickecie znajdziesz informacje na temat zgłoszonego błędu. Czasami zdarza się zgłoszenie, które nie jest bugiem, lecz błędem związanym z obsługą (ktoś nie został właściwie przeszkolony do korzystania z czegoś, jest leniwy lub zwyczajnie głupi).

- Jeśli dochodzisz do wniosku, że jest to faktyczny bug, przechodzisz do szukania przyczyny. **Dobra diagnoza to już połowa sukcesu.**

- Szukasz rozwiązania, korzystając ze swojej wiedzy, doświadczenia i Google.

- Implementujesz poprawkę i oddajesz ticket do przetestowania przez testera.

- Czasami problem jest marginalny i dosyć skomplikowany: wymaga dużego nakładu pracy, a występuje na tyle rzadko, że zespół świadomie rezygnuje z jego naprawy. Takie zadanie zamyka się wówczas jako „Won't fix".

Ogólnie powinno się pracować tak, by bugi naprawiać na bieżąco i nie zaczynać nowej funkcjonalności, aż zdecydowana większość błędów nie zostanie naprawiona. Jednak w praktyce różnie z tym bywa… :)

12.5. Czasami wszystko idzie gładko!

Dość już tego straszenia. Jeśli wszystko w firmie jest dobrze zorganizowane, to zdarzają się tygodnie, w których nic złego się nie dzieje (nawet udaje się skończyć pracę nad zadaniem przed czasem!).

Gdy masz już pewne doświadczenie, to tygodnie zaczynają zlatywać na zleceniach, które już wiele razy robiłeś. Ewentualnie trafia się coś nowego, co jednak dzięki doświadczeniu sprawnie rozwiązujesz. Z jednej strony to komfortowa sytuacja, bo jest mniej stresu (i to w całkiem fajnej i nieźle płatnej pracy), z drugiej – cierpi na tym Twój rozwój i zaczynasz się nudzić. Tygodnie, w których wszystko idzie jak po maśle, bywają po prostu nudne – jednak wielu osobom to pasuje. Programista to w końcu zawód jak każdy inny, a nie zajęcie dla nadludzi, jak to czasami przedstawia część frustratów.

12.6. Nauka w pracy

Nikt nie jest wszystkowiedzący: w świecie szybkiego rozwoju nowych technologii częścią Twojej pracy będzie nauka – nauka w pracy. Nikt nie będzie od Ciebie wyma-

gał, żebyś uczył się po godzinach, a w pracy tylko implementował. Jeśli tak jest, to pomyśl nad zmianą firmy.

W typowym tygodniu pojawiają się reserach taski. Wówczas trzeba sporo poczytać i nauczyć się różnych nowych rzeczy. Gdy zdarzy się luźniejszy czas w projekcie/firmie (kiedy np. coś nie jest na tyle opracowane przez ludzi od produktu, że programista ma trochę luzu), będziesz mógł poduczyć się w X czy Y. Taki czas raportuje się wtedy jako „samodoskonalenie/nauka technologii", o ile jest konieczność raportowania. Wiele firm od tego odchodzi, bo to marnotrawienie czasu raportującego i osoby, która taki raport kontroluje (o ile ktokolwiek to robi…). Albo się ufa pracownikowi, albo w ogóle z nim nie pracuje.

W moim przypadku często bywało tak, że parę dni mogłem poświęcić na naukę i pracę nad prototypem jakiegoś rozwiązania. To zdecydowanie najlepsza część tej pracy, jednak na takie zaufanie trzeba sobie zapracować. Jeśli dasz się poznać jako solidna i kompetentna osoba, to zdziwisz się, ile wolności można dostać wewnątrz firmy, by testować różne rozwiązania i poświęcać czas na reserach. Długofalowo takie podejście się po prostu firmie opłaca.

12.7. Krótkie podsumowanie typowego tygodnia

Podsumowanie: z czego składa się tydzień programisty:

- Pojawią się problemy z zewnątrz: ktoś coś popsuje albo Ty coś popsujesz. Pamiętaj: szukaj rozwiązania, a nie winnego.

- Możesz ułatwić sobie pracę, korzystając z gotowych bibliotek. Używaj ich jednak rozważnie, bo z czasem mogą zacząć sprawiać problemy.

- Naprawianie błędów (bugfixing) to chleb powszedni: czasem zgłoszony błąd nie jest błędem albo występuje na tyle marginalnie, że nie jest wart potencjalnego nakładu pracy.

- Bywa i tak, że idzie gładko! Wkrada się wtedy nuda, ale zatęsknisz za tą nudą, gdy coś się poważnie popsuje.

- Część czasu w pracy to nauka. Nauka, za którą Ci płacą.

Każdy tydzień to mieszanka wymienionych zdarzeń: implementacja funkcjonalności, naprawianie tego, co się popsuło, ciągła nauka i nieskończona seria meetingów. Koniec końców jest podobnie jak w wielu innych zawodach – nie oczekuj cudów i implementowania zawsze interesujących rzeczy.

13.

Jesteś gotowy do pierwszej pracy? Pytania

Przygotowałem listę pytań, na które warto sobie odpowiedzieć, zamiast wysyłać do mnie maila z pytaniem: „Czy jestem już gotowy, by aplikować do pierwszej pracy jako programista?". Do każdego pytania dodałem też przykładową odpowiedź.

Czy jesteś w stanie zrozumieć **angielskojęzyczną** dokumentację dowolnej biblioteki?

Odpowiedź: Tak, do wielu małych projektów podpinałem różne biblioteki, które wykorzystałem po przeczytaniu dokumentacji.

Czy intuicyjnie wiesz, czego szukać, jeśli musisz zaimplementować coś, z czym nie miałeś wcześniej styczności?

Odpowiedź: Tak. Zazwyczaj szukam prostych, praktycznych przykładów, czytam kod, a później w razie potrzeby poznaję teorię, którą lepiej rozumiem po styczności z przykładem.

Czy potrafisz samodzielnie organizować swoją pracę, by była ona efektywna?

Odpowiedź: Tak. Zanim zacznę pisać kod źródłowy, rozpisuję całość z grubsza na papierze i rozbijam na mniejsze zadania. Gdy utknę, staram się szukać rozwiązania w sieci. Dopiero gdy nie znajdę nic sensownego, pytam kolegów.

Czy napisałeś jakiś projekt do końca, tak by można było krótko omówić, przedstawić problemy, które się pojawiły itd.?

Odpowiedź: Tak, napisałem aplikację mobilną, w której użyłem technologii pozwalających ponownie wykorzystać kod źródłowy pod różne systemy operacyjne. Największym problemem była konfiguracja xCode pod iOS i połączenie tego z pluginem w Eclipse (plugin w Eclipse korzystał z lokalnych plików, które wygenerowało xCode). Podczas projektu stosowałem narzędzia do kon-

troli wersji (Git), dzięki czemu zdobyłem duże doświadczenie w pracy z tym systemem.

Wyjaśnienie: plugin/wtyczka to dodatek do oprogramowania. Możesz np. zainstalować wtyczkę do swojego IDE, by korzystać w nim z Gita.

Czy zapisywałeś/czytałeś dane z/do jakiegoś typu plików?

Odpowiedź: Tak, pisałem aplikację, która pobiera dane z plików CSV i generuje raporty w formacie PDF.

Czy masz doświadczenie w pracy z bazą danych?

Odpowiedź: Tak, napisałem już parę aplikacji webowych, które korzystały z bazy MySQL. Potrafię poruszać się w bazie danych, ale nie jest to dziedzina, w której się specjalizuję.

Czy masz jakieś doświadczenie związane z technologiami webowymi?

Odpowiedź: Tak, stworzyłem parę prostych stron WWW, wykorzystywałem też jQuery do pobierania danych z webserwisów.

Czy znasz najważniejsze protokoły sieciowe i czy stosowałeś któryś z nich?

Odpowiedź: Tak, pisałem aplikacje na Androida, które korzystały z webserwisów za pomocą protokołu HTTP.

Jaka jest Twoja wiedza o programowaniu obiektowym? Czy wiesz, czym różni się np. (w przypadku Javy) interfejs od klasy abstrakcyjnej?

Odpowiedź: Stosowałem już programowanie obiektowe w praktyce. Odnośnie do pytania o interfejs i klasę abstrakcyjną: w interfejsach nie ma ciał metod (które mogą, ale nie muszą być obecne w klasie abstrakcyjnej), klasa może implementować wiele interfejsów, ale dziedziczyć tylko jedną klasę abstrakcyjną (w Javie, w C++ możliwe jest dziedziczenie po wielu klasach).

[Zamiast pytania o interfejs i klasę abstrakcyjną możesz podstawić dowolne znalezione w sieci pytanie o obiektowość i programowanie obiektowe w języku, który wybrałeś].

Pytanie: Czy pisałeś własny webserwis lub korzystałeś z webserwisu?

Odpowiedź: Jedno i drugie: napisałem prosty webserwis, który pobiera dane z bazy danych, oraz aplikację mobilną korzystającą z tego webserwisu.

Czy wiesz, czym są testy jednostkowe?

Odpowiedź: Tak, długofalowo podnoszą one jakość kodu. Pozwalają wykryć, czy kod jest kiepskiej jakości – jeśli trudno napisać testy pod dany kod, to znaczy, że mógł on zostać lepiej przemyślany.

Czy potrafisz korzystać z danych dostarczonych w formacie JSON lub XML?

Odpowiedź: Tak, dane dostarczane przez webserwisy były zapisane w formacie XML i musiałem je samodzielnie parsować.

Czy korzystałeś kiedyś z narzędzi do kontroli wersji?

Odpowiedź: Tak, we wszystkich projektach (również w tych pisanych w ramach nauki) używałem Gita. Oto lista do projektów na GitHubie.

W jakim IDE sprawnie się poruszasz?

Odpowiedź: Eclipse oraz IntelliJ. Spędziłem w nich sporo godzin i stosuję skróty usprawniające pracę.

Jeśli potrafisz sobie szczerze odpowiedzieć na powyższe pytania, to jesteś gotowy, by zacząć poszukiwania pierwszej pracy jako programista. Podchodź do rekrutacji jak

do nauki: z każdą rozmową będziesz wiedział więcej (osoby z jakimi umiejętnościami szukają na stanowisko, które Cię interesuje itp.), dzięki czemu w przypadku niepowodzenia lepiej ukierunkujesz swoją naukę. **Lepiej jest zacząć chodzić na rozmowy za wcześnie i dowiedzieć się, czego należy się douczyć, niż zacząć chodzić za późno i zmarnować parę miesięcy na opanowanie zbędnej wiedzy**. Wiem, że to brzmi jak banał typu cytat z Paulo Coelho, ale ja właśnie tak zmarnowałem sporo czasu i żałuję, że nie zacząłem chodzić wcześniej na rozmowy kwalifikacyjne.

14.

Pierwsze zadania programisty. Faktyczne zadania od czytelników

Plus dwa zadania ode mnie na koniec. :)

Kiedyś poprosiłem czytelników mojego bloga o podesłanie pierwszych zadań, które dostali w pracy do wykonania. Te przykłady dają pogląd na to, czego można się spodziewać w pierwszych tygodniach i miesiącach pracy.

Michał

Pierwsze zadania to na razie front-end, i to dość specyficzny – pluginy robione w Eclipse przy użyciu narzędzi SWT i JFace. Takie narzędzia wynikają z architektury u klientów (duże korporacje i ich poddostawcy). Mam to ogromne szczęście, że firma inwestuje w ludzi i ich szkolenie. Dlatego moje pierwsze zadania to:

- przeniesienie etykiety wyświetlającej numer wersji w odpowiednie miejsce w oknie;

- budowa okienka z tabelą, w której będzie działało filtrowanie wyświetlonych wyników i ich edycja;

- przeniesienie poleceń z paska do menu kontekstowego;

- z dostępnością pewnych opcji zależnie od rodzaju wybranego obiektu;

- stopniowe porządkowanie niezbyt elegancko napisanego programu;

- przenoszenie pewnych funkcjonalności do oddzielnych klas w celu ich późniejszego wykorzystania.

Na razie sporo jest pracy odtwórczej i zakładającej wykorzystanie gotowych rozwiązań – ale to też ponoć ważna umiejętność (wg naszego team leadera), a przy okazji poznajemy, jak zbudowane jest oprogramowanie, z którym pracujemy. Jest ponoć mocno nietypowe. :) Podpisana umowa o pracę zakłada spory poziom poufności, więc nie mogę pisać za bardzo o szczegółach pracy.

Następnym etapem będzie (zaczynamy jutro) stworzenie prostej przeglądarki, która ma wyświetlać dane zależnie od informacji pochodzących z softu tworzonego przez drugiego, nieco tylko bardziej doświadczonego ode mnie kolegi.

Co mnie zaskoczyło? Sam nie wiem, nie miałem niemal żadnych konkretnych oczekiwań. Na duży plus – ale tego się spodziewałem, bo część osób tu pracujących znam – świetna atmosfera. Żaden Janusz-Soft, jesteśmy wydziałem programowania zachodniej firmy obsługującej potentatów z różnych branż. Kilka liczb: niemal 10 lat temu firmę stanowiły 4 osoby. Dziś to 20 osób, a w tym czasie odeszły z firmy zaledwie 2 osoby. Podczas rozmowy przed podjęciem pracy pytałem szefa, pod jakim kątem mam się szkolić. Usłyszałem: „JAVA ogólnie, kolekcje i XML". Tymczasem pierwsze zadania są wybitnie front-endowe, ale co tam. Na kolekcjach też sporo pracuję, a i programowanie obiektowe poznaję.

To moje doświadczenia z pierwszych dni pracy jako programista. Może w jakiś sposób się przydadzą. Dodam tylko, że na razie jestem bardzo zadowolony z podjętej decyzji o przebranżowieniu, choć wiem, że spora w tym zasługa faktu, że wiedziałem, z kim będę pracować. Pewnie mało kto ma taki komfort.

Marcin

Moim pierwszym zadaniem było przepisanie dwóch projektów opartych na CMS Liferay (portlety + themes) na framework Bootstrap (wtedy w wersji 2). Niby nic dziwnego, ale – jak się okazało – Liferay ma pod maską swojego przerobionego Bootstrapa. Moim zadaniem było więc nadpisanie Bootstrapa Liferaya (AUI) na czystego Bootstrapa. Wtedy też pierwszy raz miałem do czynie-

nia z freemarkerami .ftl. Od klikania i pisania odpadały mi ręce… Najgorszy był jednak zawód, gdy AUI nadpisywało moje style, JavaScript nie działał bądź ID szalały po stronie.

PS. Jak przyszedłem do pracy pierwszego dnia, to musiałem formatować komputer. :) I tak miałem szczęście, ponieważ mój kumpel nie miał komputera przez 2 tygodnie.

Mój blog: www.blog.webbd.pl

Jan Jakub Jurec

W te wakacje miałem przyjemność pierwszy raz pracować. Wskoczyłem na staż wakacyjny w dużej korpo do działu zajmującego się zarządzaniem VCS. Wymagany był Python, Bash i trochę Linuxa. Przez 3 miesiące zrealizowałem (z kolegą stażystą) następujące zadania:

- Odczytywanie struktury repozytorium SVN-owego z XML-a i sprawdzanie poprawności tego repozytorium.

- Sprawdzanie, czy externale SVN-owe są poprawne, a ich dokumentacja – aktualna. Oba zajęły tydzień. Rozgrzewka.

- Przetłumaczenie, otestowanie i odokumentowanie programu (~2000 linii kodu) napisanego w Pythonie 2.7.

Miał działać on pod 2.6, 2.7, 3.3, 3.5. Program pythonowy był wrapperem do SVN-a. To już zajęło 4 tygodnie. Około 200 UT i FT napisanych.

- Zaprojektowanie, stworzenie i opracowanie dokumentacji systemu, który miał samodzielnie realizować taski wystawione na Redmine. Trzeba było ogarnąć Redmine, Jenkins, Celery, Flask, Docker. Niecałe 2 miechy.

Bardzo wiele się nauczyłem. Uczyłem się w pracy. W domu odpoczywałem. Świetnie to wspominam.

Witkowski Marek

Moim pierwszym zadaniem w pracy jako junior developer było wyniesienie do śmieci kartonów, w których przyszedł mój sprzęt do pracy. :-) Następnie musiałem wszystko rozpakować i podłączyć, a były to 2 monitory 24-calowe, laptop, mysz bezprzewodowa i klawiatura. Cały sprzęt nowy, więc – wiadomo – fajne uczucie.

Kolejnym zadaniem było przygotowanie środowiska pracy, czyli instalacja maszyn wirtualnych, IDE i reszty niezbędnego oprogamowania.

Wszystko to w sumie trwało cały dzień, a przez kolejny tydzień do moich obowiązków należało przeczytanie dokumentacji frameworka Symfony.

Nasza strona: ideaspot.pl :-)

Tomasz

Cześć, pracuję jako PHP developer w firmie związanej z branżą e-commerce.

Pierwsze dwa dni pracy poświęciłem na spokojne postawienie środowiska oraz konfigurację Dockera, co przy wielkości platformy nie było proste, lecz pomogli starsi stażem współpracownicy.

Pierwszym zadaniem, o ile można to tak nazwać, było stworzenie testowego modułu promocyjnego w serwisie: moduł administracyjny z dodawaniem/edycją/usuwaniem promocji na produkt/kategorię oraz wpięcie mojego modułu do strony głównej serwisu. Zadanie zajęło mi około dwóch tygodni, w trakcie których poznałem najważniejsze moduły platformy oraz napotkałem masę problemów, które są częste w późniejszej pracy (zagadnienie stref czasowych, struktura bazy danych, migracje, praca ze słabo udokumentowanym kodem, poznanie IDE oraz firmowych zasad używania Gita). W międzyczasie uczestniczyłem w spotkaniach scrumowych, co także pozwoliło mi wdrożyć się w tę metodologię. Tyle i aż tyle. Nie było aż tak strasznie, jedynie na początku przeraził mnie ogrom platformy, który dopiero po dłuższym czasie odkrywa swoje tajemnice.

Kamil

Aplikacja, która umożliwiała importowanie plików XLS i XLSX do bazy danych. Po stronie serwera był Tomcat, Hibernate i Spring MVC. Baza danych PostgreSQL. Do parsowania plików XLS i XLSX wykorzystano bibliotekę Apache POI. Front-end aplikacji: HTML 5, JSP, jQuery i Bootstrap.

Michał Franc

Dawno temu to było… hmm. Pierwsza praca to była taka śmiechowa firma znajomych, których serdecznie pozdrawiam. Było ich trzech i wzięli pod swoje skrzydła darmowych stażystów. Tak, pracowałem za darmo. Robiłem jakiś śmieszny portal społecznościowy, który nie miał szans przetrwać na rynku i ogólnie był „bez sensu". Trochę ASP.NET + SQL. Głównie siedziałem w bazie i pisałem repozytorium.

Pierwsza poważna praca to był projekt w SharePoincie. Aplikowałem na stanowisko programisty webowego, nie wiedząc w ogóle, co to jest HTTP, znając lekko podstawy HTML-a i CSS. Jedyne moje doświadczenie to było wyklikiwanie webformsowych appek w Visual Studio. W tej pracy siedziałem i przerzucałem tysiące XML-i z jednego miejsca na drugie. Uszkodziłem bazę danych SharePointa do tego stopnia, że nie dało się jej już zmodyfikować. Wtedy też pierwszy raz zetknąłem się z JavaScriptem i czymś, co nazywało się jQuery! John Resig – twór-

ca jQuery – wtedy to był guru. Oj, jakie cuda można było z jQuery robić. Pamiętam, jak czytałem, co to Ajax, i nie wiedziałem, o co chodzi i po co to. Dużo czasu spędziłem, mecząc się z SharePointem i narzekając na kod, że jak to taki badziewny kod można pisać… Zapamiętałem słowa senior developera: „Dorośniesz, to zrozumiesz". No i zrozumiałem :) Do dzisiaj nie wiem, czy cały ten projekt sharepointowy został kiedykolwiek użyty. Raczej nie.

Blog: mfranc.com/blog

Rafał

Cześć, w moim wypadku była to po pierwsze instalacja Eclipse, Android Studio, SVN. Na początek jakieś proste zadania w istniejących projektach w branży mobilnego handlu, zadania takie jak poprawa layoutu + jakieś proste zadania programowe, np. wychwycenie listenera + akcja, dodanie filtra, zmiana formatowania wyświetlanej liczby. Przede wszystkim chodziło o to, aby trochę poznać projekt oraz sprawdzić moje umiejętności radzenia sobie z zagadnieniem/problemem.

Następnie – chyba nawet w pierwszym miesiącu – jednoosobowa praca nad nowym projektem Pictor Digital Signage Player.

PS. Ze względu na studia stacjonarne praca 3 dni w tygodniu + jak mam czas, to nadgodziny – ile tylko chcę ;)

gonczor

Jestem backendowcem robiącym w Django, pierwsze zadania w pracy dotyczyły jednak postawienia serwera testowego. Linuks + wbudowany djangowski serwer do testowania aplikacji androidowej projektu. Nawet bez konfigurowania jakoś szczególnie środowiska, dockerów itd. Pierwsze stricte programistyczne zadanie polegało na stworzeniu komend serwerowych (Django ma coś, co się nazywa management commands do automatyzacji części pracy, która musi być powtarzana cyklicznie) do obsługi płatności kartą. Miałem ściągać maile ze skrzynki, parsować raporty CSV i zapisywać w bazie danych. Zacząłem z dość grubej rury, do tego stopnia, że część kodu była copypastą, której za bardzo nie zrozumiałem i nieco się stresowałem tym, że nie sprostam dużemu zadaniu, ale po miesiącu siedzenia nad tym jakoś się udało. Szefo okazał się pomocny i wyrozumiały dla moich braków. Raporty schodziły, nauczyłem się, co to jest CSV, jak działają modele w Django, jak działa cała masa niuansów pythonowych (takich jak przechodzenie między strefami czasowymi, co przyprawiło mnie o sporo bólu głowy na początku), jak dodawać to do panelu administracyjnego, tworzenie relacji, obsługa bazy danych i masa innych. Trochę mi to też uświadomiło, że wszystkiego idzie się nauczyć, wolniej lub szybciej, ale ciężką pracą można wiele zwojować.

Ciekawa jest też historia otrzymania pracy – otóż zniechęcony niepowodzeniami szukania pierwszej pracy jako programista (szukałem pracy jako javowiec) zacząłem

rozglądać się za czymś innym i trafiłem na ofertę kuriera rowerowego. Pierwszym pytaniem wysyłającego ofertę, kiedy się do niego zgłosiłem, było: „A nie robi pan przypadkiem w Pythonie, bo mamy nasze systemy w Django". Cóż, skojarzył mnie chyba z jakiejś grupy dla programistów. :D

Jeśli mogę zareklamować projekt, to chciałbym, żeby to było to: github.com/gonczoor/ServerPy – opensource'owy projekt stworzenia inteligentnego domu oparty na Raspberry, Arduino i Androidzie, do którego szukam pomocników.

Michał Gellert

Mam na imię Michał, półtora roku doświadczenia jako programista, cały czas w tej samej firmie. Wspomnienie mojego pierwszego zadania jest zabawne, ponieważ nie poradziłem sobie z nim. :) To znaczy na początku oczywiście była to trauma, ale po kolei.

Dla mnie była to w ogóle zmiana branży. Pracowałem wcześniej na magazynie, z dala od cywilizacji, w malutkiej wiosce, ale dojeżdżałem na studia i po ich skończeniu udało mi się złapać pracę (w ogóle przez Facebooka, napisałem, czy ktoś mnie chce, dostałem kilka telefonów i zatrudnienie). Pierwszym zadaniem było w module w aplikacji pamiętającej Javę 6, w technologii GWT sprawić, żeby spinnery (pola rozwijane) w zależności od konfiguracji w innym miejscu dodawały się po kliknięciu

z coraz mniejszą liczbą wartości. Tzn. jeśli było maks. 6 wartości, to można było dodać maks. 6 spinnerów. Jeśli któraś wartość została ustawiona, to liczba możliwości na pozostałych spinnerach była zmniejszana właśnie o tę wartość. Na bieżąco konsultowałem swoje pomysły na rozwiązania, ale żadne z nich nie spodobało się szefowi. Teraz nie wyobrażam sobie, żebym nie mógł sobie poradzić z tym zadaniem, ale wtedy to była masakra. Po tygodniu wreszcie szef usiadł ze mną i je rozwiązał. Także początek dla mnie był bardzo ciężkim okresem, z następnymi zadaniami już sobie radziłem, ale nauczyłem się też najważniejszego – rozmawiać, pytać, jeśli czegoś nie potrafisz. Szkoda tracić czas i zmóżdżać się nad czymś przez 6 godzin (co i tak nie doprowadzi do rozwiązania), jak zajęcie 15 minut koledze sprawi, że rozwiążesz w 3 godzinki. Aktualnie, jeśli zdarzy mi się zadanie, z którym sobie nie radzę, to po prostu pytam. Ustawiam sobie okno czasowe, powiedzmy 30 minut, jeśli w tym czasie nie poczyniłem kroków do przodu, to się do kogoś zgłaszam i proszę o pomoc. Chociaż zdarza się to coraz rzadziej. Aktualnie jestem w projekcie, który tworzyłem praktycznie od połowy, czyli sporo rzeczy po prostu wiem, jak działa.

Blog: michalgellert.pl/blog

William

Może nie jest to Java, tylko webdev, ale pierwsze zadanie dla gościa jeszcze w czasie nauki w technikum: samodzielnie wykonaj system CRM w wersji wielojęzykowej wraz z API dla aplikacji mobilnej spełniający wymagania podane w specyfikacji. To było trochę dużo jak na człowieka bez jakiegokolwiek doświadczenia, tym bardziej że do tamtego momentu specjalizowałem się raczej w C++, a w technologiach webowych stawiałem dopiero pierwsze kroki.

thedilly

Moje pierwsze zadanie w pracy jako programista dotyczyło oprogramowania SolidWorks. Moim zadaniem było stworzenie programu, który „konwertowałby" pliki SolidWorks z rozszerzeniem .sldprt i .sldasm do rozszerzenia .smf, które było obsługiwane przez program SolidWorks Composer. Cały program był wykonany w Visual Studio w WPF (język C#) z użyciem bibliotek SolidWorks. Program był prosty i głównie polegał na wyborze plików do „konwersji" (poprzez dodawanie do listy wybranych plików z okna dialogowego) i kliknięciu przycisku, który robił całą robotę – otwierał wybrane pliki jeden po drugim w programie SolidWorks i zapisywał je z rozszerzeniem .smg w wybranej przez użytkownika ścieżce.

Program był potrzebny, gdyż większe pliki potrafiły się otwierać i zapisywać kilka dobrych godzin, więc o wiele łatwiej i wygodniej było zautomatyzować ten proces. :-)

Tomasz

Takie pierwsze zadania, które chyba większość dostaje:

- Uruchomić sobie środowisko. ;)
- Zamienić wywołanie jakiejś metody na inną lub jakieś inne proste zadanie, po to by zapoznać się z workflow (w moim przypadku z JIRA, na początku SVN, później Git), tzn. gdzie commitujemy, jak wygląda review, poprawki, merge.
- Otestować jakąś klasę/serwis.
- Drobne refactoringi, np. zamienić jakieś operacje na listach na Streamy i Lambdy z Javy 8.

Bartosz

Jako Android developer na praktykach miałem za zadanie czysto programistyczne dodać do istniejącej aplikacji śledzenie GPS, zapisywanie je do lokalnej bazy danych + notyfikacje wysyłane z GCM oraz zaproponować, jaka zwrotka ma być zwracana z serwera. Dodatkowo projekt był w Eclipse i miałem go przenieść do Android Studio.

Marcin

W pierwszej pracy, a właściwie podczas dwumiesięcznych praktyk, m.in.

- najpierw miałem zapoznać się z dokumentacją Springa;
- skonfigurować środowisko, Git…;
- zrobić code review jednej klasy;
- skomunikować za pomocą REST-a już napisane microservice'y;
- napisać klasę przechwytującą rzucane exception i zapewnić odpowiedni komunikat;
- napisać testy jednostkowe i funkcjonalne (Junit);
- napisać microservice odpowiedzialny za logowanie błędów.

Aktualnie pracuję nad aplikacją dla youtuberów o nazwie Comments Cutter: commentscutter.tk

Adam

W pierwszej pracy jako pierwsze zadania dostawałem poprawę i pisanie interpreterów, które przez kopiowanie faktur VAT z PDF za pomocą regexów wyciągały interesujące firmę dane. :) Aplikacja typowo ERP.

Bartłomiej Kiełbasa

W mojej pierwszej pracy od razy rzucili mnie na głęboką wodę. Mój szef posiadał swojego autorskiego CMS-a. Podpięte pod niego było forum PhpBB by Przemo (chyba) w taki sposób, że użytkownicy się dublowali. Jak ktoś zakładał konto w CMS-ie, to równocześnie było zakładane na forum, czyli miały 2 niezależne bazy danych z synchronizowaną tabelą użytkowników.

Problem polegał na tym, że gdy zalogujesz się w CMS-ie i przejdziesz na forum, to musiałeś się zalogować jeszcze raz. Działało to też tak w drugą stronę. Moim zadaniem było zrobienie pojedynczego logowania. Jak logujesz się na forum, to automatycznie ma być tworzona sesja po stronie CMS-a i vice versa.

Moim kolejnym pierwszym zadaniem było naprawienie buga w rozliczeniach. Przy przeliczaniu kwot prowizji na koniec miesiąca tkwił jakiś błąd, który u niektórych osób powodował za nisko wystawiające się faktury. Zmieniłem wtedy pracę i jako jedyny developer rozwijałem CRM-a dla małej firmy parafinansowej.

Innym moim pierwszym zadaniem była zmiana tekstu w stopce w sklepie opartym na Magento. To było moje pierwsze spotkanie z tym tworem, więc zajęło mi to „tylko" 4 godziny.

Blog: bkielbasa.pl

Maciej

Może miałem trochę łatwiej niż inni. Nie przechodziłem procesu rekrutacji. U mnie w firmie szukali kogoś do wsparcia przy systemie (w jednej chwili odeszło 3 programistów). Podjąłem wyzwanie. Głównie dlatego, że wcześniej przez 6 lat pracowałem na bazach danych i… miałem tego dość.

Oczywiście mam podstawy Javy i ostrzegałem szefów, że nie od razu zastąpię kolegów „seniorów" – potrzebuję czasu na wdrożenie, poznanie technologii (Maven, Spring, Hibernate, GWT itd.). Powoli się wkręcam.

Do tej pory miałem zadania typowo na poznanie systemu, na którym pracuję. W skrócie – usuń jakąś funkcjonalność, dodaj kolumnę do raportu, zmodyfikuj nazwy w słownikach, dodaj nowe zasady autoryzacji, a także (tego chyba nikt nie lubi) – testowanie. Dużo mi to wszystko dało. Może pełnej swobody w poruszaniu się po paczkach jeszcze nie mam, ale na pewno jest dużo lepiej niż na samym początku.

Teraz dostałem następne zadanie – wdrożenie panelu do zgłaszania błędów – z założenia jest to prosta funkcjonalność, ale dzięki temu przejdę całą ścieżkę wdrażania nowych funkcjonalności.

Ku mojemu zdziwieniu – mam bardzo duże wsparcie ze strony zespołu. Nie spodziewałem się tego. Na każde, nawet najbardziej noobskie pytanie chętnie odpowiadają

i w każdej chwili mogę liczyć na pomoc. Oczywiście przyjąłem, że zapytanie kolegów to jest ostateczna instancja – próbuję sam doszukać się rozwiązania problemu.

Jestem zadowolony, chociaż mam świadomość, jak długa droga mnie jeszcze czeka. Mam nadzieję, że to pomoże.

PS. Tak naprawdę dzięki Twojemu e-bookowi zrozumiałem, że nie ma się czego bać i dlatego postanowiłem spróbować. Nie żałuję. Dziękuję.

Maciej Aniserowicz

Swoje pierwsze prawdziwe zadanie jako programista otrzymałem na praktykach po IV roku studiów, ponad 10 lat temu. Musiałem napisać program, który zintegruje się z bramką SMS-ową Ery i pozwoli na wysyłanie wiadomości, biorąc pod uwagę aktualny stan konta użytkownika, realizowany za pomocą „tokenów". Prawdziwym wyzwaniem było jednak osadzenie tego programu na toolbarze w przeglądarce… Internet Explorer 6! Do dziś pamiętam satysfakcję, gdy po dwóch tygodniach walki z obiektami COM i wpisami w rejestrach dostarczyłem działające rozwiązanie.

Blog: devstyle.pl

Jacek Poczwarka

Jakie były moje pierwsze zadania z pracy? Pierwszego dnia jedyne, co dostałem, to laptopa z Windowsem 10, a resztę miałem sobie sam skonfigurować według własnych potrzeb. Miałem wolną rękę, jeżeli chodzi o wybór programów, których będę używać. Była to mała firma, więc nie było z tym problemu.

Jako że była to praca web developera, to użyłem wszystkiego, czego używam do celów prywatnych. Atom, File Zilla, Git i Chrome Canary z podstawowych narzędzi. I do tego kilka mniejszych narzędzi. Gdy do południa skonfigurowałem już wszystko jak należy, to zabrałem się za przerabianie projektu. Musiałem się nauczyć CMS-a Kirby, jak działają w nim skrypty, jak działają w nim pluginy.

Do końca dnia udało mi się napisać jeden z kalkulatorów używany na jednej ze stron firmy do przeliczania wartości liczbowych.

Kolejne zadania to głównie pisanie wtyczek i pluginów w PHP i JS-ie. Czasem używałem Ajaxa i jQuery. Jak trzeba było, to rozbudowywało się głównego CMS-a według własnych potrzeb.

Kirby jest jednym z „lekkich" CMS-ów, więc praktycznie w sobie nic nie ma. Dzięki temu strona się szybciej ładuje i jest lepiej indeksowana w Google. Ale jednak trzeba poświęcić czas, aby wszystko działało po naszej myśli. :)

Blog: igitara.pl

Arkadiusz Benedykt

Akcja taka: zostałem zatrudniony, mimo że nie znałem C#. To nie były czasy, gdzie ofert dla devów było na pęczki, a C# był językiem tak niszowym, że nawet nie wiedziałem, w co się pcham. Rozmowa poszła spoko i developer, który mnie rekrutował, powiedział „dyrektorom", że chce mnie, a jeśli nie, to on ma to w dupie. „Dyrektory" chciały kogoś innego, kto na rozmowę przyszedł w garniturze. Koniec końców jestem. Oglądam system, który mamy rozwijać. Kupa długu, kupa dziwnego kodu i ja bez wiedzy o C#. Za co by się zabrać? Najlepiej za coś, co można zrobić na boku, co da wartość, ale nie zepsuje istniejącego kodu, coś, co jak się nie zrobi, to nie będzie tragedii. Padło na konwersję kwoty na fakturze do zapisu słownie. Obecny był mniej więcej „Kali chcieć tyle". Więc postanowiłem to ucywilizować, a przy okazji nauczyć się C# na tym. Zatem pierwsze zadanie to konwersja kwoty do wersji słownej.

Na koniec obiecane moje pierwsze dwa zadania

W ostatniej firmie, w której pracowałem, moje pierwsze zadanie polegało na zmianie koloru czcionki w przypadku określonego typu zamówienia. Proste zadanie, pozwalające zrobić pierwsze kroki w projekcie. Pomogło mi zapoznać się z logiką odpowiedzialną za typy zamówień. Była to mobilna aplikacja na Androida, za pomocą której kie-

rowca odbierał zamówienia w taksówce. Niby banał, ale czułem satysfakcję, wiedząc, że miałem swój wkład w aplikację, z której będzie korzystało parędziesiąt tysięcy taksówkarzy.

Drugim zadaniem w tej samej firmie było dodanie „debug dialog" do ekranu logowania aplikacji. O co chodziło? Gdy budowaliśmy aplikację, wybieraliśmy, na jaki serwer mają być wysyłane zapytania (był serwer główny i parę testowych). Gdy tester chciał testować na innym serwerze, musiał prosić nas o zbudowanie innej wersji albo szukać, skąd pobrać wersję na dany serwer. Moim zadaniem było sprawdzenie, czy aplikacja jest budowana tak, by zostać udostępniona na zewnątrz (np. by wrzucić ją do Google Play), czy też jest budowana w trybie do wewnętrznego testowania – w przypadku użytku wewnętrznego na ekranie logowania miała pojawić się dodatkowa funkcjonalność odpalenia dialogu, w którym można zmienić serwer – było parę predefiniowanych, albo można było jego dane wpisać ręcznie.

15.

Co dalej? Parę słów na koniec

Mam nadzieję, że nie zanudziłem Cię na śmierć i wyniosłeś coś z tej książki. Pamiętam, że zawsze gdy kupowałem książkę, to zaczynałem od lektury zakończenia. Ciekawe, czy oprócz mnie ktoś tak robi.

Jeśli wszystko, o czym napisałem, powoli staje się dla Ciebie zrozumiałe i zrobiłeś parę projektów w ramach nauki, to możesz zacząć rekrutować do firm. Tylko w ten sposób skonfrontujesz się z rzeczywistością i dowiesz, nad czym jeszcze musisz popracować, czego się poduczyć. Zawsze możesz spróbować ponownie za parę miesięcy czy pół roku. Bez paniki.

Możesz też dojść do wniosku, że nie jest to praca dla Ciebie. Wielu programistów programuje tylko w ramach hobby, a ich **umiejętności procentują w innych zawodach**. Czas poświęcony na sensowne poszerzanie swoich horyzontów nigdy nie jest czasem zmarnowanym.

Jeśli przeczytałeś książkę od deski do deski, to prześlij mi krótką opinię na Facebooku, Twitterze lub na maila: matt@javadevmatt.pl. Albo za pośrednictwem jakiejkolwiek innej platformy, która jest w tej chwili popularna – 10 lat temu (w 2006 r., w momencie powstawania tej książki jest 2016) popularny był Myspace, a wiemy, jak skończył. Ewentualnie podeślij jakiegoś zmutowanego gołębia pocztowego, jeśli nadszedł czas apokalipsy, a Ty jakimś cudem czytasz tę treść na swoim pip-boyu. Jeśli gołębia nie zjedzą mutanty, to postaram się odpisać.