Java. Tworzenie aplikacji sieciowych za pomocą Springa, Hibernate i Eclipse

Autor: Anil Hemrajani
Tłumaczenie: Konrad Rymczak, Grzegorz Skorczyński, Ewa Bucka
ISBN: 978-83-246-0682-5
Tytuł oryginału: Agile Java Development with Spring, Hibernate and Eclipse
Format: B5, stron: 352

Tworzenie zaawansowanych projektów korporacyjnych w Javie

- Poznaj sprawdzone metodologie i procesy
- Naucz się stosować praktyczne technologie i narzędzia
- Twórz w Javie kompletne rozwiązania w optymalny sposób

Chciałbyś tworzyć rozbudowane aplikacje w języku Java szybciej i w prostszy sposób? Liczne narzędzia i technologie budowania programów w Javie często ułatwiają wykonywanie niektórych zadań, ale jednocześnie niepotrzebnie komplikują i wydłużają proces powstawania gotowych produktów. Jak dobrać odpowiednie techniki i zastosować metodologię, która usprawni i przyspieszy pracę? Niektórzy już to wiedzą – Ty też możesz skorzystać z ich doświadczeń!

„Java. Tworzenie aplikacji sieciowych za pomocą Spring, Hibernate i Eclipse" to praktyczny poradnik opisujący wydajną i sprawdzoną metodologię szybkiego pisania oprogramowania w języku Java. Dzięki tej książce poznasz techniki programowania ekstremalnego oraz metodologii Agile i nauczysz się stosować je podczas pracy nad programami. Dowiesz się, jak zaprojektować aplikację, przygotować środowisko i korzystać z wiersza poleceń oraz proponowanego zestawu narzędzi – platformy Spring, mechanizmu odwzorowań Hibernate i IDE Eclipse – co pozwoli Ci w prosty sposób użyć zaawansowanych rozwiązań.

- Techniki programowania ekstremalnego (XP)
- Wprowadzenie do metodologii Agile
- Korzystanie z platformy Spring
- Utrwalanie obiektów za pomocą Hibernate
- Praca w IDE Eclipse
- Debugowanie oprogramowania
- Monitorowanie i profilowanie aplikacji
- Refaktoryzacja kodu

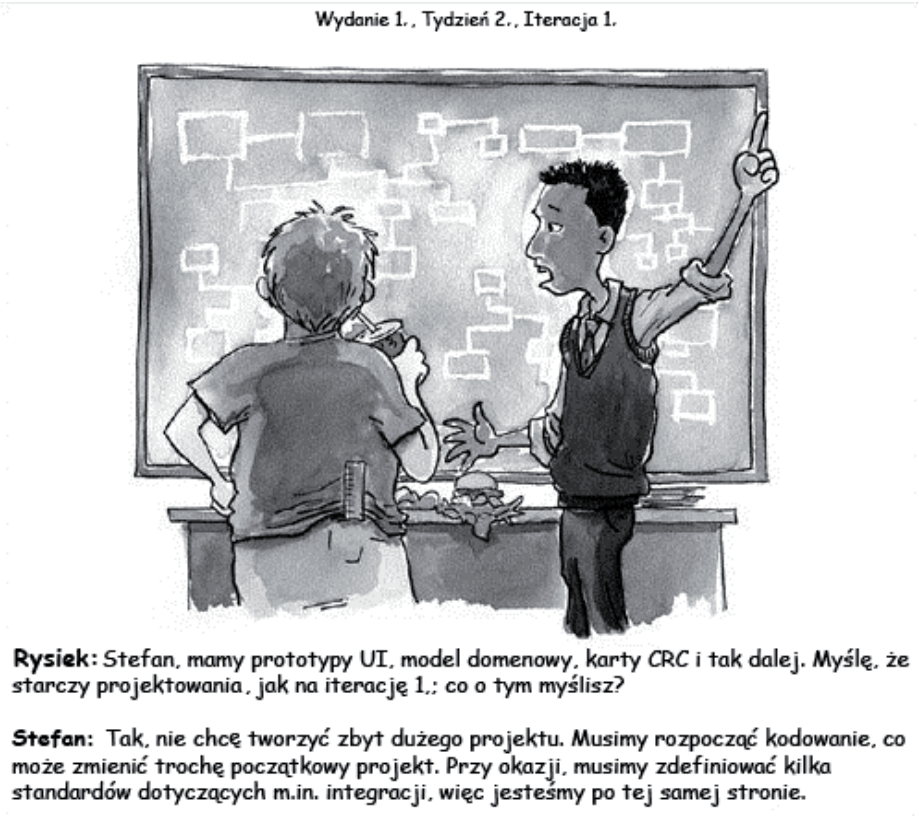
Zwiększ swą wydajność dzięki zastosowaniu efektywnych procesów i narzędzi do tworzenia oprogramowania w języku Java.

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 3.

Metodologie XP i AMDD



W tym rozdziale w końcu zabierzemy się za technologiczną stronę projektu, więc od teraz zaczyna się prawdziwa zabawa.

W prawdziwie iteracyjnym środowisku programistycznym wszystkie zagadnienia dotyczące architektury i projektowania nie muszą być koniecznie definiowane na samym początku. Refaktoryzacja (poprawianie kodu bez naruszania funkcjonalności) odgrywa

ważną rolę w stałej poprawie początkowo opracowanego projektu, ponieważ ciągle będziesz znajdował lepsze sposoby implementacji pewnych funkcji. Ponadto kiedy zakres projektu jest wstępnie zdefiniowany, to wymagania użytkownika mogą ciągle się rozwijać; jest to lepsze rozwiązanie niż blokowanie z góry jakichkolwiek zmian. Dobrym pomysłem, oprócz zmiennych wymagań, jest interakcja z użytkownikami, dająca możliwość zadawania pytań.

Chociaż część pracy może zostać wykonana na początku, na przykład historie użytkownika, wysokopoziomowa architektura, prototypy interfejsu użytkownika, model domenowy, standardy itd., to inne zagadnienia projektowe powinny być rozwiązane w iteracji, która obejmuje dany problem. Ponadto, jak będziesz mógł zobaczyć w rozdziałach 5. i 7., wstępne pisanie testów może również pomóc w projektowaniu Twoich klas. Nie musisz więc mieć dopracowanych wszystkich szczegółów na samym początku; inaczej mówiąc, możesz stosować przy projektowaniu podejście „w locie" (ang. *just-in-time*).

Jednakże należy opracować jakikolwiek wstępny projekt, być może w iteracji 0 (być może będziesz chciał zaprezentować dowód koncepcji projektu (ang. *proof-of-concept*), pokazać, jak technologie działają od początku do końca, od interfejsu użytkownika do bazy danych).

Prawdopodobnie w iteracjach 1. i 2. zostanie zaimplementowana mniejsza liczba historii użytkownika, ponieważ potrzeba dodatkowego czasu na projekt i przygotowanie środowiska pracy — w tym model domenowy (omówiony później), definicje obiektów biznesowych, konwencje nazw Javy, proces (skrypt) budowania (integracji) wspólny dla zespołu itd.

W tym rozdziale chcę pokazać od początku do końca metodologię AMDD (Agile Model Driven Development; *agilemodeling.com*) i metody programowania ekstremalnego (XP; *extremeprogramming.org*).

Co jest omawiane w tym rozdziale?

W tym rozdziale zrealizujemy założenia architektury i projektu przyjęte dla naszej przykładowej aplikacji Time Expression:

- Przygotujemy diagram architektury.
- Zbadamy obiekty, korzystając z kart CRC.
- Opracujemy mapę witryny (diagram przepływu UI).
- Wykonamy diagramy klas i pakietów dla Time Expression.
- Założymy strukturę katalogów do przechowywania naszych plików (które utworzymy w kolejnych rozdziałach).
- Przyjrzymy się krokom, które wykonamy w następnych rozdziałach w celu kompletnego oprogramowania naszych ekranów.

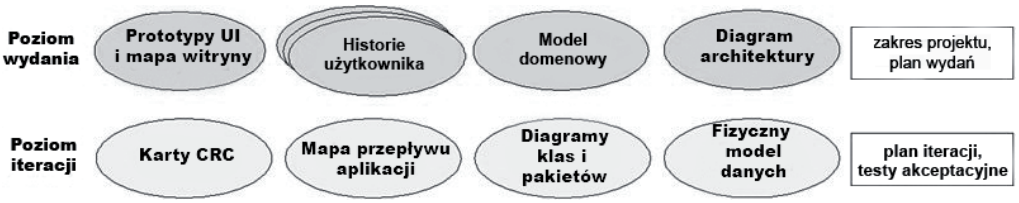
- Przejrzymy listę zaawansowanych pojęć, które będziemy musieli rozważyć, tworząc naszą przykładową aplikację: obsługa wyjątków, zadania zaplanowane, zarządzanie transakcjami, zapis logów i inne.

Projektowanie i wybór artefaktów

W poprzednim rozdziale przyjrzeliśmy się metodologii XP i jej zastosowaniu przy definiowaniu wymagań biznesowych i pracy z klientem. W tym rozdziale pójdziemy o krok dalej i opracujemy minimalną architekturę i projekt, aby ułatwić pracę przy tworzeniu Time Expression, korzystając z popularnych technologii: Hibernate, Spring Framework, Eclipse SDK i wielu innych związanych z nimi narzędzi, na przykład Anta, JUnit itp.

Jeśli natknąłeś się na mit, że programiści XP nie projektują i nie dokumentują swojej pracy, to mam nadzieję, że zmienisz błędne zdanie na ten temat pod koniec rozdziału. Pozwól, że przedstawię Ci podstawowe informacje na ten temat.

Spójrz na rysunek 3.1, który pokazuje kilka możliwych artefaktów, jakie możesz zrealizować w kolejnej wersji lub iteracji. Artefakty na poziomie wydania są to te, które tworzysz przed nowym wydaniem; analogicznie artefakty na poziomie iteracji są wytwarzane przed każdą iteracją. Nie jest to obowiązkowe dla każdego projektu, więc możemy wybrać i używać tego, czego naprawdę potrzebujemy. Jednakże pomiędzy rozdziałem 2., „Przykładowa aplikacja: Time Expression", a tym rozdziałem zdecydowałem się zaprezentować jak najwięcej praktycznych informacji o Time Expression. Na końcu tego rozdziału pokażę inny diagram, który będzie wiązać razem wszystkie artefakty, które powstały w wyniku naszych wysiłków pomiędzy poprzednim i tym rozdziałem (ale nie oszukuj i nie zaglądaj tam teraz, ponieważ jest to szczegółowy diagram i nie chciałbym Cię nim teraz przytłoczyć).



Rysunek 3.1. *XP (AMDD) — poziom wydania i poziom iteracji*

To, co zobaczysz w tym rozdziale, da Ci przynajmniej pewien punkt widzenia. Ten proces może (lub też nie) zadziałać w Twoim przypadku. Jednakże muszą być jakieś pozytywne aspekty użycia tych metodologii, ponieważ deweloperzy uwielbiają je i widziałem wiele pomyślnie ukończonych projektów, które były wynikiem ich użycia. Ponadto w naszym przypadku artefakty przedstawione w tym rozdziale są istotne dla reszty tej książki i omówiony proces może nam tutaj pomóc.

Jak widać na rysunku 3.1, mamy do stworzenia w tym rozdziale kilka artefaktów, więc zacznijmy pracę. Jednakże zanim zaczniemy, chciałbym dostarczyć dwóch punktów widzenia użytkowników XP.

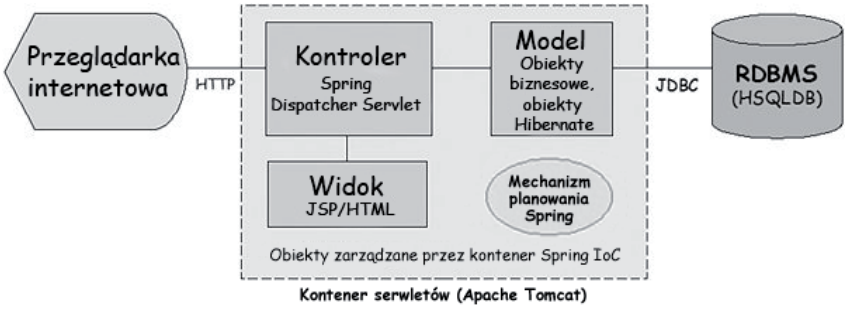
Szef projektu pracujący w firmie Fortune 50 powiedział mi ostatnio: „Kiedy zaczynamy iterację, to pierwszy jej dzień zwykle spędzamy na przeglądaniu historii i dzieleniu ich na zadania. Ćwiczenie polegające na dzieleniu zadań jest tak naprawdę sesją projektową. Na koniec widać, że około 20% czasu każdej iteracji deweloperzy poświęcają na projektowanie. Jeśli dodasz czas wykorzystywany przez wszystkich deweloperów w czasie wszystkich iteracji, to otrzymasz dużą liczbę, która obala stwierdzenie «bez projektu»".

Aby dać Ci również inny punkt widzenia na metodologię XP, przedstawię rozważania starszego architekta, który od dłuższego czasu pracuje w branży IT i ukończył wiele projektów przy pomocy metodologii XP i AMDD: „Jest również inny poziom projektowania XP — dzienny. Refaktoryzacja jest częścią projektowania. Chociaż początkowy projekt iteracji jest ważnym krokiem, to projektowanie po napisaniu kodu tworzy różnice pomiędzy projektem dobrym a naprawdę eleganckim".

Różnice tkwiące w podejściu do XP dotyczą architektury i projektowania w całym procesie wytwarzania, a nie tylko na początku projektu. Inaczej mówiąc, aplikacja ciągle ewoluuje w ciągu różnych iteracji. Korzyścią płynącą z takiego podejścia jest to, że projektowanie dotyczy tego, co aktualnie tworzysz, a nie pochłania od trzech do sześciu miesięcy przed kodowaniem, w sytuacji gdy wymagania mogą ewoluować w dzisiejszym ciągle zmieniającym się świecie.

Diagram architektury

Rysunek 3.2 przedstawia diagram opisujący architekturę naszej przykładowej aplikacji. Zauważ, że został zmieniony na formę elektroniczną z wersji tablicowej, jaką widziałeś w poprzednim rozdziale. Konwersja na formę elektroniczną jest kwestią gustu; mógłbyś zrobić zdjęcie aparatem cyfrowym diagramu znajdującego się na tablicy, ale osobiście wolę jasne i czytelne diagramy.



Rysunek 3.2. *Diagram architektury dla Time Expression*

Architektura jest dosyć prosta. Mamy trzy standardowe warstwy architektury sieciowej z warstwą klienta (przeglądarka WWW), warstwą środkową (serwer aplikacji) i naszą warstwą danych (baza danych).

Ponadto używamy standardu model-widok-kontroler (MVC); jest to wzorzec projektowy stosowany w większości aplikacji sieciowych bazujących na języku Java. Kontroler jest punktem wejścia żądań sieciowych (HTTP), kontroluje widok i model. Model zajmuje się danymi, które zostały uzyskane z kontrolera i są przekazywane do widoku, ten z kolei jest odpowiedzialny za wyświetlenie danych. W naszym przypadku widok stworzymy za pomocą JSP (JavaServer Pages).

To, co czyni naszą architekturę interesującą, to nie fakt, że korzysta ze wzorca MVC, ale to, że warstwa środkowa zawiera Spring Framework i Hibernate, dwie technologie omówione szczegółowo w dalszej części tej książki. Hibernate, jak będziesz mógł zobaczyć później, zapewnia proste utrwalanie obiektów w bazie danych, ponieważ pozwala traktować tabele i rekordy jak *plain old Java objects* (POJO). Spring Framework (*springframework.org*) dostarcza wielu korzyści, szczególnie jeśli pracujesz z obiektami POJO. Na przykład użyjemy Spring MVC jako naszego frameworka sieciowego, ponieważ chcemy uczynić kod bardziej przejrzystym (w porównaniu na przykład ze Struts). Inną wartą uwagi cechą frameworka Spring jest wsparcie dla zadań zaplanowanych, niezależnych od zewnętrznych usług, takich jak CRON w systemach uniksowych lub *Harmonogram zadań* w Windows. Oczywiście, główną zaletą jest dostarczana przez Spring funkcja odwrócenia kontroli (ang. *inversion of control — IoC*), o której dowiemy się więcej w dalszych rozdziałach.

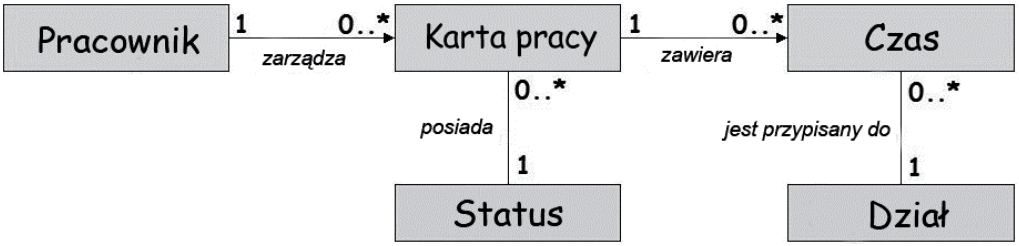
Od historii użytkownika do projektu

Omówiliśmy różne historie użytkownika w rozdziale 2. Dla zachowania zwięzłości nie zaimplementujemy ich wszystkich w tej książce. Jednakże historie, które wybrałem, dostarczą Ci kompletnych, działających przykładów, obejmujących ekrany zarówno zawierające formularze, jak i niezawierające ich. Dodatkowo przyjrzymy się zawansowanym elementom takim jak implementacja zabezpieczeń za pomocą obiektów przechwytujących (ang. *interceptors*), wysyłanie e-maili i zadania zaplanowane, które dotyczą kilku historii omówionych w rozdziale 2.

W dalszej części rozdziału dostarczę przykładów bazujących na dwóch historiach użytkownika, nazwanych w rozdziale 2. „Wprowadź godziny" i „Lista kart pracy".

Analizujemy klasy przy pomocy kart CRC

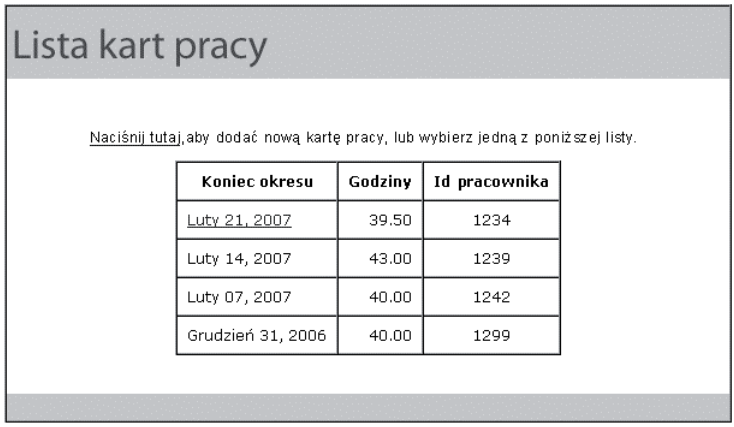
Rysunek 3.3 pokazuje model domenowy wykonany w rozdziale 2. Model domenowy umożliwia nam analizę obiektów biznesowych lub domenowych. Historie użytkownika umożliwią nam analizę klas kontrolerów internetowego interfejsu użytkownika. Tak więc przyjrzyjmy się obiektom, które chcemy wykorzystać do implementacji historii „Lista arkuszy", abyśmy mogli się dokładnie przyjrzeć, jak działają karty CRC.



Rysunek 3.3. *Model domenowy dla Time Expression*

Rysunek 3.4 pokazuje prototyp UI „Lista kart pracy" z rozdziału 2. Jak wspomniałem wcześniej, wiemy, że interfejs użytkownika będzie się opierał na przeglądarce WWW i że będziemy korzystać z wzorca MVC. Tak więc przyjrzyjmy się naszym klasom z punktu widzenia wzorca projektowego MVC.

Rysunek 3.4. *Ekran „Lista kart pracy"*



W modelu znamy już nazwy niektórych obiektów Time Expression — znajdują się one w naszym modelu domenowym. Znamy również nazwę historii użytkownika dla kontrolera (w tym przypadku *Lista kart pracy*) z rozdziału 2. Wiedząc to wszystko, możemy zająć się naszymi pierwszymi klasami, korzystając z kart CRC.

W tym przypadku karty CRC określają nazwę klasy, odpowiedzialność i klasy współpracujące. Tabela 3.1 pokazuje budowę przykładowej karty CRC wraz z wyjaśnieniami dotyczącymi trzech komponentów, które tam widać. Zauważ, że chociaż skorzystałem tutaj z wersji elektronicznej, to karty możesz wykonać z papieru o wymiarach 7×12 cm i później przetworzyć je na formę elektronicznej (jeśli okaże się to konieczne).

Tabela 3.1. *Budowa prostej karty CRC*

Nazwa klasy (rzeczownik)	
Odpowiedzialność (obowiązki tej klasy, metody biznesowe, obsługa wyjątków, metody związane z bezpieczeństwem, atrybuty, zmienne)	Współpracownicy (inne klasy wymagane do dostarczenia kompletnego rozwiązania)

Karty CRC dostarczają nieformalnej techniki do zorientowanej obiektowo analizy interakcji zachodzących pomiędzy klasami. Lubię karty CRC, ponieważ mogą być użyte w nieformalnej sesji z deweloperami lub użytkownikami do analizy obiektów bez potrzeby użycia komputera. Ponadto karty CRC mogą być stosowane w celu utworzenia formalnego diagramu klas, jeśli będzie to konieczne (zrobimy do w dalszej części tego rozdziału).

Tabele od 3.2 do 3.4 pokazują przykładowe karty CRC dla klas, które stworzymy później w tym rozdziale; zostały przedstawione tutaj, abyśmy mogli zapoznać się z wymaganiami dla ekranu *Lista kart pracy*.

Tabela 3.2. *Przykładowa karta CRC dla klasy Timesheet*

Timesheet
Zna datę końca okresu
Zna czas
Zna kod działu

Tabela 3.3. *Przykładowa karta CRC dla klasy TimesheetManager*

TimesheetManager
Pobiera kartę(y) pracy z bazy danych
Zapisuje kartę pracy w bazie danych

Tabela 3.4. *Przykładowa karta CRC dla klasy TimesheetListController*

TimesheetListController
Kontroler (ze wzoru MVC) służący do wyświetlania listy kart pracy

Omówiliśmy podstawowe pojęcia związane z kartami CRC. Możemy teraz pójść krok dalej.

Mapa przepływu aplikacji (artefakt domowej roboty)

W starszych projektach korzystałem z tabeli podobnej do tabeli 3.5. Jej format opracowałem sam, nie jest on zdefiniowany. Nazwałem ją mapą przepływu aplikacji, ponieważ pokazuje mi, w jaki sposób będzie działać interfejs użytkownika od początku do końca (jakimi ścieżkami może poruszać się użytkownik). Ta technika również dobrze odwzorowuje historie użytkownika na widoki (litera V we wzorcu MVC), które z kolei odwzorowuje na kontroler i ostatecznie na obiekty modelu.

Tabela 3.5. *Przykładowa mapa przepływu aplikacji*

Nazwa historii	Widok	Klasa kontrolera	Współpracownicy	Zależne tabele
Lista kart pracy	`timesheetlist`	`TimeSheetListController`	TimesheetManager	Timesheet
Wprowadź godziny	`enterhours`	`EnterHoursController`	TimesheetManager	Timesheet Department

Technika uzupełniająca

Porównując tę mapę przepływu aplikacji do diagramów klas lub kart CRC, zauważysz, że uzupełnia ona karty CRC i diagramy klas. Lista kart CRC, która zawiera między innymi zakres odpowiedzialności klas, uzupełnia braki mapy przepływu aplikacji. Z drugiej strony diagramy klas pokazują związki, zachowania (metody), atrybuty i prawdopodobnie jeszcze więcej niż zawiera taka mapa.

Przy połączeniu wszystkiego razem w postaci tabeli, możemy łatwo znaleźć nazwy klas (nawet przy dużych projektach) i łatwo je sortować przy pomocy arkusza kalkulacyjnego lub innych narzędzi.

Rozszerzamy mapę przepływu aplikacji o kolumny CRUD

Ta tabela może być również użyta do historii niezawierających interfejsu użytkownika, takich jak „Przypomnienie e-mailowe: Pracownik". Na przykład kolumny odpowiedzialne za widok i kontroler mogą być zastąpione kolumną nazwaną „Zadanie".

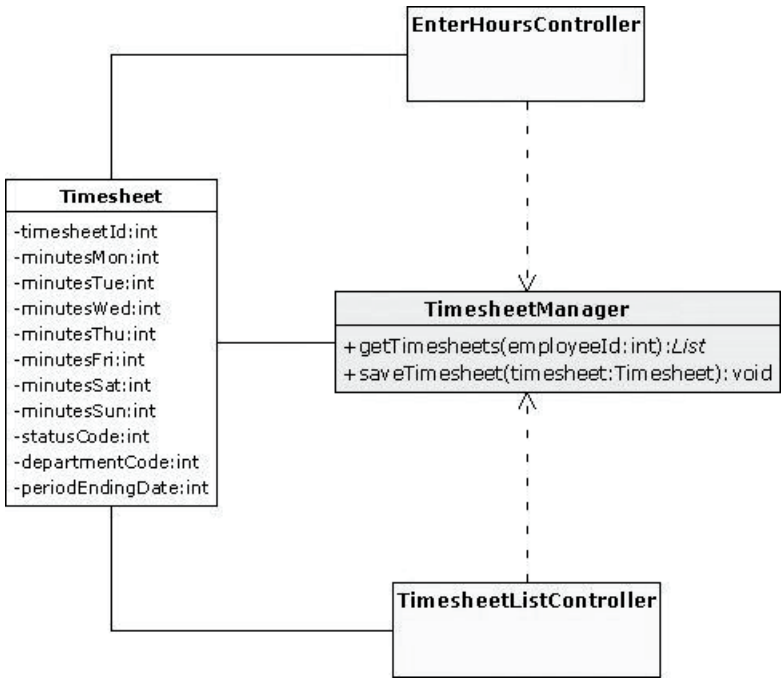
Ponadto możesz rozszerzać tę tabelę, dzieląc kolumnę „Zależne tabele" na cztery kolumny CRUD (od ang. *create* — tworzenie, *read* — czytanie, *update* — aktualizacja, *delete* — usuwanie). To pozwoli pokazać nie tylko tabele, do których się odwołujemy, ale również to, *w jaki sposób* wpływają na nie poszczególne klasy. Dodając kolumny CRUD, zasadniczo dostarczasz pełnej mapy przepływu (od widoku do bazy danych i z powrotem) w jednym wierszu naszej tabeli.

UML — diagram klas

W dalszej części zajmiemy się podstawowym diagramem klas. Moim zdaniem jest to opcjonalny krok (patrz ramka poniżej), ponieważ nasze karty CRC i mapa przepływu aplikacji dostarczą dostatecznej ilości informacji do implementacji klas. Jednakże diagramy klas mogą być dobrą rzeczą, kiedy użyjemy ich w odpowiedni sposób.

Rysunek 3.5 pokazuje przykładowy i minimalny diagram klas dla klas zdefiniowanych do tej pory.

Rysunek 3.5.
Przykładowy diagram klas dla Time Expression



Osobista opinia: Diagramy UML

Przez wiele lat używałem różnych rodzajów diagramów UML, włączając w to klasyczne diagramy UML, diagramy pakietów (moje ulubione) i rzadko używane diagramy wdrożenia.

Jest również rodzaj diagramów, których nie jestem wielkim fanem: diagramy sekwencji. Nie lubię ich, ponieważ zauważyłem, że szybko stają się zbyt złożone i nieporęczne. Jednakże będę pierwszym, który Ci powie, że nie znam innego, lepszego sposobu niż te diagramy (w każdym razie jeszcze nie, ale pracuję obecnie nad lepszymi sposobami modelowania (tworzenia) diagramów. Sprawdź stronę *visualpatterns.com*, jeśli interesuje Cię ten temat).

Tymczasem staram się używać diagramów UML wtedy, kiedy wydają się odpowiednie, ponieważ myślę, że są cenne, gdy są użyte w odpowiednim miejscu i w odpowiednim czasie. W rzeczywistości sądzę, że diagramy UML są najbardziej użyteczne, kiedy są generowane do dokumentacji na postawie istniejącego już systemu (na przykład podczas przekazywania systemu).

Mam nadzieję, że nie przedstawiłem UML w całkowicie złym świetle, bo nie to było moim celem, zwłaszcza że przez wiele lat wielu inteligentnych ludzi pracowało nad tym, aby powstał standard UML. (W rzeczywistości opieram całe moje badania na pracy wykonanej wcześniej, zamiast wymyślać od nowa koło).

Moim głównym zarzutem wobec diagramów UML jest to, że stają się szybko zbyt skomplikowane, szczególnie w przypadku dużych projektów. Innym problemem związanym z UML jest fakt, że wymaga specjalnych narzędzi, które są zwykle komercyjne i podnoszą koszty firmy. Dodatkowo niektóre z tych narzędzi mają stromą krzywą uczenia i w związku z tym wymagają specjalnego szkolenia pracowników (najlepszym przykładem jest Rational Rose), w wyniku czego znowu podnoszą się koszty utrzymania projektu.

Ponadto proste narzędzia, takie jak OpenOffice.org, Microsoft PowerPoint, Microsoft Visio, umożliwiają łączenie różnych figur (na przykład prostokątów) za pomocą łączników, które są zachowywane nawet w przypadku zmiany położenia któregoś obiektu. Są to potężne narzędzia, ponieważ pozwalają w prosty sposób tworzyć diagramy przepływu. Często korzystam z łączników, jak widać na diagramach umieszczanych w tej książce; w rzeczywistości prawie wszystkie diagramy zrobiłem, korzystając z OpenOffice.org!

Ponadto często stosuję praktyki rekomendowane przez metodologię Agile, na przykład modelowanie w celu osiągnięcia określonego efektu lub tworzenie wystarczająco dobrych artefaktów. Idąc dalej, uaktualniam je tylko wtedy, gdy jest to niezbędne, ponieważ wiele artefaktów można wyrzucić, gdy spełnią swoje zadanie. Po zaimplementowaniu zaprojektowanego kodu posiadasz już gotową dokumentację — tak, sam kod. (Jak wspominałem wcześniej, na podstawie kodu jesteśmy w stanie stworzyć niezłe diagramy klas i inne).

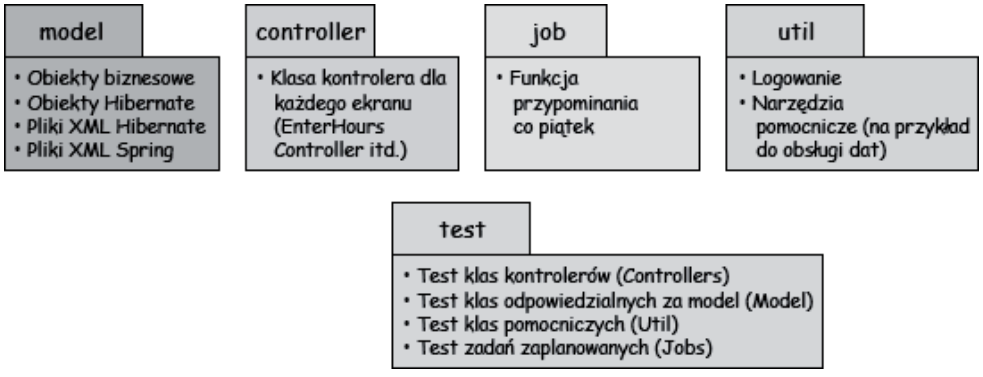
Głównym powodem tego, że ciężka dokumentacja wydaje się czystym obłędem, jest to, iż nie jestem w stanie sobie przypomnieć projektu, który utrzymał do samego końca kompletną dokumentację. Dzieje się tak, ponieważ w szybko zmieniającym się świecie, w którym zdarzają się nierealistyczne terminy ukończenia projektów, bardzo trudno jest utrzymać aktualną dokumentację.

Podsumowując, używaj diagramów UML tam, gdzie są odpowiednie, ale nie wstydź się i nie wahaj użyć prostszych, lecz efektywniejszych własnych diagramów. Na koniec chciałem zacytować jedno z haseł reklamowych ze strony *agilemodeling.com*: „Twoim celem jest budowanie ogólnego zrozumienia, a nie tworzenie szczegółowej dokumentacji".

Diagramy pakietów UML

W przypadku naszej przykładowej aplikacji, Time Expression, korzystamy z prefiksu `com.visualpatterns.timex` dla nazw pakietów.

Jeśli pracowałeś już z Javą, to prawdopodobnie wiesz, że pierwszy człon nazwy pakietu to zazwyczaj nazwa domeny pisana od końca. Na przykład nazwa pakietu `com.visualpatterns` pochodzi od nazwy domeny *visualpatterns.com*. Część `timex` w nazwie pakietów pochodzi od skróconej nazwy naszej przykładowej aplikacji. Reszta przyrostków nazw jest widoczna na rysunku 3.6 przedstawiającym diagram pakietów UML.



Rysunek 3.6. *Diagram pakietów UML dla Time Expression*

Wybrałem bardzo proste nazwy dla pakietów Javy, aby pasowały do wzorca bazującego na MVC. Mogliśmy nazwać pakiet zawierający model na przykład czymś w stylu domeny, ale preferuję prostotę i jednoznaczność; staram się dopasować nazwy pakietów do nazw przyjętych w architekturze lub mapie przepływu aplikacji. W ten sposób każdy nowy człowiek, który zajmie się moim kodem, od razu zrozumie jego organizację. Tak więc w pełni kwalifikowana nazwa naszego pakietu zawierającego model będzie miała postać `com.visualpatterns.timex.model`.

Jak możesz się domyślać, pakiet kontrolera posiada w sobie klasy związane z kontrolerem. Pakiet `job` zawiera nasze zadania zaplanowane, odpowiedzialne za wysłanie przypomnienia. Pakiet `util` zawiera klasy z kodem pomocniczym.

Ostatnim, ale nie mniej ważnym pakietem, jest `test`, który będzie zawierał kod testów jednostkowych. Chociaż ja wolałem umieścić testy w osobnym pakiecie, to wielu deweloperów decyduje się umieszczać kod testowy w tych samych katalogach co testowana klasa. Jest to kwestia gustu, ale moim zdaniem oddzielenie pakietu (katalogu) z klasami testowymi pozwala utrzymać jasną i przejrzystą strukturę pozostałych pakietów.

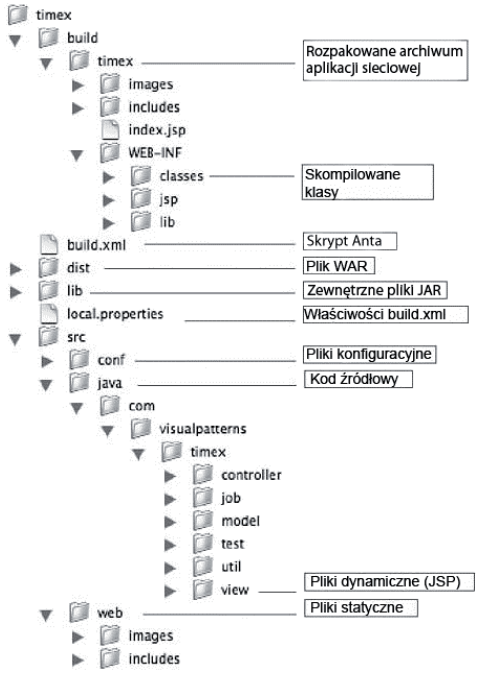
Struktura katalogów

Rysunek 3.7 pokazuje strukturę katalogów, jakiej użyjemy w naszej przykładowej aplikacji. Powinno to wyglądać dosyć prosto i znajomo; najważniejsze podkatalogi to *src*, *build*, *lib* i *dist*. Będę się odwoływał do tego rysunku w dalszych rozdziałach (na przykład w rozdziałach 4., 5. i 7.), a podane w tym rozdziale katalogi będą szerzej omówione, kiedy przyjdzie taka potrzeba. Tymczasem rysunek 3.7 dostarcza krótkiego opisu kluczowych katalogów.

Przykładowe nazwy plików

Mając strukturę katalogów widoczną na rysunku 3.7, możemy teraz przejść do utworzenia kilku przykładowych nazw plików dla klas omawianych wcześniej w tym rozdziale. Na przykład, implementując ekran *Lista kart pracy*, omawiany wcześniej w tym rozdziale, utworzymy następujące pliki w katalogu *timex/src/java/com/visualpatterns/timex*:

- *controller/TimesheetListController.java*
- *model/Timesheet.java*
- *model/TimesheetManager.java*
- *test/TimesheetListControllerTest.java*
- *test/TimesheetManagerTest.java*
- *view/timesheetlist.jsp*



Rysunek 3.7.
Struktura katalogów dla Time Expression

Wytwarzanie oprogramowania od początku do końca

Podsumowując wszystko, czego się do tej pory nauczyliśmy w tym rozdziale, możemy zapisać kolejne kroki wymagane do zaimplementowania (utworzenia) naszej pierwszej historii, od interfejsu użytkownika do bazy danych. Poniżej zamieszczam zadania wymagane do ukończenia pierwszej historii użytkownika:

- Skonfiguruj środowisko pracy, włączając w to JDK, Ant i JUnit (w rozdziale 4.).
- Napisz testy i implementację klas znajdujących się w pakiecie `model` (korzystając z Hibernate w rozdziale 5.).
- Napisz testy i implementację klas znajdujących się w pakiecie `controller` (korzystając z frameworka Spring w rozdziale 7.).

Testy akceptacyjne

Testy akceptacyjne mogą dostarczyć nam wielu szczegółowych wymagań, tak jak to dzieje się w wielu projektach opartych na metodologii Agile. Jednym z przykładów jest lista prawidłowych operacji, jakie może wykonać użytkownik na danym ekranie.

Pomysł, aby korzystać z testów akceptacyjnych jako specyfikacji wymagań, jest sensowny, ponieważ te testy określają to, czego oczekuje klient od naszej aplikacji. W naszym przypadku użyjemy ich tylko do testów jednostkowych, aby ustalić szczegółowe wymagania. Jednakże jest wiele więcej zastosowań testów akceptacyjnych w prawdziwym świecie.

Poniższy podrozdział jest listą naszych testów akceptacyjnych i elementów historii, które będziemy implementować.

Logowanie

- Identyfikator pracownika może mieć długość do 6 znaków. Hasło musi się zawierać w 8 – 10 znakach.
- Tylko użytkownicy o prawidłowych identyfikatorach mogą się logować.

Lista kart pracy

- Użytkownik ma dostęp tylko do swoich kart pracy.

Wprowadź godziny

- Godziny muszą być wprowadzane w formacie numerycznym.
- Dzienna liczba godzin nie może przekraczać 16. Tygodniowa liczba godzin nie może przekraczać 96.
- Zapłata za godziny wpływa do określonego działu.
- Godziny mogą być wprowadzone w postaci dziesiętnej, z dwoma cyframi po przecinku.
- Pracownik może oglądać i edytować tylko własne karty pracy.

Inne rozważania

Jak wspomniałem wcześniej, potrzebujemy opracować minimalny projekt i architekturę, aby można było zacząć. Chociaż stworzyliśmy rozsądnej wielkości architekturę i projekt, w tym rozdziale zostało wiele nieomówionych elementów:

- Bezpieczeństwo aplikacji — omówię ten problem w rozdziale 7., „Framework Spring Web MVC”, i 10., „Elementy zaawansowane”.
- Zarządzanie transakcjami — omówię ten element w rozdziale 5. Zobaczysz, jak zaimplementować programowe zarządzanie transakcjami z użyciem Hibernate.

- Obsługa wyjątków — w rozdziale 10. przyjrzymy się sprawdzalnym i niesprawdzalnym wyjątkom; przyjrzymy się też, kiedy warto używać jednych, a kiedy drugich.
- Inne elementy — funkcje wymagane przez Time Expression, na przykład zadania odpowiedzialne za wysyłanie e-maili, omawiane w rozdziale 10. W dalszych rozdziałach będą omawiane również inne tematy, włączając w to zapis logów, biblioteki znaczników i inne.

Duży początkowy projekt czy refaktoryzacja?

Odnosząc się do słów Martina Fowlera (*refactoring.com*), refaktoryzacja „to technika służąca do restrukturyzacji kodu, zmieniająca jego wewnętrzną strukturę bez zmiany zewnętrznego interfejsu". Wielu deweloperów od lat korzysta z refaktoryzacji kodu, ale dopiero Martin Fowler nadał tej technice formalną nazwę (i cieszę się, że to zrobił).

Kiedy zaczniesz pisać kod aplikacji, prawdopodobnie znajdziesz kilka lepszych sposobów na zrealizowanie niektórych funkcji. Na przykład możesz usunąć nadmiarowy kod lub przenieść powtarzający się kod do klasy nadrzędnej. W związku z tym głęboko wierzę, że refaktoryzacja powinna zostać otwartą opcją nie tylko w przypadku kodu, ale również w przypadku bazy danych, architektury, dokumentacji, skryptów do integracji (budowania) i innych. Uwalnia nas to również od tworzenia idealnego projektu i architektury na samym początku pracy.

Ostatnio przeczytałem esej na stronie *agiledata.org*, który podsumowuje ten temat. Warty uwagi jest fragment: „Deweloperzy Agile przechodzą w tę i z powrotem pomiędzy zadaniami takimi jak modelowanie danych, modelowanie obiektów, refaktoryzacja, odwzorowania, implementacja i poprawa wydajności".

Weźmy na przykład tę książkę. W zasadzie dla mnie to jest projekt, ponieważ pisząc książkę, tworzę przykładowy projekt. Część projektu opracowałem na początku, ale nie miałem odpowiedzi na wszystkie pytania, kiedy rozpoczynałem pisanie; nie martwiło mnie to jednak, ponieważ mogłem refaktoryzować architekturę, projekt, kod lub sam proces użyty do tworzenia Time Expression w dalszych rozdziałach, a chciałem ruszyć do przodu, zamiast stać w miejscu, próbując przewidzieć każdy możliwy scenariusz.

W skrócie — zdecydowanie powinieneś opracować niewielki początkowy projekt i architekturę, ale pamiętaj że możesz w każdym momencie zmienić lub ulepszyć początkowe założenia, możesz uprościć kod i o ile nie jest za późno (na przykład nie jest to dzień testów akceptacyjnych lub wdrożenia), to możesz refaktoryzować.

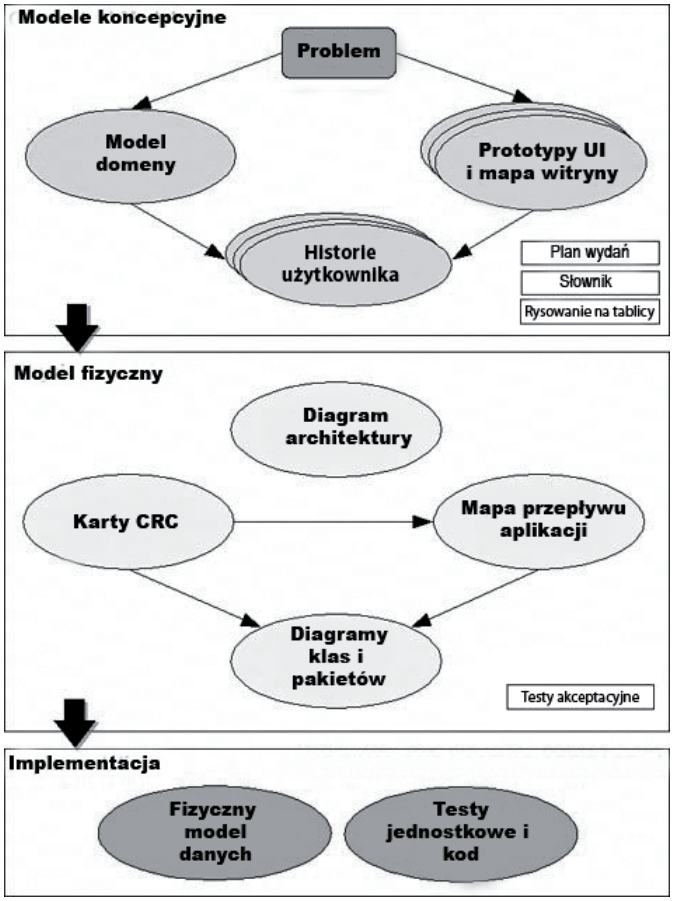
Podsumowanie

W tym rozdziale zrealizowaliśmy założenia architektury i projektu przyjęte dla naszej przykładowej aplikacji Time Expression:

- Opracowaliśmy diagram architektury.
- Zbadaliśmy obiekty, korzystając z kart CRC.
- Wykonaliśmy mapę witryny (mapę przepływu aplikacji).
- Opracowaliśmy diagramy klas i pakietów dla Time Expression.

- Założyliśmy strukturę katalogów do przechowywania naszych plików (które utworzymy w kolejnych rozdziałach).
- Przyjrzeliśmy się kolejnym etapom tworzenia naszych ekranów w następnych rozdziałach.
- Przejrzeliśmy listę zaawansowanych pojęć, które będziemy musieli rozważyć, tworząc naszą przykładową aplikację: obsługa wyjątków, zadania zaplanowane, zarządzanie transakcjami, zapis logów i inne.

Na początku rozdziału obiecałem, że przedstawię diagram pokazujący proces tworzenia niektórych artefaktów. (Czyżbyś oszukiwał i wcześniej zerknąłeś na niego?). Rysunek 3.8 pokazuje ten diagram. Oczywiście, dodatkowo jasno pokazuje artefakty XP na różnych poziomach — konceptualnym, fizycznym i nawet implementacyjnym. Zauważ, że linie na rysunku 3.8 są jednokierunkowe; pokazują, w jaki sposób tworzyliśmy artefakty w tym i poprzednim rozdziale. Jednakże w prawdziwym świecie te linie mogą przebiegać w dwie strony, ponieważ występują sprzężenia zwrotne.



Rysunek 3.8. *Koncepcja, model fizyczny i implementacja artefaktów dla Time Expression*

Ostatnia myśl związana z tematem artefaktów i dokumentacji. Zapamiętaj, że baza danych i kod to najważniejsze artefakty ze wszystkich. Nie jestem w stanie tego podkreślić w sposób wystarczający. Pozostałych artefaktów opisanych w tej książce możesz użyć lub nie (nie są niezbędne). Ponadto wiele z tych opcjonalnych artefaktów może być potencjalnie odrzuconych, ponieważ po tym, jak zostały wykorzystane, wielu ludzi ich nie aktualizuje. Jednakże kod jest zawsze aktualny, ponieważ jest tym, czym aplikacja dla klienta; baza danych przetrwa każdy program napisany do jej obsługi, a więc powinna być uważana za najważniejszy składnik systemu.

Skoro mówimy teraz o bazie danych i kodzie — nadszedł czas, aby zakasać rękawy i zacząć przygotowywać nasze środowisko pracy, korzystając z narzędzi Ant i JUnit w kolejnych rozdziałach, abyśmy mogli zacząć pisać właściwy kod.

Rekomendowane źródła

Poniższe strony dostarczają dodatkowych informacji na tematy omówione w tym rozdziale:

- Agile Model Driven Development: *http://www.agilemodeling.com*
- Agile Data: *http://www.agiledata.org*
- Programowanie ekstremalne: *http://extremeprogramming.org*
- Karty CRC: *http://c2.com/doc/oopsla89/paper.html*