

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Java. Obsługa wyjątków, usuwanie błędów i testowanie kodu

Autor: Stephen Stelting
Tłumaczenie: Adam Bochenek
ISBN: 83-7361-796-5
Tytuł oryginału: [Robust Java](#)
Format: B5, stron: 376



W każdej aplikacji mogą wystąpić błędy.

Przygotuj się na to pisząc odpowiednią obsługę wyjątków

- Poznaj techniki programowania, dzięki którym Twoje aplikacje staną się odporne na błędy
- Naucz się przewidywać błędy i zapobiegać ich występowaniu
- Zabezpiecz aplikacje przez skutkami wystąpienia błędów stosując odpowiednie wzorce projektowe

Żaden kod nie jest idealny – nawet najbardziej doświadczony programista popełnia błędy. Tym, co w takich przypadkach wyróżnia doświadczonego programistę jest fakt, że jego aplikacje posiadają mechanizmy pozwalające na obsługę tych błędów. Dzięki nim program nie zawiesi się ani nie „pogubi” żadnych danych. Oczywiście, aby napisać odpowiednią obsługę wyjątków, należy poznać ich typy oraz mechanizmy ich powstawania. Niezbędna jest również znajomość wzorców projektowych oraz narzędzi do testowania kodu.

W książce „Java. Obsługa wyjątków, usuwanie błędów i testowanie kodu” znajdziesz wszystkie te informacje. Dowiesz się, jak zaimplementować obsługę wyjątków i poprawić jakość kodu źródłowego. Poznasz modele wyjątków i nauczysz się zarządzać mechanizmami ich obsługi na poziomie aplikacji i pojedynczych modułów. Przeczytasz tu także o wzorach projektowych zapewniających prawidłową obsługę wyjątków.

- Koncepcje obsługi wyjątków
- Obsługa wyjątków w aplikacjach wielowątkowych
- Przyczyny występowania wyjątków w różnych elementach języka Java
- Programowanie rozproszone w Javie
- Wyjątki w aplikacjach J2EE
- Wzorce projektowe
- Testowanie kodu i usuwanie błędów

Po przeczytaniu tej książki Twoja wiedza pozwoli Ci na podejmowanie odpowiednich decyzje dotyczące architektury aplikacji i odpowiadającego jej modelu wyjątków.



Spis treści

Wstęp	11
Część I Podstawy obsługi wyjątków	17
Rozdział 1. Obsługa wyjątków. Wprowadzenie.....	19
Wstęp	19
Pojęcie wyjątku	22
Hierarchia klas reprezentujących wyjątki	24
Opcje związane z przechwytywaniem oraz deklaracją wyjątków.....	25
Przechwytywanie i obsługa wyjątków: bloki try, catch i finally	26
Reguły dotyczące stosowania bloków try-catch-finally.....	27
Deklarowanie metod zgłaszających wyjątki	28
Reguły dotyczące deklarowania metod zgłaszających wyjątki	28
Wyjątki kontrolowane i niekontrolowane	28
Podstawowe własności każdego wyjątku	29
Podsumowanie	31
Rozdział 2. Obsługa wyjątków. Techniki i praktyka	33
Wstęp	33
Kiedy przechwytywać wyjątki, a kiedy je deklarować?	34
Typowe sposoby obsługi wyjątków	35
1. Zapis błędu lub związanej z nim informacji do dziennika	36
2. Zwrócenie się do użytkownika z prośbą o podjęcie odpowiedniej decyzji	38
3. Użycie wartości domyślnych lub alternatywnych	39
4. Przekazanie sterowania do innej części aplikacji	40
5. Konwersja wyjątku do innej postaci	40
6. Zignorowanie problemu.....	41
7. Powtórzenie operacji.....	41
8. Wywołanie operacji alternatywnej	42
9. Przygotowanie aplikacji do zamknięcia	42
Co należy, a czego nie wolno robić w ramach obsługi wyjątków?	43
Dobre praktyki	43
1. Obsługuj wyjątki zawsze, gdy jest to możliwe	43
2. Obsługa wyjątku powinna ściśle zależeć od jego typu	43

3. Zapisuj do dziennika te wyjątki, które mogą mieć wpływ na działanie aplikacji	44
4. Dokonuj konwersji wyjątków, jeśli tylko dojdiesz do wniosku, że lepiej reprezentują one istotę problemu	44
Czego robić nie należy?	44
1. Nie ignoruj wyjątków, chyba że masz absolutną pewność, iż jest to niegroźne dla działania aplikacji	44
2. Nie dopuszczaj do zbyt ogólnego traktowania wyjątków	45
3. Nie dokonuj konwersji wyjątku polegającej na zmianie typu szczegółowego na ogólny (chyba że przechowasz kontekst zdarzenia w inny sposób)	46
4. Wyjątków lepiej unikać, niż je obsługiwać	47
Rozdział 3. Zaawansowane koncepcje obsługi wyjątków	49
Wstęp	49
Tworzenie własnych wyjątków	50
1. Definicja klasy wyjątku	51
2. Deklaracja wyjątku przez metody, które mogą być potencjalnym źródłem błędu	52
3. Znalazienie tych miejsc, które są bezpośrednią przyczyną wystąpienia błędu, wygenerowanie w nich wyjątków i wyrzucenie ich za pomocą instrukcji throw	52
Łańcuchy wyjątków	54
Lokalizacja wyjątków	55
1. Stworzenie klasy dziedziczącej po ResourceBundle, która będzie przechowywała opis wyjątku	56
2. Zdefiniowanie klasy odpowiadającej danemu językowi	57
3. Definicja klasy wyjątku, w której pokrywamy metodę getLocalizedMessage	57
Wyjątki a pokrywanie metod	59
Deklaracja wyjątków w interfejsach i klasach abstrakcyjnych	60
Stos wywołań wyjątku	61
Wyjątki z perspektywy kodu pośredniego	64
Rozdział 4. Obsługa wyjątków w aplikacjach wielowątkowych	69
Wstęp	69
Kiedy należy dzielić program na wątki?	70
Wyjątki w systemach wielowątkowych	71
Wyjątki w synchronizowanych blokach kodu	72
Praca wielowątkowa a ryzyko wystąpienia wyjątku	74
Wyjątki związane z komunikowaniem się wątków	75
Zakleszczenie	78
ThreadDeath	79
Rozdział 5. Zapis do dziennika i asercje	81
Wstęp	81
Wprowadzenie do Logging API	81
Kiedy stosować Logging API?	82
Struktura Logging API	82
Szczegóły dotyczące używania biblioteki	83
Konfigurowanie dziennika	88
Przykład pierwszy: Proste użycie domyślnego obiektu Logger	88
Przykład drugi: Użycie i konfiguracja kilku obiektów typu Logger	89
Przykład trzeci: Dziennik rozproszony	90
Asercje	91
Używanie asercji	92
Jak i kiedy korzystać z asercji?	92

Część II Planowanie obsługi wyjątków 95**Rozdział 6. Planowanie obsługi wyjątków97**

Wstęp	97
Zasady dobrego projektowania obiektowego	98
Projektowanie z uwzględnieniem błędów	103
Etap 1. Identyfikacja przypadku użycia	104
Etap 2. Precyzyjny opis procesu i poszczególnych jego składników	105
Etap 3. Identyfikacja potencjalnych błędów i ocena ich ryzyka	106
Etap 4. Wskazywanie miejsc wystąpienia potencjalnych błędów w ramach operacji	110
Etap 5. Przygotowanie strategii obsługi błędów	111
Projektowanie z myślą o łatwiejszym zarządzaniu — korzyści i wady	113
Identyfikacja błędów	114
Podsumowanie	115

Rozdział 7. Wyjątki a podstawowe elementy języka Java..... 117

Wstęp	117
Typy podstawowe	118
Wprowadzenie	118
Stosowanie	118
Potencjalne problemy	119
Ogólne zalecenia	121
Klasa Object i obiekty tego typu	122
Wprowadzenie	122
Potencjalne problemy	122
Tablice	128
1. Indeksowanie	129
2. Wielkość tablicy	129
3. Typ elementów tablicy	129
Interfejsy pakietu java.lang	130
Klasy String i StringBuffer	131
Klasa String	132
Klasa StringBuffer	132
Klasy BigDecimal i BigInteger	132
Klasy opakowujące	133

Rozdział 8. Kolekcje i operacje wejścia-wyjścia..... 135

Wstęp	135
Biblioteka obsługi kolekcji	135
Wprowadzenie	135
Korzystanie z kolekcji	136
Problemy i wyjątki typowe dla kolekcji	141
System wejścia-wyjścia	144
Problemy typowe dla operacji wejścia-wyjścia	146
Błędy i wyjątki w klasach wejścia-wyjścia	148
Problemy ogólne	148
Problemy dotyczące strumieni określonego typu	149
Biblioteka New I/O (NIO)	156
Wyjątki w New I/O API	158
Bufory	158
Zestawy znaków	159
Kanały	160
Przełączniki	164

Rozdział 9. Programowanie rozproszone w Javie	167
Wstęp	167
Podstawy komunikacji między aplikacjami	168
Modele komunikacji w Javie	170
Dodatkowe zagadnienia związane z pracą rozproszoną	170
Model obsługi wyjątków	172
Typowe problemy	173
RMI — zdalne wywoływanie metod	174
Model komunikacyjny RMI	174
Typowe zagadnienia związane z używaniem RMI	175
Model wyjątków RMI	178
Java Naming and Directory Interface, JNDI	184
Typowe zagadnienia związane z JNDI	185
Model wyjątków JNDI	186
Java Database Connectivity, JDBC	189
Wyjątki w JDBC	189
Ogólne zagadnienia dotyczące obsługi baz danych	191
Cykl życia aplikacji korzystającej z JDBC	193
Podsumowanie	198
Rozdział 10. J2EE	199
Wstęp	199
Podstawowy model aplikacji J2EE	200
Model aplikacji J2EE	202
Wyjątki w J2EE	203
Warstwa klienta	204
Warstwa aplikacji sieciowej	206
Deklaratywna obsługa błędów	207
Komponenty aplikacji sieciowej	208
Serwlety i filtry: obsługa wyjątków na poziomie programu	209
Model wyjątków serwetów i filtrów	210
Typowe zagadnienia i ryzyko związane z używaniem serwetów	211
Model wyjątków JSP	215
Bezpośrednie przekierowanie błędów w JSP	215
Sposób obsługi wyjątków w JSP	216
Błędy związane z translacją i uruchamianiem stron JSP	217
Biblioteki własnych znaczników	217
Warstwa komponentów EJB	218
Standardowe metody EJB	219
Ogólne kwestie związane z obsługą EJB	221
Zagadnienia związane z typem komponentu	223
Model wyjątków EJB	225
Zarządzanie cyklem życia komponentów. Metody wywoływane przez klienta	227
Metody komponentu wywoływane przez kontener	230
Wyjątki z punktu widzenia kontenera	231
Obsługa transakcji	232
J2EE i obsługa wyjątków — kwestie ogólne	236
Czynniki, które należy rozważyć podczas obsługi wyjątków w J2EE	237
Obsługa dziennika	237
Koszty związane ze zgłaszaniem wyjątków	238
Obciążenie sieci	239
Inne zalecenia dotyczące systemów J2EE	239

Część III Skuteczne posługiwanie się wyjątkami i ich obsługa 241**Rozdział 11. Architektura i projekt modelu obsługi wyjątków 243**

Wstęp	243
Dlaczego architekt powinien interesować się obsługą błędów?	244
Koszt awarii	245
Cena sukcesu	246
Architektura, projekt i rozwój	246
Role architekta i projektanta: cienka linia na piasku	247
Programista: na linii frontu	247
Kluczowe decyzje architektoniczne dotyczące modelu wyjątków	248
Priorytety i cechy architektury	249
Cechy modelu wyjątków	250
Strategia obsługi i propagacji wyjątków	250
Model klas reprezentujących wyjątki i błędy	253
Usługi uniwersalne	256
Podsumowanie	257
Kilka słów na temat kolejnych rozdziałów	258

Rozdział 12. Wzorce 259

Wstęp	259
Wzorce architektoniczne (oparta na wzorcach architektura oprogramowania, POSA-ASOP)	260
1. Wzorzec warstw	261
2. Model-widok-kontroler (MVC)	262
Wzorce projektowe „Bandy czworga”	264
Wzorce konstrukcyjne	264
1. Builder (budowniczy)	264
2. Singleton	266
Wzorce strukturalne	267
1. Adapter	267
2. Composite	268
3. Facade (fasada)	270
4. Proxy (pośrednik)	271
Wzorce czynnościowe	272
1. Chain of Responsibility (łańcuch odpowiedzialności)	272
2. Command (polecenie)	273
3. Observer (obserwator)	274
4. State (stan)	275
5. Strategy (strategia)	276
Wzorce projektowe J2EE	277
Warstwa integracyjna	278
1. Data Access Object, DAO (obiekt dostępu do danych)	278
Warstwa prezentacyjna	280
1. Front Controller (sterownik frontalny)	280
2. Intercepting Filter (wzorzec przechwytyjący)	281
Warstwa biznesowa	282
1. Service Locator (lokalizator usług)	282
2. Session Facade (fasada sesji)	283
Podsumowanie	283

Rozdział 13. Testowanie	285
Jaki jest cel testowania i dlaczego jest ono tak ważne?	285
Nieprawdziwe opinie na temat testowania	286
Mit pierwszy: „Nie ma potrzeby, by programiści testowali swój kod”	287
Mit drugi: „Programiści w pełni odpowiadają za poprawność kodu”	287
Mit trzeci: „Do zweryfikowania aplikacji wystarczy «jeden test»”	288
Mit czwarty: „Istnieje możliwość kompletnego przetestowania aplikacji”	288
Warianty testowania, czyli na zewnątrz i wewnątrz skrzynki.....	289
Co znajduje się w skrzynce? (Typy testów)	289
Podział ról i odpowiedzialności w procesie testowania	291
„Gdyby to było takie łatwe...”, czyli problemy z testowaniem w Javie	293
Testowanie w praktyce.....	294
W jaki sposób zinstytucjonalizować testowanie?	294
Organizacja struktury testów.....	298
Taktyka i technologia, czyli jak zarządzać testami i jak je przeprowadzać	303
Gotowe narzędzia obsługi testów	304
Zintegrowane środowiska programistyczne.....	305
Miara jakości aplikacji	306
Kiedy testy należy uznać za zakończone?	306
Ogólna miara testowania	306
Miara testowania stosowana przez programistów	307
Rozdział 14. Usuwanie błędów	309
Wstęp	309
Zaczarowane słowo: błąd.....	310
Zasady i praktyka debugowania	311
Strategie usuwania błędów.....	313
Widoczne wyniki oraz sposoby usuwania błędów	318
Techniki testowania — poziom niski.....	320
Techniki testowania — poziom średni	322
Techniki testowania — poziom wysoki	323
Dodatkowe kwestie związane z usuwaniem błędów	325
Debugowanie innych technologii	325
Wyjątki zgłaszane na poziomie infrastruktury	326
Dodatki	327
Dodatek A Wpływ obsługi i deklarowania wyjątków na szybkość działania aplikacji	329
Dodatek B Krótkie i łatwe wprowadzenie do JUnit	335
Dodatek C MyBuggyServlet — kwestie poprawności komponentu	347
Słownik używanych w książce terminów technicznych	349
Bibliografia	353
Skorowidz	357

6

Planowanie obsługi wyjątków

Wstęp

W poprzedniej części skoncentrowałem się przede wszystkim na czysto programistycznych aspektach posługiwania się wyjątkami. Są to umiejętności podstawowe, niezbędne każdemu, kto chce w swych aplikacjach obsługiwać sygnalizowane w ten sposób problemy. Jednak sama znajomość pewnych technik programistycznych nie jest jedynym czynnikiem mającym wpływ na skuteczne wykorzystanie możliwości, jakie daje nam mechanizm zgłaszania i obsługi wyjątków. Oprócz wiedzy technicznej, musimy znać bardziej ogólne zasady i wiedzieć, jak i kiedy je stosować.

Inaczej mówiąc, bycie doskonałym koderem nie gwarantuje wcale, że pisane przez nas aplikacje będą świetne. Dlatego w niniejszym rozdziale, podobnie jak w całej drugiej części książki, przeniesiemy nasze zainteresowania na nieco wyższy poziom abstrakcji. Zajmiemy się prawidłową strukturą obsługi wyjątków wewnątrz metod, pomiędzy metodami a także na poziomie współpracujących ze sobą komponentów aplikacji. Moim zamiarem jest prezentacja sposobów pozwalających dotychczas zdobyte umiejętności zamienić na dobrze zorganizowane i łatwe w utrzymaniu konstrukcje kodu. W tej części książki z programistów zmienimy się w projektantów aplikacji.

Ogromny błąd wielu twórców aplikacji polega na tym, że obsługę wyjątków traktują oni jak zło konieczne, przypominając sobie o tym jedynie wtedy, gdy zmuszą ich do tego sygnalizowane przez kompilator błędy. Tworzenie oprogramowania bez wcześniejszego, starannego rozważenia potencjalnych błędów, które mogą podczas jego eksploatacji wystąpić, przypomina konstruowanie samolotu (lub budynku mieszkalnego) bez sporządzenia odpowiednich planów i stosowania powszechnie znanych reguł sztuki. Zakładamy naiwnie, że powinno się udać, choć oczywiście rezultatu takiego zagwarantować nie sposób.

Gwoli sprawiedliwości nadmienić jednak należy, że słabość procesów tworzenia software'u polega między innymi na tym, iż wiele tradycyjnych metodyk lekceważy problem obsługi błędów. Czy należy się więc dziwić, że aplikacje projektowane przy ich użyciu wykazują w tej dziedzinie braki?

Wiele metodologii koncentruje się przede wszystkim na próbie stworzenia wydajnego, uniwersalnego i obiektowego modelu rozwiązania danego problemu. Nie zrozum mnie źle — nie jest to z mej strony żaden zarzut, tak właśnie powinno się podchodzić do projektowania. Jednak pominięcie czynności mających na celu identyfikację potencjalnych zagrożeń występujących podczas pracy systemu czyni ten proces niekompletnym. Każdy dobrze skonstruowany projekt powinien zawierać analizę ryzyka.

W niniejszym rozdziale mam nadzieję pokazać Ci, w jaki sposób składniki dobrze zorganizowanego projektu aplikacji uzupełnić można o element odpowiadający za poprawną obsługę wyjątków, zaplanowany po analizie potencjalnych problemów. Nie wiąże się to z jakimś ogromnym, dodatkowym nakładem pracy, a w zamian otrzymujemy znacznie bezpieczniejszy, i dzięki temu łatwiejszy w utrzymaniu, kod.

Zasady dobrego projektowania obiektowego

Często słyszy się, że rozpoczynając dane przedsięwzięcie dobrze jest mieć wizję jego zakończenia. Zastosujmy tę regułę w dziedzinie projektowania aplikacji obiektowych: jakie są główne cele i reguły ich tworzenia? Czym charakteryzuje się dobrze zaprojektowany i napisany, obiektowy kod? Najbardziej podstawowym kryterium, które kod musi spełnić, jest zapewnienie założonej funkcjonalności przy dającej się zaakceptować wydajności. Powinien on też być przejrzysty i elastyczny, dając dzięki temu możliwości wykorzystania go w innych projektach. O idealnej sytuacji można mówić, gdy pisząc aplikację, stworzymy zbiór uniwersalnych modułów, które tworzyć będą funkcjonalne i gotowe do ponownego użycia komponenty o jasno zdefiniowanych funkcjach¹.

Brzmi wspaniale. Założenia te wydają się rozsądne i nawet możliwe do realizacji. Szczególnie na etapie ich formułowania. Łatwo się o nich mówi, znacznie trudniej wprowadzić je w życie. Nie zważajmy jednak na trudności, zasad tych powinniśmy się trzymać podczas tworzenia każdej klasy, komponentu, systemu czy biblioteki². Istnieje kilka ogólnych reguł, które charakteryzują dobrze zaprojektowaną aplikację obiektową. Nasza dyskusja będzie łatwiejsza, gdy podczas ich prezentacji założymy, że odnoszą się one do pojedynczej klasy. Pamiętaj jednak, że stosujemy je także przy konstruowaniu bardziej złożonych komponentów. A zatem: projektując klasę, nie wolno nam zapomnieć o następujących zasadach, których istotę stanowi:

Abstrakcja: klasa jest modelem określonego pojęcia lub danych. Zasadę tę przenosimy na każdy obiekt danej klasy, dotyczy ona jego stanu i funkcjonowania.

- Pole: każde z pól reprezentuje pewien wewnętrzny stan obiektu, spełniając w ten sposób zadania stawiane przed modelem odpowiadającym pewnemu pojęciu i jego cechom.

¹ Dodatkowo kod powinien być kompletnie udokumentowany oraz dostarczony na czas i w ramach założonego budżetu. Co więcej, jasne sprecyzowanie wymagań użytkownika nie powinno zachwiać naszej równowagi psychicznej.

² Posługując się podczas dyskusji pojęciem komponent, mam na myśli zbiór klas zaprojektowany w taki sposób, by mogły one współpracować ze sobą i tworzyć moduł. Dobrze zaprojektowany komponent, system czy biblioteka posiadać musi te same cechy co dobrze skonstruowana klasa.

- **Metoda:** każda metoda reprezentuje operację, którą można na danym obiekcie przeprowadzić. Prowadzić ona może do zmiany stanu obiektu bądź też wykonywać zadanie związane z jego cyklem życia.

Enkapsulacja (hermetyzacja): pola i metody, które wspólnie tworzą klasę, podzielić możemy na dwie podstawowe kategorie.

- **Implementacja:** w jej skład wchodzi pola zawarte w klasie i metody zarządzające tymi danymi, mające wpływ na stan obiektu. Implementacja odpowiada za prywatną, niedostępną z zewnątrz część obiektu.
- **Interfejs:** metody stanowiące usługi udostępnione na zewnątrz, do których dostęp mają inne obiekty. Interfejs reprezentuje sposób, w jaki obiekt widziany jest przez pozostałą część aplikacji. Niektóre modele dokonują dalej idącego podziału, wyróżniając interfejs wewnętrzny, odpowiedzialny za sterowanie oraz zewnętrzny, związany z prezentacją.

Spójność: klasa ma jasno sprecyzowany cel i charakter. Odpowiada pojedynczemu pojęciu, którego dotyczą wszystkie zdefiniowane w niej operacje.

Niezależność: klasa powinna w jak najmniejszym stopniu zależeć od innych klas. Jeśli takich związków łączących klasy jest wiele, może pojawić się wątpliwość co do jakości projektu³.

Konsekwentne stosowanie powyższych reguł doprowadzić powinno do powstania dobrze zdefiniowanych klas, z których powstaną następnie komponenty, podsystemy, a na końcu kompletne aplikacje.

Patrząc z bardziej odległej perspektywy, aplikacja obiektowa postrzegana może być jako zbiór współpracujących za sobą obiektów, którym przyświeca pewien wspólny cel. A współpraca ta to komunikacja realizowana za pomocą wywoływania metod, co czynione jest według odpowiednich zasad i w określonej kolejności. Połączenie tej koncepcji z praktyką dostarczania przez klasy metod o jasno sprecyzowanych funkcjach w konsekwencji daje nam posiadające określoną hierarchię łańcuchy wywołań. Metody niższego poziomu wywoływane są przez metody wyższego poziomu, które to z kolei są używane przez jeszcze inną warstwę aplikacji itd. Struktura ta pozwala łączyć pojedyncze, w miarę proste operacje w czynności bardziej złożone, co ostatecznie doprowadzi nas do kompletnych procesów biznesowych i przypadków użycia, stanowiących podstawowe zadania aplikacji⁴.

Żadna z przedstawionych dotychczas zasad nie dotyczy jednak bezpośrednio zagadnienia obsługi wyjątków. Jaki jest związek tego tematu z dziedziną projektowania aplikacji?

³ Próba zdefiniowania pożądanych cech, które spełniać powinien dobrze zaprojektowany system obiektowy spowodowała powstanie zbioru zasad. W rezultacie temat ten jest wyczerpująco opisany, a duża część tej dokumentacji znajduje się w sieci. Polecam następujące adresy: <http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign> (Wiki Web) oraz <http://ootips.org/ood-principles.html> (OOTips).

⁴ W programowaniu strukturalnym proces dzielenia złożonej funkcjonalności na mniejsze operacje nazywany jest dekompozycją funkcjonalną. Podobnie jest w programowaniu obiektowym, ale tutaj odpowiedzialność za poszczególne operacje przypisywana jest klasom, co prowadzi do powstania modułów bardziej elastycznych i gotowych do ponownego użycia.

W realnych warunkach są z tym niemałe problemy. Wiele zespołów programistycznych (a także, niestety, metodyk przez nie stosowanych) do obsługi błędów podchodzi w sposób nie dość poważny. O konieczności obsługi błędów myślimy dopiero w momencie, gdy zaczynają nam one rzeczywiście przeszkadzać, a dzieje się tak podczas końcowych etapów realizacji przedsięwzięcia. Struktura systemu jest już wtedy ustabilizowana i niełatwo wprowadzić do niej zmiany lub rozszerzenia. Większość członków zespołu odczuwa w tym okresie niemałe zmęczenie, które potęgowane jest przez wyjątkowo szybko mnożące się problemy związane z pojawiającymi się w wielu miejscach aplikacji błędami. Jak w takiej sytuacji należy się zachować? Mogę zaproponować trzy możliwe rozwiązania polegające na próbie dołączenia obsługi błędów do aplikacji w końcowej fazie jej powstawania:

- 1.** Przechwytywanie i lokalna obsługa wyjątków w tych metodach, w których się one pojawiają. Rozwiązanie to nie wpływa na istniejącą już, dotychczasową strukturę aplikacji. Często jednak pociąga za sobą pewien nieład, brak konsekwencji i kojarzy się z cichą obsługą wyjątków, którą porównałem do zamiatania śmieci pod dywan. Inne części aplikacji nie zostaną przez to poinformowane o poważnych zagrożeniach, których zasięg nie musi mieć zawsze jedynie lokalnego charakteru.
- 2.** Zmiana projektu dokonana w ograniczonym zakresie, uwzględniająca modyfikacje konieczne do tego, by zarządzanie problemami przenieść na poziom komponentu. To rodzaj kompromisu, który zwykle nie burzy nam dotychczasowej, ogólnej struktury aplikacji. Podejście to jednak ma dwie wady. Po pierwsze, istnieje niebezpieczeństwo, że zbyt wiele modyfikacji na poziomie komponentu może odbić się niekorzystnie na jego jakości. Po drugie, ciągle istnieje ryzyko, że wyjątki obsługiwane na poziomie pojedynczego komponentu nie poinformują o ważnym zdarzeniu pozostałych elementów całej aplikacji.
- 3.** Reorganizacja całej aplikacji. Decyzja taka pociąga za sobą tak wiele koniecznych zmian, że tak naprawdę cofamy się do jednego z pierwszych etapów powstawania aplikacji. Co więcej, istnieje ryzyko, że nigdy fazy tej nie ukończymy, jeśli zmiana projektu uwzględniać ma pojawiające się na bieżąco błędy.

Nietrudno dojść do wniosku, że żaden z przedstawionych powyżej wariantów nie jest zachęcający. Fakt ten stanowi najlepszy dowód na to, że zagadnień związanych z obsługą błędów nie powinno się ignorować i czekać z tym do ostatniej fazy cyklu projektowego. W takim języku jak Java, błędy wiążą się z obsługującą je na poziomie aplikacji infrastrukturą, która swą konstrukcją nie odbiega od innych klas reprezentujących np. procesy biznesowe. Model zarządzający przesyłaniem komunikatów o błędach może więc być dobrze zaprojektowany, oferując tym samym skuteczny sposób ich raportowania i obsługi. Równie dobrze może też być bezładną, losową i zupełnie nieprzejrzystą zbieraniną bloków `try-catch`, a to na pewno nie pomoże nam w procesie testowania, debugowania i późniejszego zarządzania kodem.

W poprawnym planowaniu obsługi wyjątków pomoże nam wiedza wynikająca z dotychczasowej praktyki, zebrana jak zwykle w zbiór pewnych zasad. Tych reguł jest w sumie dziewięć. Mam nadzieję, że pozwolą co najmniej na uniknięcie podstawowych błędów popełnianych na etapie projektowania aplikacji:

- 1.** Każda sekwencja stanowiąca pewien sposób użycia aplikacji (przypadek użycia, ang. *use case*) powinna być przeanalizowana i opisana także z punktu widzenia potencjalnych błędów, jakie mogą się tam pojawić. Scenariusze błędów mogą tworzyć hierarchię, w której uwzględnimy zarówno prawdopodobieństwo danego zdarzenia, jak też jego konsekwencje.

2. Skuteczne projektowanie metod powinno zakładać, że oprócz podstawowych operacji przez nie realizowanych pojawi się dobrze zdefiniowany zbiór punktów wskazujących na potencjalne problemy. Gdy z metod tych budować będziemy proces lub przypadek użycia, listę potencjalnych błędów tej złożonej operacji określimy na podstawie wchodzących w jej skład metod.
3. Istnieją dwa podstawowe podejścia określające sposób, w jaki zajmować się będziemy potencjalnymi błędami. Pierwszy z nich zakłada napisanie takiego kodu, który do sytuacji nie dopuści. Drugi to odpowiednio zaplanowana obsługa błędu. Może to być zdefiniowany lokalnie zbiór czynności neutralizujących problem lub przesłanie informacji o zdarzeniu na zewnątrz metody, do zawierającego ją komponentu, podsystemu bądź całej aplikacji.
4. Nie należy próbować za wszelką cenę obsługiwać wszystkich możliwych błędów, zwiększy to bowiem znacznie czas potrzebny na napisanie kodu i zmniejszy możliwość jego późniejszym zarządzaniem. Dużo lepiej jest skoncentrować się na scenariuszach prawdopodobnych i trudnych w wielu przypadkach do uniknięcia, a także błędach mogących mieć poważny wpływ na działanie całej aplikacji⁵.
5. Metoda powinna przekazywać na zewnątrz jedynie te wyjątki, które zasługują na szczególną uwagę innych elementów aplikacji bądź też takie, których obsługa wewnątrz metody ze względu na brak znajomości kontekstu zdarzenia, nie jest wskazana.
6. Metoda powinna zgłaszać wyjątki w formie możliwie zrozumiałej dla ich konsumenta⁶, biorąc pod uwagę rolę i zadania odbiorcy.
7. Metoda może zgłaszać wiele wyjątków tylko w sytuacjach, gdy:
 - a. wzajemnie się one wykluczają (posiadają różne przyczyny),
 - b. mają inne znaczenie dla odbiorców,
 - c. nie dadzą się połączyć w pojedynczy wyjątek, ponieważ odpowiadają zupełnie różnym scenariuszom błędów.
8. Problemy powinny być obsługiwane lub raportowane jak najszybciej od momentu pojawienia się ich w kodzie. Najlepiej, gdy będzie to w tej samej metodzie, w której zostały zgłoszone. Zwlekanie z ich obsługą może być ryzykowne. W przypadkach, gdy takie odroczenie wydaje się niezbędne z powodu określonych wymogów stawianych przez klasę bądź bibliotekę, należy stworzyć metodę, która pozwoli nam uzyskać dostęp do oczekującego wyjątku.
9. Wyjątki dotyczące każdej jednostki stanowiącej część projektu (klasy, komponentu, modułu czy biblioteki) powinny w sposób jasny i formalny określać warunki, na jakich odbywa się ich współpraca ze światem zewnętrznym.

⁵ Unikajmy natomiast błędów, których naprawę da się uniknąć. Pozostałe problemy postarajmy się zapisać do dziennika.

⁶ Pod pojęciem „konsument” kryje się każdy odbiorca komunikatu, jakim jest zdarzenie. Może być to zarówno kolejna warstwa aplikacji, jak też użytkownik czytający komunikat wyświetlany na ekranie. Dlatego też nadanie mu zrozumiałej formy jest tak samo ważne w przypadku człowieka, jak innego podsystemu aplikacji.

Oczywiście klasy nie są jedyną strukturą w ramach pisanych przez nas aplikacji, na poziomie której należy rozważać i planować obsługę wyjątków. Java pozwala nam także korzystać z interfejsów i opisywać za ich pomocą pewne ogólne cechy i zachowania⁷. Zalecenia dotyczące definiowania wyjątków w ramach interfejsów są nieskomplikowane, chociaż wymagają jeszcze większych zdolności przewidywania niż te, jakimi musimy wykazać się tworząc zwykłe metody. Jeśli interfejs ma uczestniczyć w obsłudze wyjątków, należy dość starannie rozważyć jego użycie w ramach budowanego zestawu klas. Bazując na wymaganiach tworzonej konstrukcji i prawdopodobnych sposobach implementacji interfejsu, jego twórca może przewidzieć i zadeklarować pewne wyjątki zgłaszane przez wchodzące w skład interfejsu metody. Poprawne dodawanie deklaracji wyjątków w ramach interfejsów powinno być zgodne z dwiema następującymi regułami:

- 1.** Dobrze zaplanowany interfejs zwykle zgłasza tylko jeden rodzaj wyjątku, chyba że istnieje jakaś wyjątkowa przyczyna uzasadniająca większą ich liczbę. Typowy interfejs reprezentuje określony model bądź ściśle zdefiniowany zestaw zachowań. Dlatego dobrze jest przypisać mu jeden rodzaj wyjątku.
- 2.** Wyjątki zgłaszane przez metody interfejsu powinny bezpośrednio dotyczyć zadań, do jakich interfejs został przewidziany — ich znaczenie musi odpowiadać kontekstowi sytuacji, w której mogą się potencjalnie pojawić. Dlatego też definiowanie interfejsów idzie często w parze z tworzeniem własnych wyjątków.

Naturalną cechą interfejsów jest to, że nie definiują one dokładnego zachowania. Ich projektant nie posiada kontroli nad tym, jak dokładnie zostaną one w przyszłości zaimplementowane oraz w jaki sposób użyte zostaną zadeklarowane w metodach interfejsu wyjątki. Osoby wykorzystujące w swych aplikacjach biblioteki, w których implementowane są te interfejsy, są w lepszej sytuacji, mają bowiem wpływ i na generowanie wyjątków, i na ich obsługę. Trudno natomiast zazdrościć tym programistom, którzy nie mają wpływu ani na sam interfejs, ani na jego implementację⁸.

Twórcy interfejsu naprawdę trudno dobrać odpowiedni poziom ogólności wyjątków generowanych przez metody interfejsu, dlatego też błędy mogą być reprezentowane w sposób bardziej elastyczny. Trafiając na problem tego typu, możemy skorzystać z jednego z takich rozwiązań:

- 1.** Tworzymy interfejs ze stałą listą specyficznych dla niego wyjątków i wymuszamy niejako na osobach odpowiedzialnych za implementację wyrzucanie tylko tych wyjątków (bądź ich podzbioru), które zostały przez nas przewidziane. Jest to zwykle najlepsza opcja, gdy zakładamy narzucenie pewnego standardu posługiwania się interfejsem. Dotyczy to zwykle wyjątków związanych z błędami o średnim oraz wysokim priorytecie.
- 2.** Definiujemy interfejs, który nie deklaruje żadnych wyjątków. Ich obsługa musi więc zostać zawarta wewnątrz implementacji metod. Jest to opcja polecana szczególnie w sytuacjach, gdy zadania interfejsu sprecyzowane są w sposób bardzo ogólny i nie

⁷ Interfejs jest czasami porównywany w innych językach programowania do koncepcji „kontraktu”. Interfejsy zwiększają możliwości Javy, pozwalają na reprezentowanie pewnych współdzielonych pojęć i zachowań bez uciekania się do mechanizmu dziedziczenia.

⁸ Dobrym przykładem problemów tego typu jest biblioteka dostępu do baz danych, czyli JDBC. Twórcy interfejsu nie mieli żadnej kontroli ani nad ich implementacją (tworzone niezależnie sterowniki JDBC), ani nad kodem aplikacji (pisanym przez programistów używających JDBC).

sposób na tym etapie przewidzieć dalszych szczegółów. Dobrym przykładem jest interfejs `Runnable`, który mówi wyłącznie o tym, że jakieś zadanie będzie uruchomione w odrębnym wątku.

Miejmy także na uwadze, że implementując interfejs, zawsze możemy wyrzucić wyjątek wywodzący się z klasy `RuntimeException` lub `Error` (może się to zdarzyć w sposób zamierzony bądź przypadkowy). Wymienione wyjątki niekontrolowane stanowią pewien problem przy posługiwaniu się interfejsami. Nie wiemy bowiem, które z nich są celowo generowane przez implementację interfejsu, i kiedy możemy się ich spodziewać. Z tego też powodu dobrym pomysłem jest obsługa również takich wyjątków, na równi z wyjątkami jawnie zadeklarowanymi. Wskazane jest to szczególnie wtedy, gdy chcemy zagwarantować maksimum bezpieczeństwa podczas używania implementacji, których zachowania nie da się w pełni przewidzieć.

Projektowanie z uwzględnieniem błędów

Przestrzeganie zasad dotyczących skutecznej obsługi wyjątków już na etapie projektowania aplikacji pomoże stworzyć program dużo bardziej niezawodny. Oczywiście możliwe jest bezpośrednie włączenie tych zasad w normalny cykl rozwoju aplikacji. Jeśli na dodatek reguły dotyczące projektowania obiektowego połączymy z zaprezentowanymi wcześniej dobrymi praktykami obsługi wyjątków w pisanym przez nas kodzie, otrzymamy w rezultacie technikę, którą nazwać możemy „projektowaniem z uwzględnieniem błędów”⁹. Praktyka ta nie jest niczym rewolucyjnym — u jej podstaw leży solidne projektowanie obiektowe, przewidujące jednak i uwzględniające już na tym etapie potencjalne problemy, które mogą wystąpić podczas działania aplikacji. Dołączenie wyjątków już podczas projektowania kodu może naprawdę w znaczący sposób przyczynić się do lepszego zarządzania i łatwiejszego utrzymania aplikacji w przyszłości.

Powyższe reguły możemy stosować niezależnie od szczegółów preferowanej przez nas metodyki rozwoju oprogramowania. Punktem wyjścia jest analiza modelowanych przez nas procesów biznesowych. Model ten definiujemy obecnie najczęściej za pomocą przypadków użycia, ale równie dobrze możemy skorzystać z metod strukturalnych i dekompozycji funkcjonalnej. Schemat tego procesu, wraz z uwzględnieniem sytuacji wyjątkowych, przedstawiony jest na rysunku 6.1¹⁰.

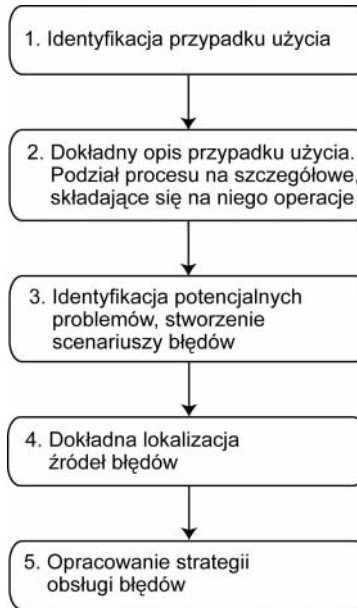
Oczywiście najlepszym sposobem, by poznać daną technikę i docenić jej walory jest demonstracja, w jaki sposób jej używać. Dlatego każdą fazę schematu zamierzam zilustrować przykładem. Scenariusz całego rozwiązania zakłada sytuację, w której klient chce za pomocą aplikacji sieciowej złożyć zamówienie — zadanie typowe dla wielu współczesnych aplikacji. Problem będzie bardziej interesujący, jeśli założymy, że zamówienie będzie się mogło składać z wielu pozycji, a każda z nich określa dowolną ilość danego towaru. Omówię dokładnie każdy z kroków, opisując czynności z nim związane.

⁹ Koncepcja „projektowania z uwzględnieniem błędów” w sposób naturalny zawiera w sobie wszystkie zasady projektowania obiektowego. Należy myśleć o niej jako o połączeniu dotychczas stosowanych praktyk i strategii zakładających uwzględnienie kwestii dotyczących przewidywania i obsługi potencjalnych błędów aplikacji.

¹⁰ Doświadczeni twórcy aplikacji często w instynktowny sposób wykonują taką właśnie sekwencję czynności, uwzględniając potencjalne błędy i planując strategię ich obsługi. Jednak do rzadkości należy oficjalnie i konsekwentnie stosowanie podobnych praktyk w zespołach programistycznych.

Rysunek 6.1.

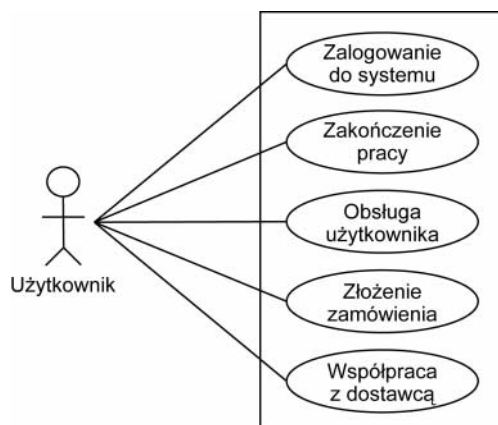
Proces tworzenia aplikacji uwzględniający obsługę wyjątków

**Etap 1. Identyfikacja przypadku użycia**

Na początku zidentyfikujemy przypadek użycia, który opisuje cały proces. Będą nim operacje, które rozpoczną działanie w chwili, gdy użytkownik zdecyduje się przesłać zamówienie, korzystając z odpowiedniego polecenia przeglądarki internetowej. Oczywiście, zakładamy w tym momencie, że użytkownik doprowadzając do takiego stanu, musiał wcześniej wykonać pewną liczbę czynności (przeglądanie i wybór zamawianych artykułów, określenie sposobu płatności itp.). W modelu UML operacje te traktowane są jako warunki wstępne przypadku użycia (często stanowiące inny przypadek użycia). Diagram przypadku użycia reprezentujący typowe wypełnienie zamówienia widzimy na rysunku 6.2.

Rysunek 6.2.

Diagram przypadków użycia związanych z systemem realizacji zamówień



Etap 2. Precyzyjny opis procesu i poszczególnych jego składników

Podczas tego etapu definiujemy poszczególne akcje związane z naszym przypadkiem użycia. Krok ten polega przede wszystkim na zrozumieniu, jakie dokładnie operacje należy wykonać i jakiej technologii do tego użyć. Oczywiście, im więcej znamy szczegółów, tym lepiej — posiadając informacje na temat każdej wykonywanej czynności, jesteśmy w stanie przewidzieć ewentualne błędy. Proces biznesowy zwykle powinien być opisany z uwzględnieniem trzech aspektów:

- zadań wykonywanych w ramach aplikacji,
- zadań wykonywanych pomiędzy aplikacjami,
- stosowanych technologii i bibliotek.

Wynik tej pracy warto zapisać za pomocą notacji UML: diagramy sekwencji i diagramy aktywności wydają się do tego najodpowiedniejsze.

Na początku wystarczy przypadek użycia o nazwie „złożenie zamówienia” podzielić na składające się na niego operacje, tak jak to widać poniżej:

Proces: **Złożenie zamówienia**¹¹.

Weryfikacja zamówienia:

1. Sprawdzenie, czy kompletne są wszystkie informacje stanowiące zamówienie.
2. Sprawdzenie danych dotyczących dostawy.
3. Sprawdzenie danych dotyczących płatności.

Realizacja zamówienia:

1. Tworzenie zamówienia i rezerwacja towaru.

Obliczenie ceny końcowej.

Sfinalizowanie zamówienia.

Patrząc na nasz model biznesowy z innej perspektywy, możemy wskazać te miejsca, które odpowiadają za współpracę z systemami zewnętrznymi:

- baza danych towarów (krok 4., rezerwacja towaru);
- baza danych zamówień (krok 4., akceptacja zamówienia);
- usługa obsługująca płatność (kroki 3. i 6., operacje finansowe);
- usługa związana z dostarczeniem towaru (krok 6., finalizowanie zamówienia).

¹¹W przykładzie zakładamy, że pewna wstępna weryfikacja zamówienia ma miejsce jeszcze przed jego złożeniem. Za tą „wczesną walidację” odpowiadać może któraś z wcześniejszych faz procesu, może się to odbywać jeszcze po stronie przeglądarki. My w każdym razie planujemy wykonanie jedynie „później walidacji”.

Wiedza na temat tych systemów pozwala zidentyfikować technologie i protokoły komunikacyjne, które najprawdopodobniej będą używane w aplikacji. To z kolei znowu przybliży nas do wskazania możliwych problemów w ramach realizacji przypadku użycia. W naszym przykładzie do obsługi baz danych wykorzystamy najprawdopodobniej bibliotekę JDBC. Wybór API, za pomocą którego będziemy się komunikowali z pozostałymi systemami zależy od sposobu, w jaki oferują one swoje usługi. W przypadku przesyłania komunikatów będzie to najprawdopodobniej JMS, usługi sieciowe to JAXM lub JAX-RPC.

Rysunek 6.3 prezentuje diagram aktywności zawierający sześć zdefiniowanych przed chwilą zadań. Możemy traktować go jako model procesu widziany na średnim poziomie szczegółowości.

Rysunek 6.3.
Diagram aktywności dla przypadku użycia „złożenie zamówienia”



Zauważ, że dwie pierwsze fazy procesu odpowiadają czynnościom, które każdy zespół projektowy musi przeprowadzić. I to niezależnie od tego, czy traktowane są one jako ściśle przestrzegane fazy zdefiniowane w którejś z metodyk, czy też jako intuicyjny i naturalny etap niezbędny podczas tworzenia działającego systemu. Kolejna, trzecia faza korzysta z wyników fazy pierwszej i drugiej. Pozwala nam ona zdobytą wiedzę wykorzystać w celu lepszego zidentyfikowania potencjalnych błędów, które mogą pojawić się w czasie eksploatacji aplikacji.

Etap 3. Identyfikacja potencjalnych błędów i ocena ich ryzyka

Gdy proces biznesowy zaczyna przybierać realny i stabilny kształt, możemy rozpocząć wyszukiwanie podstawowych błędów, które mogą zagrozić całej operacji. Jak już wspominałem, problemy dotyczące przypadku użycia w naturalny sposób związane będą z samym procesem, stosowanymi technologiami i współpracującymi systemami. Zrozumienie poszczególnych

zadań składających się na proces znacznie pomaga nam uświadomić sobie, jakie ryzyko niesie każde z nich. A znając to ryzyko, możemy przypisać mu priorytet i zastanowić się nad strategią obrony.

Znajomość określonych technologii lub API używanych podczas realizacji pozwala jeszcze dokładniej sprecyzować prawdopodobne błędy. Java jest pod tym względem doskonałym językiem, ponieważ miejsca zgłoszenia dużej części błędów definiowane są jawnie w nagłówkach wywoływanych metod. Znajomość używanego API wiąże się z precyzyjną identyfikacją potencjalnych problemów. Co więcej, umiemy przyporządkować je określonym metodom.

Wróćmy teraz do sześciu zadań opisanych już w punkcie 2., starając się tym razem przewidzieć i wyszukać potencjalne źródła błędów:

1. Sprawdzenie, czy kompletne są wszystkie informacje stanowiące zamówienie

Operacje: Wewnętrzna weryfikacja w ramach obiektów aplikacji

Czynniki ryzyka: Uszkodzenie danych systemu

Ocena ryzyka: Niskie

Operacja weryfikuje kompletność danych stanowiących zamówienie — sprawdzane są dane klienta, lista zamówionych towarów, sposób dostawy i szczegóły dotyczące płatności. Do wykonania tych czynności nie jest konieczne żadne złozone API i odwoływanie do zewnętrznych systemów, dlatego jest to operacja o niewielkim stopniu ryzyka. Podstawowe niebezpieczeństwo wiąże się ze zniszczeniem danych. Ten typ ryzyka wskazuje na potrzebę upewnienia się, że reguły biznesowe są poprawnie stosowane, co wymaga wykonania określonych testów w czasie cyklu projektowego.

2. Sprawdzenie danych dotyczących dostawy

Operacje: Wewnętrzna reguła biznesowa

Czynniki ryzyka: Uszkodzenie danych, nieprawidłowy przepływ sterowania

Ocena ryzyka: Niskie

Kolejna operacja wykonywana wewnętrznie. Choć reguła sprawdzająca poprawność może być bardziej złożona od poprzedniej (sprawdzenie sposobu dostawy i weryfikacja danych adresowych), czynność ta nie wiąże się z poważnym ryzykiem utraty integralności danych. Oczywiście pewność taką uzyskamy dopiero po wykonaniu odpowiedniej liczby szczegółowych testów.

3. Sprawdzenie danych dotyczących płatności

Operacje: Wewnętrzna reguła biznesowa, zewnętrzna weryfikacja

Czynniki ryzyka: Komunikacja w środowisku rozproszonym, uszkodzenie danych, nieprawidłowy przepływ sterowania

Ocena ryzyka: Średnie

Operacja obciążona jest większym ryzykiem niż dwie poprzednie. Weryfikacja informacji dotyczących płatności realizowana jest na zewnątrz systemu, dlatego istnieje prawdopodobieństwo awarii związanej z komunikacją pomiędzy systemami. Co więcej, błąd pojawiający się na tym etapie może później odbić się niekorzystnie na całym procesie zamawiania. Jednak ewentualny problem nie stanowi zagrożenia dla danych systemu i dotyczy wyłącznie pojedynczego zamówienia. Dlatego też ryzyko operacji oceniamy na średnie.

4. Tworzenie zamówienia, rezerwacja towaru

Operacje: Proces biznesowy, współpraca z bazą danych

Czynniki ryzyka: Komunikacja w środowisku rozproszonym, obsługa transakcji, integralność danych zamówienia i stanu towarów

Ocena ryzyka: Wysokie

To jedna z najbardziej newralgicznych operacji omawianego przypadku użycia. Korzysta z bazy danych i otwiera transakcję, na którą składać się może wiele wysłanych do niej poleceń. Co więcej, istnieje możliwość, że w tym samym czasie wielu użytkowników będzie chciało odwoływać się do tych samych danych, co może być przyczyną konfliktów. Problemy, które się pojawią na tym etapie mogą mieć poważne konsekwencje dla aplikacji klienta oraz systemu jako całości.

5. Obliczenie ceny końcowej

Operacje: Wewnętrzna kalkulacja

Czynniki ryzyka: Powstanie nieprawidłowych danych

Ocena ryzyka: Niskie

Na podstawie informacji zdobytych podczas realizacji poprzednich etapów generowana jest ostateczna cena zamówienia. Jest to czynność wykonywana wewnętrznie i obciążona niskim ryzykiem. Podobnie jak w poprzednich przypadkach, ważne jest przygotowanie właściwej strategii testowania — tym bardziej, że proces dotyczy operacji na kwotach.

6. Finalizacja zamówienia

Operacje: Proces biznesowy, uzgodnienia na styku dwóch systemów

Czynniki ryzyka: Komunikacja w środowisku rozproszonym, obsługa transakcji, zgodność pomiędzy towarem zamówionym i wysłanym

Ocena ryzyka: Wysokie

Ostatnia operacja przypadku użycia obciążona jest dużym ryzykiem. Wymagana jest dokładna koordynacja pomiędzy różnymi systemami, która gwarantuje nam, że przyjęte zamówienie będzie zrealizowane. Prawdopodobnie trzeba będzie wykonać rozproszoną transakcję, zapewniającą spójność między dwoma systemami. Zapewnienie integralności podczas realizacji tego etapu i minimalizacja ryzyka wydaje się być zadaniem niezwykle istotnym.

Jak widać, dokładne zdefiniowanie kolejnych etapów procesu pomogło nam w precyzyjnym wskazaniu natury i zakresu potencjalnych problemów, jakie mogą się podczas realizacji pojawić. Prawdopodobnym błędom warto nadać priorytet, w ten sposób zdamy sobie sprawę, które z nich są najgroźniejsze. W naszym przykładzie operacjami o największym stopniu ryzyka są te zadania, w których odwołujemy się do zewnętrznych systemów i baz danych.

W przykładzie skorzystaliśmy z dwóch technik służących do identyfikacji potencjalnych błędów. Możemy dokładnie wskazać obszary ryzyka podczas dzielenia całego przypadku użycia na składające się na niego, poszczególne operacje, a znajomość technologii, za pomocą których aplikacja będzie realizowana, pozwoli nam zakres ten dodatkowo ograniczyć. Stosując obie metody, uzyskamy wyraźny obraz problemów, które należy wziąć pod uwagę, a także stopień, w jakim nam zagrażają¹². Tabela 6.1 zawiera informacje będące podsumowaniem wymienionych metod.

Tabela 6.1. Działania przydatne podczas lokalizacji potencjalnych błędów

Technika/Obszar zastosowań	Wynik/Korzyść
Wyodrębnienie szczegółowych operacji w ramach przypadku użycia	Zlokalizowanie potencjalnego problemu
Wskazanie używanych technologii bądź API	Dostarczenie większej liczby szczegółów na temat ewentualnego błędu
Zdefiniowanie roli systemów zewnętrznych	Podkreślenie kluczowych punktów, w których zachodzi współpraca między systemami

W języku Java miejsca wystąpienia potencjalnych błędów definiowane są jawnie przez wyjątki, które dane API lub biblioteka może zgłosić. Znając API, z którego usług zamierzamy korzystać, posiadamy niemałą wiedzę na temat tego, co i dlaczego może się wydarzyć. Co więcej, pewne operacje są z natury obciążone wysokim ryzykiem. Dowolna forma komunikacji pomiędzy dwoma systemami wiąże się z dużym prawdopodobieństwem wystąpienia większej bądź mniejszej awarii. Podobnie jest w przypadku współdzielenia danych czy obsługi transakcji.

Scenariusze błędów i ryzyko ich wystąpienia opiszemy tym dokładniej, im lepiej zdefiniujemy operacje, które zachodzą w trakcie wykonywania procesu. Ogólne spojrzenie na proces biznesowy i potencjalne błędy z nim związane stanowią jedynie pewien zarys sytuacji, które mogą mieć miejsce. Wchodząc coraz głębiej w jego szczegóły będziemy w stanie dość dokładnie opisać te miejsca, które trzeba uwzględnić przy planowaniu obsługi błędów.

¹²Nie napisałem zbyt wiele na temat identyfikacji źródeł błędów oprócz tych, które wynikają ze znajomości określonego API. Istnieją sposoby wykrywania potencjalnych błędów w ramach określonego zadania bądź metody opierającej się na tworzeniu scenariuszy „co-jeśli”. Bierzemy wtedy pod uwagę parametry bądź warunki wstępne zadania.

Lista potencjalnych zagrożeń operacji „tworzenie zamówienia i rezerwacja towaru”.

1. Czynności związane z obsługą bazy danych towarów:

- zarządzanie połączeniem z bazą (metody `create` i `destroy`);
- operacje na bazie danych (`reserveItem`).

Uzyskanie połączenia z bazą za pomocą usługi katalogowej:

- przeszukiwanie puli dostępnych połączeń;
- obsługa połączenia (metody `getConnection` i `release`).

Konflikty związane z dostępem do zasobów w obiektach `OrderProcessingService` i `InventoryService`:

- unikanie problemów związanych z pracą wielowątkową (metody `createOrder` i `reserveItems`).

Kontrola transakcji w obiekcie `OrderProcessingService`:

- obsługa transakcji w metodzie `createOrder`.

Etap drugi (czyli precyzyjny opis składników procesu), trzeci (identyfikacja potencjalnych błędów) i prezentowany właśnie etap czwarty (wskazywanie miejsc wystąpienia tych błędów) wykonywane mogą być wielokrotnie, za każdym razem zbliżając nas do momentu, w którym cały proces będzie naprawdę dobrze przemyślany i zweryfikowany. Wtedy z czystym sumieniem przystąpić możemy do jego realizacji.

Etap 5. Przygotowanie strategii obsługi błędów

Faza finałowa procesu polega na zaplanowaniu strategii obsługi każdego ewentualnego błędu, który pojawi się podczas działania aplikacji. Możemy zastosować podejście ogólne — opisując po prostu sposób, w jaki powinniśmy zareagować. Z drugiej strony istnieje możliwość stworzenia bardzo szczegółowego scenariusza, który określi kod obsługi błędu oraz sposób zapisu informacji do dziennika. Obsługa wyjątków może mieć charakter jednopoziomowy, ale możemy zdecydować się też na przesyłanie wyjątków do innych warstw aplikacji. W naszym przykładzie scenariusze te moglibyśmy naszkicować mniej więcej tak:

Strategie obsługi błędów dla operacji „tworzenie zamówienia i rezerwacja towaru”

Klasa: `OrderProcessingService`

Metoda: `createOrder`

Ryzyko: Obsługa transakcji

Strategia: Przechwyтуjemy wyjątki zgłoszone wewnątrz metody. Gdy błąd jest poważny, wycofujemy transakcję, zapisujemy informację o błędzie do dziennika, przywracamy poprzedni stan aplikacji. W zależności od tego, jak implementacja jest złożona, można rozważyć stworzenie osobnej metody służącej do wycofania transakcji.

Ryzyko: Zakłócenie pracy wątku

Strategia: Unikajmy współdzielonych zmiennych odpowiadających za stan obiektu. Jeśli są one nieodzwonne, synchronizujemy dostęp do nich podczas tych operacji, które wiążą się ze zmianą stanu.

Klasa: InventoryService

Metoda: reserveItems

Ryzyko: Zakłócenie pracy wątku

Strategia: Unikajmy współdzielonych zmiennych odpowiadających za stan obiektu. Jeśli są one nieodzwonne, synchronizujemy dostęp do nich podczas tych operacji, które wiążą się ze zmianą stanu.

Klasa: InventoryDAO

Metoda: create

Ryzyko: Nieudana próba uzyskania połączenia z bazą danych

Strategia: Przekazujemy wyjątek na zewnątrz metody w celu przerwania całego procesu (wycofanie transakcji nie jest konieczne). Rozważyć warto zachowanie zamówienia i próbę przetwarzania w tle.

Metoda: reserveItem

Ryzyko: Błąd zgłoszony przez bazę danych podczas rezerwacji towarów

Strategia: Przekazujemy wyjątek na zewnątrz metody w celu przerwania całego procesu (tym razem należy wycofać transakcję). Rozważyć warto zachowanie zamówienia i próbę przetwarzania w tle.

Metoda: destroy

Ryzyko: Problemy związane z zakończeniem połączenia z bazą danych

Strategia: Notujemy błąd w dzienniku, usuwamy nieprawidłowo zakończone połączenie z bazą. Jeśli jest to możliwe, informujemy pulę połączeń z bazą o konieczności sprawdzenia tego, z którym mieliśmy problemy.

Klasa: NamingDelegate

Metoda: getConnection

Ryzyko: Problem ze znalezieniem lub dostępem do puli połączeń z bazą danych

Strategia: Próbuje znaleźć pulę połączeń ponownie. Gdy jest ona nadal niedostępna, przekazujemy wyjątek na zewnątrz metody w celu przerwania całego procesu. Nie jest wymagane wycofanie żadnej transakcji. Możemy także wysłać komunikat do administratora z prośbą o interwencję lub zgłosić czasową niedostępność systemu.

Metoda: release

Ryzyko: Problemy ze zwolnieniem połączenia

Strategia: Gdy pula połączeń jest niedostępna, wysyłamy komunikat do administratora z prośbą o interwencję lub zgłaszamy czasową niedostępność systemu.

Zauważ, że dyskusja na temat potencjalnych wyjątków mogących wystąpić w omawianym przykładzie nie zajęła nam aż tak wiele czasu. Jest to jedna z mocnych stron prezentowanego podejścia. Charakteryzuje się też ono dużą elastycznością w sposobie identyfikacji, a następnie uszczegółowienia zagrożeń związanych z działaniem aplikacji. Proces ten zastosować możemy bazując na istniejącym już planie aplikacji lub też dopasować jej strukturę do zidentyfikowanych wcześniej problemów. Pojęcie błędu aplikacji może wykraczać też poza jeden konkretny wyjątek. Potencjalne zagrożenie może być związane z całą serią wyjątków — nie jest to wcale rzadki przypadek.

Podstawową zaletą „projektowania z myślą o łatwiejszym zarządzaniu” są znacznie większe możliwości planowania obsługi wyjątków, uwzględniając również sytuacje, w których przekazywane są one między warstwami aplikacji. Gdy poświęcisz nieco czasu na dokładną analizę problemów, które mogą zaistnieć, Twoja strategia ich obsługi na pewno będzie lepiej przemyślana i przez to skuteczniejsza. Postawa taka jest na pewno o wiele dojrzsza niż traktowanie obsługi błędów jako niewiele znaczącego dodatku podczas pisania kodu.

Jeśli tylko chcesz, możesz podejść do zagadnienia z jeszcze większą precyzją. Przypomnij sobie listę opcji obsługi wyjątków omawianą w rozdziale 2. W każdym przypadku warto bowiem rozważyć, jaki sposób reakcji jest w danym wypadku najlepszy.

W sytuacji, gdy rozważamy obsługę wyjątków w systemie o ustabilizowanej już strukturze, zastanawiając się nad strategią wyjątków nie musimy koncentrować się już na pojedynczych metodach. Mamy warunki, by spojrzeć na to zagadnienie z perspektywy całej klasy lub nawet komponentu. Co więcej, wynik takiego planowania może zostać włączony na poziom całej aplikacji, dzięki czemu powstanie system, który na wyjątki reaguje zarówno na poziomie lokalnym, jak też globalnym.

Dalsze rozwijanie i dopracowywanie projektu powinno wiązać się z ponowną analizą i ulepszaniem strategii obsługi błędów. Warto więc ten etap włączyć na stałe do procesu tworzenia aplikacji. Takie formalne podejście wydaje się pożądane i bezpieczne dla całego zespołu, jasno wskazując na konieczność wykonania pewnych czynności i określając, kto jest za to odpowiedzialny.

Projektowanie z myślą o łatwiejszym zarządzaniu — korzyści i wady

Do tego miejsca pisałem o tym, w jaki sposób wprowadzić w życie postulowane przeze mnie podejście do projektowania aplikacji. Jest to technika, która pozwala nam identyfikować i obsługiwać te miejsca, które mogą w naszych aplikacjach sprawiać kłopoty. W ten sposób lokalizujemy i oceniamy ryzyko, a następnie przygotowujemy całościowy plan na wypadek, gdyby nasz czarny scenariusz się spełnił. Najlepsze jest to, że dodatkowe etapy związane z kompleksową obsługą wątków wplatają się zgrabnie w standardowy proces tworzenia aplikacji, dlatego nie musimy wprowadzać radykalnych zmian i zmieniać dotychczasowych przyzwyczajeń.

„Projektowanie z myślą o łatwiejszym zarządzaniu” jest techniką elastyczną. Można stosować ją w dowolnym momencie po ustaleniu operacji składających się na proces biznesowy. Korzyści pojawią się zarówno, gdy analizę potencjalnych zagrożeń przeprowadzimy mając ogólny zarys czynności, jak też w momencie, gdy gotowy jest obiektowy model systemu.

Rozważania dotyczące konieczności obsługi problemów mogą stanowić podstawę wyznaczenia pewnych działań, które następnie zdefiniujemy w postaci metod. Tak więc granice pomiędzy kolejnymi metodami mogą zostać wyznaczone nie tylko na podstawie czynności, które mają one realizować, ale też potencjalnych błędów wymagających określonej obsługi.

Zaproponowane modyfikacje sposobu projektowania aplikacji są na tyle ogólne, że istnieje możliwość zajęcia się określonymi wyjątkami nawet w dosyć późnej fazie procesu tworzenia aplikacji. Wcześniej możemy potencjalne błędy traktować w kategorii ogólnych problemów. Przypisanie odpowiednich wyjątków odbyć się może później, gdy będziemy posiadali większą wiedzę np. na temat użytej technologii.

Czy w takim razie „projektowanie z myślą o łatwiejszym zarządzaniu” jest w stanie rozwiązać wszystkie problemy, które mogą pojawić się w kodzie naszych aplikacji? Na pewno nie. Co więcej, nie jest celem tego podejścia identyfikacja wszystkich możliwych błędów — chodzi jedynie o te, które mogą mieć duże znaczenie dla poprawnego działania całości. Technika została stworzona po to, by zajmować się kluczowymi obszarami ryzyka. Teoretycznie proces analizy systemu możemy powtarzać wielokrotnie i za każdym razem wyszukamy kolejny, potencjalny błąd, ale takie jego stosowanie prowadzi donikąd. Po raz kolejny okazuje się, że korzystając z jakichkolwiek metod i zaleceń, nie wolno nam zapominać o podstawowej zasadzie tworzenia oprogramowania, która każe polegać przede wszystkim na zdrowym rozsądku.

Zauważ, że opisywany proces nie jest metodyką i nie jest idealnym rozwiązaniem radzenia sobie z obsługą błędów, które sprawdzi się w każdej sytuacji. Dlatego korzyści ze stosowania tej techniki należy starannie rozważyć i używać jej tylko tam, gdzie naszym zdaniem przyniesie korzyści.

Identyfikacja błędów

„Projektowanie z uwzględnieniem błędów” wymaga od nas dobrej identyfikacji i zrozumienia problemów, które mogą wystąpić podczas działania aplikacji. Już na etapie projektu musimy wskazać prawdopodobne błędy, jeśli chcemy się nimi zająć. Nie pomoże nam w tym kompilator języka Java, który ostrzega o nieobsłużonych wyjątkach. Ta bardzo przydatna cecha jest bezużyteczna, gdy szkicujemy schemat aplikacji, ponieważ najczęściej nie ma wtedy jeszcze ani jednego wiersza kodu. Kompilator potraktujemy raczej jak narzędzie, które sprawdzi, czy nasz plan obsługi wyjątków zrealizowaliśmy prawidłowo.

Nawet wówczas, gdy nie zdecydujesz się na włączenie do procesu tworzenia aplikacji punktu związanego tylko i wyłącznie z obsługą wyjątków, zawsze dobrym pomysłem będzie próba znalezienia czyhających na nas pułapek. Czynność, która analizuje pod tym kątem dany komponent, bibliotekę czy API, nazywamy identyfikacją błędów. W ten sposób staramy się znaleźć słabe punkty określonych klas bądź komponentów i znaleźć rozwiązanie, które pozwoli w niebezpiecznych dla działania aplikacji momentach odpowiednio zareagować. W ten sposób identyfikacja błędów daje nam wyraźniejsze spojrzenie na potencjalne błędy i minimalizuje ryzyko, że zostaniemy przez nie boleśnie zaskoczeni.

Podsumowanie

W niniejszym rozdziale rozważałem podstawowe zasady projektowania aplikacji z uwzględnieniem potencjalnych problemów, które mogą pojawić się w czasie jej działania. Przedstawiłem technikę pozwalającą włączyć identyfikację i plan obsługi błędów do typowego procesu projektowania aplikacji. W kolejnych rozdziałach przystąpię do analizy błędów, które są typowe dla konkretnych obszarów języka Java i wykorzystywanych powszechnie bibliotek. Zadaniem będzie zrozumienie działania poszczególnych składników i poznanie ryzyka, jakie się z tym wiąże. W tabeli 6.2 znajdują się tematy, które opisane są w kolejnych rozdziałach.

Tabela 6.2. *Tematy rozdziałów części drugiej*

Rozdział	Tematy
7.	Język Java: typy podstawowe, tablice, klasy ogólnego przeznaczenia
8.	Popularne API: obsługa kolekcji, system wejścia-wyjścia, biblioteka New I/O
9.	Aplikacje rozproszone: RMI, JNDI, JDBC
10.	J2EE: serwlety, JSP, EJB

W każdym z tych rozdziałów opisuję zadania poszczególnych API wraz z ich słabymi punktami i typowymi problemami, które wiążą się z ich używaniem. Wskażę najważniejsze wyjątki oraz sugerowane sposoby ich obsługi. Lektura kolejnych rozdziałów to bogaty zbiór bardzo konkretnych porad, które pozwolą Ci pisać znacznie lepsze i bezpieczniejsze programy.