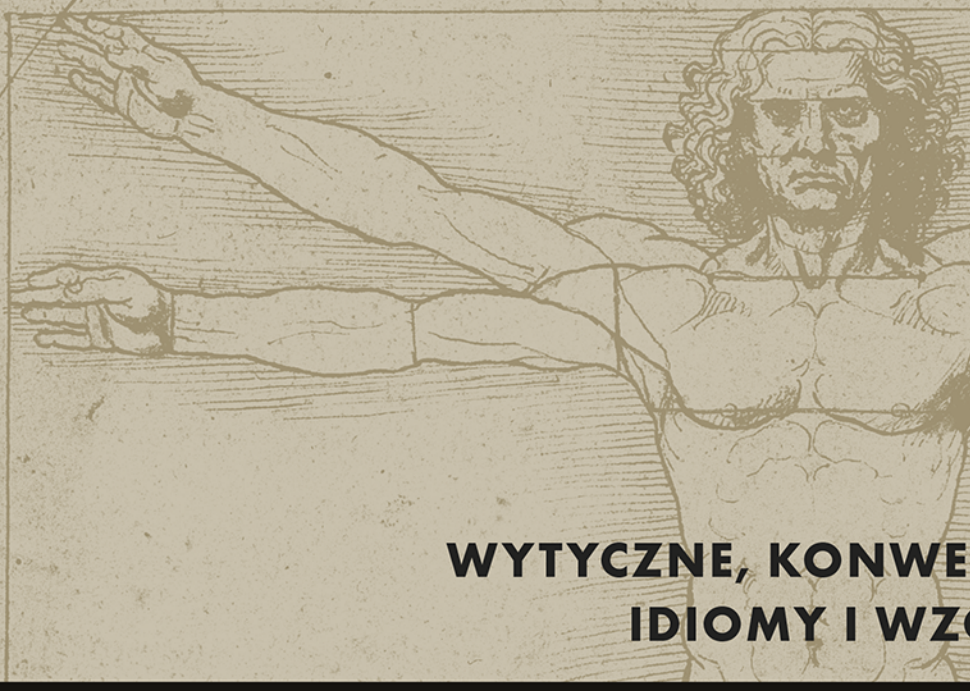
WYDANIE III



WYTYCZNE, KONWENCJE
IDIOMY I WZORCE

PROJEKTOWANIE FRAMEWORKÓW W .NET

KRZYSZTOF CWALINA
JEREMY BARTON
BRAD ABRAMS

Przedmowa SCOTT GUTHRIE
MIGUEL DE ICAZA, ANDERS HEJLSBERG

Helion

Tytuł oryginału: Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries (3rd Edition) (Addison-Wesley Microsoft Technology Series)

Tłumaczenie: Krzysztof Bąbol

ISBN: 978-83-283-7606-9

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/prfra3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/prfra3.zip

Printed in Poland.

Spis treści

2

Podstawy projektowania frameworków

Udany framework ogólnego przeznaczenia musi być projektowany dla szerokiego grona programistów o różnych wymaganiach, umiejętnościach i doświadczeniach. Jednym z największych wyzwań, przed którymi stoją projektanci frameworków, jest zaoferowanie tej zróżnicowanej grupie użytkowników odpowiednich możliwości przy zachowaniu prostoty.

Kolejnym ważnym celem projektanta frameworków musi być zapewnienie jednolitego modelu programowania, niezależnie od tego, jakiego rodzaju aplikację[1] pisze programista, a w przypadku wielojęzycznego środowiska wykonawczego — również od używanego przez niego języka programowania.

Dzięki korzystaniu z powszechnie akceptowanych ogólnych zasad projektowania oprogramowania i postępowaniu według zaleceń opisanych w tym rozdziale można stworzyć framework oferujący spójny zestaw funkcji odpowiednich dla szerokiego grona programistów, którzy budują różnego rodzaju aplikacje przy użyciu wielu języków programowania.

✓ **TAK** Projektuj frameworki, które mają duże możliwości i są zarazem łatwe w użyciu.

Dobrze zaprojektowany framework pozwala na łatwe zaimplementowanie prostych scenariuszy. Nie uniemożliwia przy tym implementacji tych bardziej zaawansowanych, choć może to być trudniejsze. Jak powiedział Alan Kay: „Proste rzeczy powinny być proste, a złożone — możliwe".

Wytyczna ta ma również związek z zasadą Pareta (zasadą 80/20), która mówi o tym, że w każdej sytuacji występuje 20 procent przypadków ważnych i 80 procent łatwych. Podczas projektowania frameworka trzeba koncentrować się na 20 procentach istotnych scenariuszy i interfejsów API. Inaczej mówiąc, należy przyłożyć się do zaprojektowania najczęściej używanych elementów frameworka.

[1] Komponent frameworka powinien na przykład, o ile to w ogóle możliwe, zachować ten sam model programowania niezależnie od tego, czy jest używany w aplikacji konsolowej, Windows Forms, czy ASP.NET.

✓ **TAK** Zapoznaj się z potrzebami szerokiego grona programistów o różnych stylach programowania, wymaganiach i poziomie umiejętności oraz projektuj framework specjalnie pod ich kątem.

> **PAUL VICK** Nie ma cudownego środka na zaprojektowanie frameworka dla programistów języka Visual Basic. Wśród naszych użytkowników są zarówno ludzie, którzy pierwszy raz stykają się z narzędziem programowania, jak i weterani pracy budujący komercyjne aplikacje na wielką skalę. Aby zaprojektować framework tak, by spodobał się programistom języka Visual Basic, należy przede wszystkim pozwolić im wykonać swoją pracę bez większego zamieszania i kłopotów. Dobrym pomysłem jest opracowanie frameworka z użyciem minimalnej liczby koncepcji, nie dlatego, że programiści VB nie są w stanie ich pojąć, ale dlatego, że konieczność zrobienia przerwy i pomyślenia o koncepcjach innych niż bieżące zadanie wybija ich z rytmu pracy. Celem programisty VB nie jest zwykle nauczenie się jakiejś interesującej lub ekscytującej nowej koncepcji ani zachwycanie się intelektualną czystością i prostotą projektu, ale wykonanie pracy i przejście dalej.

> **KRZYSZTOF CWALINA** Łatwo jest projektować dla użytkowników podobnych do siebie, a bardzo trudno dla kogoś, kto jest zupełnie inny. Istnieje wiele interfejsów API, które zaprojektowali eksperci dziedzinowi, a które, szczerze mówiąc, przydatne są tylko dla takich ekspertów. Problem leży w tym, że większość programistów nie jest, nigdy nie będzie i nie musi być ekspertami od wszystkich rozwiązań technicznych stosowanych w nowoczesnych aplikacjach.

> **BRAD ABRAMS** Choć słynne motto firmy Hewlett-Packard „Buduj dla inżyniera z sąsiedniego biurka" przyczynia się do jakości i kompletności projektów programistycznych, to w przypadku projektowania API prowadzi na manowce. Na przykład programiści z zespołu Microsoft Word dobrze zdają sobie sprawę z tego, że nie są docelowymi klientami tego programu. W dużo większym stopniu jest nim moja mama. Dlatego zespół programu Word umieszcza w nim znacznie więcej funkcji przydatnych dla mojej mamy niż takich, które byłyby pomocne dla samego zespołu. Choć w przypadku aplikacji takich jak Word jest to oczywiste, podczas projektowania interfejsów API często o tej zasadzie zapominamy. Skłonni jesteśmy projektować API tylko dla siebie, zamiast jasno pomyśleć o scenariuszach użycia przez klientów.

✓ **TAK** Poznaj wymagania różnorodnych języków programowania i projektuj framework pod ich kątem.

Istnieje wiele implementacji języków programowania obsługujących środowisko Common Language Runtime (CLR) i platformę .NET. Niektóre z nich mogą się bardzo różnić od języka używanego przy implementowaniu interfejsów API. Często należy zachować wyjątkową ostrożność, by zapewnić, że interfejsy API będą dobrze działały w różnych językach.

Na przykład programiści korzystający z dynamicznie typowanych języków z możliwością interakcji z platformą .NET (takich jak PowerShell, IronPython i innych) mogą mieć problem z użyciem interfejsów API wymagających utworzenia niestandardowego typu z atrybutami.

Innym przykładem jest język F#, w którym nie honoruje się zdefiniowanych przez użytkownika niejawnych operatorów konwersji. W rezultacie interfejsy API, do których w celu uproszczenia modelu wywoływania wprowadzono niejawne konwersje, niekoniecznie będą łatwe do użycia w tym języku.

JAN KOTAS Projektowanie z uwzględnieniem najmniejszego wspólnego mianownika istniejących języków programowania zaczęło przeszkadzać w ewolucji platformy .NET. W ostatnich latach zmniejszono nacisk na to, by wprowadzać wszędzie innowacje takie jak `Span<T>`. Do języków C# i F# wprowadzono nowe cechy, umożliwiające stosowanie `Span<T>`. Nie dokonano tego jednak w innych tradycyjnych językach platformy .NET (takich jak Visual Basic) i z poziomu tych języków nie można używać nowych interfejsów API opartych na typach `Span`.

JEREMY BARTON O ile prawdą jest, że wprowadziliśmy `Span<T>` bez wsparcia z poziomu języka VB, to w wytycznych dotyczących używania typów `Span` — zawartych w podrozdziale 9.12 — rekomendujemy wprowadzenie alternatywnych metod opartych na tablicach. Rekomendacja ta jest podyktowana częściowo względami użyteczności, ale ostatecznie powracamy w niej do tej wytycznej i różnorodności języków, które mogą korzystać ze środowiska CLR platformy .NET.

W książce zostaną też podane inne specjalne zalecenia dla różnych języków programowania.

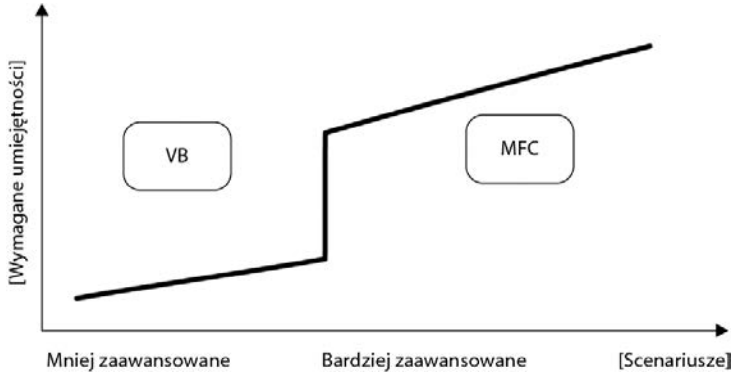
2.1. Progresywne frameworki

Projektowanie jednego frameworka pod kątem szerokiego grona programistów, różnych scenariuszy i języków jest trudnym i kosztownym przedsięwzięciem. Tradycyjnie dostawcy frameworków oferowali kilka produktów przeznaczonych dla specyficznych grup programistów i konkretnych scenariuszy. Firma Microsoft dostarczała na przykład interfejsy API języka Visual Basic, cechujące się prostotą i relatywnie wąskim zakresem scenariuszy, oraz interfejsy API Win32 API, o większych możliwościach i bardziej wszechstronne, choć nie tak łatwe w użyciu. Dla określonych grup programistów i scenariuszy przeznaczono też inne frameworki, takie jak MFC i ATL.

Chociaż to wieloframeworkowe podejście okazało się skutecznym sposobem dostarczania łatwych w użyciu interfejsów API o dużych możliwościach dla konkretnych grup programistów, ma ono też istotne wady. Główną wadą[2] jest to, że wielość frameworków utrudnia programistom korzystającym z jednego z nich przeniesienie swojej wiedzy na wyższy poziom

[2] Inne wady to dłuższy czas wejścia na rynek frameworków, które są osłonami dla innych frameworków, powielanie wysiłków i brak wspólnych narzędzi.

umiejętności lub do innego scenariusza (często wymaga to użycia innego frameworka). Jeśli na przykład programista musi zaimplementować nową aplikację, która wymaga bardziej zaawansowanego zestawu funkcji, czeka go bardzo intensywna nauka, bo będzie musiał nauczyć się zupełnie innego sposobu programowania, co pokazano na rysunku 2.1.



Rysunek 2.1. Krzywa uczenia się na platformie złożonej z wielu frameworków

ANDERS HEJLSBERG W dobrych starych czasach początków systemu Windows mieliśmy Windows API. W celu pisania aplikacji uruchamiano kompilator C, dołączano instrukcją `#include` nagłówek `windows.h`, tworzono funkcję `winproc` i pisano obsługę komunikatów okien — po prostu programowanie systemu Windows w starym stylu Petzolda. Co prawda to działało, ale nie było ani specjalnie produktywne, ani łatwe.

Z czasem powstały różne modele programowania oparte na Windows API. W języku VB przyjęto metodę błyskawicznego wytwarzania aplikacji (ang. *Rapid Application Development* — RAD). W środowisku VB można było utworzyć instancję formularza, przeciągnąć na nią komponenty i napisać procedury obsługi zdarzeń; kod był wykonywany dzięki mechanizmowi delegowania.

W świecie C++ mamy biblioteki MFC i ATL, w których przyjęto inny punkt widzenia. Kluczową koncepcją jest tutaj tworzenie podklas. Programiści tworzą je, korzystając z istniejącego monolitycznego, obiektowego frameworka. Choć daje to więcej możliwości i wyrazistości, nie zapewnia tak naprawdę produktywności opartego na kompozycji modelu środowiska VB.

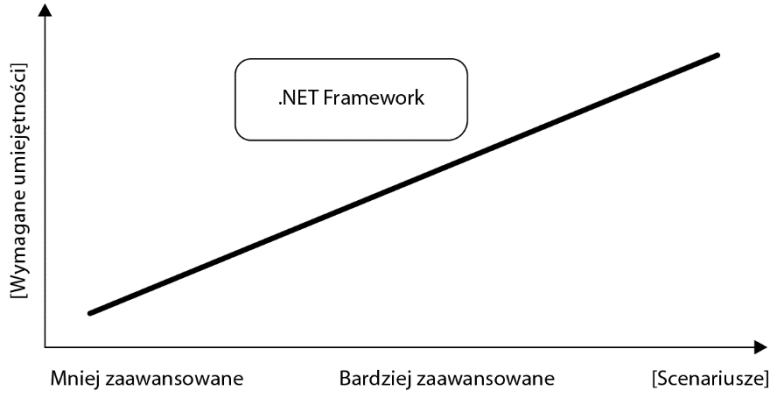
Jeśli spojrzeć na ten obraz, to widać, że jeden z problemów polega na tym, że wybór modelu programowania z konieczności pociąga za sobą wybór języka. To niepomyślna sytuacja. Jeśli wykwalifikowany programista MFC musi napisać trochę kodu w języku VB, jego umiejętności nie mają przełożenia. Podobnie jeśli ktoś ma sporą wiedzę na temat VB, to do programowania pod MFC niewiele mu się ona przyda.

Dostępne interfejsy API nie są też ze sobą spójne. W każdym z tych modeli wymyślono własne rozwiązania wielu problemów, tak naprawdę podstawowych i wspólnych dla wszystkich — na przykład tego, jak radzić sobie z plikowymi operacjami wejścia i wyjścia, jak formatować ciągi tekstowe, co z zabezpieczeniami, wątkami itd.

Te wszystkie modele unifikuje właśnie platforma .NET Framework. Zapewnia ona spójny interfejs API, który jest ogólnie dostępny bez względu na używany język czy przyjęty model programowania.

> **PAUL VICK** Warto też zauważyć, że ta unifikacja ma swoją cenę. Istnieje nierozwiązywalny konflikt: czy pisać frameworki dające programistom ogromne możliwości i znaczną kontrolę nad ich działaniem, czy tworzyć takie, które udostępniają bardziej ograniczony zestaw funkcji w bardzo uproszczony konceptualnie sposób. Zazwyczaj nie ma złotego środka i nieuchronnie trzeba szukać kompromisów między możliwościami a prostotą. Przy projektowaniu platformy .NET Framework dołożono ogromnych starań, by osiągnąć możliwie najlepszą równowagę pomiędzy oboma celami, ale myślę, że do dziś prowadzimy nad tym prace.

Dużo lepszym podejściem jest dostarczenie **progresywnego frameworka**, czyli jednego frameworka przeznaczonego dla szerokiego grona programistów, pozwalającego na przenoszenie wiedzy z mniej zaawansowanych do bardziej zaawansowanych scenariuszy. Taki właśnie progresywny charakter ma .NET Framework, a krzywa uczenia się jest w nim łagodna (rysunek 2.2).



Rysunek 2.2. Krzywa uczenia się na platformie progresywnego frameworka

Uzyskanie łagodnej krzywej uczenia się z niskim progiem wejściowym jest trudne, ale nie niemożliwe. Jest trudne, bo wymaga nowego podejścia do procesu projektowania frameworka, dużo większej dyscypliny i więcej kosztuje. Na szczęście wytyczne opisane w tym rozdziale i pozostałej części książki mają za zadanie przeprowadzić czytelnika przez ten trudny proces i pomóc mu zaprojektować świetny progresywny framework.

Należy też pamiętać o tym, że społeczność programistów jest ogromna: od pracowników biurowych rejestrujących makra aż po autorów niskopoziomowych sterowników urządzeń. Z frameworka, w którym próbowano by obsłużyć tych wszystkich użytkowników, wyszedłby mętlik, który nikogo by nie zadowolił. Progresywny framework powinien dać się dostosować do potrzeb szerokiego grona programistów, ale przecież nie każdego z nich. Widać więc jasno, że programiści, którzy nie mieszczą się w grupie docelowej, będą potrzebować specjalistycznych interfejsów API.

2.2. Podstawowe zasady projektowania frameworków

Dostarczenie zaawansowanej, a zarazem łatwej w użyciu platformy programistycznej jest jednym z głównych celów projektu .NET i powinno być też celem programistów, którzy ją rozszerzają. Pierwsza wersja platformy .NET Framework rzeczywiście zapewniała potężny zestaw interfejsów API, ale dla niektórych osób poszczególne jej elementy okazały się zbyt trudne.

RICO MARIANI Z drugiej strony interfejs API musi nie tylko być prosty w użyciu; łatwo powinno się też z niego korzystać w najlepszy możliwy sposób. Należy przemyśleć proponowane wzorce użycia i upewnić się, że najbardziej naturalny sposób korzystania z systemu przynosi poprawne rezultaty, cechuje się odpornością na ataki i bardzo dobrą wydajnością. Trzeba utrudniać robienie rzeczy w niewłaściwy sposób. Kilka lat temu pisałem:

Pułapka sukcesu
Nie proponujemy klientom zdobywania szczytów ani wędrówki przez pustynię, by metodą wielu prób i błędów dotarli do celu, ale chcemy, by dzięki użyciu naszej platformy i frameworków po prostu wpadli na najlepsze rozwiązania. Na ile jest im łatwo popaść w kłopoty, na tyle ponosimy porażkę.

Prawdziwa produktywność wynika z możliwości łatwego tworzenia świetnych produktów, a nie łatwego tworzenia śmieci. Zbuduj pułapkę sukcesu.

Uwagi użytkowników i badania nad użytecznością pokazały, że duży odsetek programistów VB miał problemy z nauczeniem się języka VB.NET. Problem wynikał częściowo z prostego faktu, że platforma .NET różni się od bibliotek VB 6.0, istniało jednak również kilka poważnych problemów z użytecznością, związanych z projektem interfejsów API. Naprawienie tych problemów stało się dla firmy Microsoft priorytetowe w czasie prac nad platformą .NET Framework 2.0.

Zasady opisane w tym podrozdziale, oznaczone etykietą „Zasada projektowania frameworków", zostały opracowane w celu uniknięcia powyższych problemów. Mają pomóc projektantom frameworków ustrzec się najpoważniejszych błędów opisywanych w wielu badaniach nad użytecznością i zgłaszanych przez użytkowników. Wierzymy, że zasady te mają kluczowe znaczenie w projektowaniu każdego frameworka ogólnego przeznaczenia. Niektóre z zasad i rekomendacji pokrywają się, co może dodatkowo świadczyć o ich słuszności.

2.2.1. Zasada projektowania opartego na scenariuszach

Frameworki zawierają często bardzo duży zestaw interfejsów API. To konieczne, by uwzględnić zaawansowane scenariusze wymagające możliwości i wyrazistości. Większość prac programistycznych obraca się jednak wokół małego zestawu najczęstszych scenariuszy, które są zależne od relatywnie małego podzbioru całego frameworka. W celu optymalizacji ogólnej produktywności programistów korzystających z frameworka bardzo ważne jest, by nie oszczędzać środków na projektowanie interfejsów API używanych w najczęściej spotykanych scenariuszach.

W związku z tym podczas projektowania frameworka należy skupić się na zbiorze najczęstszych scenariuszy, a co więcej, podporządkować im cały proces projektowania. Zalecamy, by twórcy frameworków najpierw napisali kod, który by w zasadniczych scenariuszach musieli pisać użytkownicy, a potem zaprojektowali model obiektów tak, by zapewnić obsługę tych przykładów kodu[3].

Zasada projektowania frameworków

Projektowanie frameworków musi się zaczynać od zbioru scenariuszy użycia i przykładowego kodu implementującego te scenariusze.

KRZYSZTOF CWALINA Do wyartykułowanej właśnie zasady dodałbym, że „nie ma po prostu innego sposobu na zaprojektowanie świetnego frameworka". Gdybym musiał wybrać do książki tylko jedną zasadę projektowania, byłaby to właśnie ta. Gdybym nie mógł napisać książki, ale tylko krótki artykuł o tym, co jest ważne w projektowaniu API, wybrałbym tę zasadę.

Projektanci frameworków często popełniają błąd polegający na zaprojektowaniu najpierw (przy użyciu różnych metodyk) modelu obiektów, a potem pisaniu przykładów kodu w oparciu o powstały interfejs API. Problem leży w tym, że większość metodyk projektowania (również tych najczęściej stosowanych — obiektowych) zoptymalizowano pod kątem łatwości utrzymania powstałej implementacji, a nie użyteczności tworzonych interfejsów API. Przydają się bardziej do projektowania wewnętrznej architektury, a nie warstwy publicznych interfejsów API obszernego frameworka.

Projektowanie frameworka należy rozpocząć od utworzenia specyfikacji API opartej na scenariuszach (patrz dodatek C). Może ona być oddzielona od specyfikacji funkcji albo stanowić część obszerniejszego dokumentu. W tym ostatnim przypadku tworzenie specyfikacji API powinno poprzedzać specyfikację funkcji zarówno pod względem czasu, jak i miejsca.

Specyfikacja powinna zawierać oddzielny ustęp zawierający od pięciu do dziesięciu głównych scenariuszy w danym obszarze technicznym, wraz z przykładami implementacji. Jeśli w API lub przykładach kodu użyto nowych lub nietypowych cech języka, warto napisać przykłady w co najmniej jeszcze jednym języku programowania, bo taki kod może czasem znacznie się różnić.

Ważne jest też, by zapisać te scenariusze różnymi stylami kodowania, typowymi dla użytkowników danego języka (ze specyficznymi dla niego cechami). W przykładach należy zastosować odpowiednią wielkość liter. W języku VB.NET nie jest ona rozróżniana, więc należy to uwzględnić w przykładach. W kodzie C# należy stosować standardowe reguły wielkości liter, opisane w rozdziale 3.

[3] Przypomina to procesy oparte na programowaniu sterowanym testami (ang. *test-driven development* — TDD) albo na przypadkach użycia. Są jednak pewne różnice. TDD to poważniejsza technika, a jej cele wykraczają poza wspomaganie projektowania interfejsów API. Z kolei w ramach przypadków użycia scenariusze opisywane są na poziomie wyższym niż jednostkowe wywołania API.

✓ **TAK** Dopilnuj, by specyfikacja projektowania interfejsów API była w centrum uwagi podczas projektowania każdej funkcji frameworka, która zawiera publicznie dostępne API.

Przykład takiej specyfikacji zawiera dodatek C.

✓ **TAK** Zdefiniuj najważniejsze scenariusze użycia w każdym głównym obszarze funkcji.

Specyfikacja API powinna zawierać ustęp z opisem głównych scenariuszy i przykładami ich implementacji. Powinno to nastąpić zaraz po wstępnym streszczeniu. Na przeciętny obszar funkcji (np. plikowe operacje wejścia i wyjścia) powinno przypadać od pięciu do dziesięciu głównych scenariuszy.

✓ **TAK** Upewnij się, że scenariusze mają odpowiedni poziom abstrakcji. Powinny mniej więcej odpowiadać przypadkom użycia przez programistów.

Dobrym scenariuszem jest na przykład odczyt z pliku. Otwarcie pliku, odczyt z niego wiersza tekstu lub zamknięcie pliku nie są dobrymi scenariuszami; są zbyt szczegółowe.

✓ **TAK** Przy projektowaniu API najpierw napisz przykłady kodu dla głównych scenariuszy, a potem zdefiniuj model obiektów tak, by obsługiwał te przykłady.

Przykładowo, podczas projektowania API do mierzenia upływu czasu możesz napisać przykłady kodu dla następujących scenariuszy:

```
// scenariusz #1: zmierz czas, który upłynął
Stopwatch watch = Stopwatch.StartNew();
DoSomething();
Console.WriteLine(watch.Elapsed);

// scenariusz #2: użyj ponownie stopera
Dim watch As Stopwatch = Stopwatch.StartNew()
DoSomething();
Console.WriteLine(watch.ElapsedMilliseconds)

watch.Reset()
watch.Start()
DoSomething()
Console.WriteLine(watch.Elapsed)
```

Na podstawie tych przykładów można utworzyć następujący model obiektów:

```
public class Stopwatch {
   public static Stopwatch StartNew();

   public void Start();
   public void Reset();

   public TimeSpan Elapsed { get; }
   public long ElapsedMilliseconds { get; }
   ...
}
```

JOE DUFFY My, programiści, lubimy tworzyć dla rozrywki nowe, zaawansowane funkcje i dzielić się nimi z innymi. Między innymi dlatego projektowanie API jest tak przyjemne. Niesłychanie trudno jest jednak cofnąć się o krok i obiektywnie ocenić, czy nowa funkcja, do której podchodzimy ze szczególnym entuzjazmem, będzie przydatna w praktyce. Korzystanie ze scenariuszy jest najlepszym znanym przeze mnie sposobem rozpoznania tego, czy nowa funkcja jest potrzebna i jaki powinien być wzór jej użycia. Opracowywanie scenariuszy jest w istocie niezwykle *trudne*, nie bez przyczyny: wymaga rzadko spotykanego połączenia umiejętności technicznych i zrozumienia potrzeb użytkownika. Koniec końców projektant może podjąć serię decyzji wyłącznie w oparciu o własne przeczucia i intuicję, a nawet wydać jakieś przydatne interfejsy API, ale dużo większe jest ryzyko, że będzie potem żałował dokonanego wyboru. W razie wątpliwości lepiej pominąć na razie nową funkcję i dodać ją później, gdy ta istotna potrzeba będzie bardziej rozpoznana.

STEPHEN TOUB Dodawanie nowych interfejsów API to frajda, ale każdy z nich powoduje koszty. Czasami jest to „tylko" koszt zaprojektowania, opracowania, przetestowania, udokumentowania i utrzymania danej funkcji (plus związane z nią koszty czasu wykonywania). Często się jednak niestety zdarza, że wprowadzenie nowego interfejsu API tak naprawdę blokuje na przyszłość możliwość dodania potencjalnie dużo bardziej pożądanej lub ważniejszej funkcji, która byłaby w konflikcie z zestawem funkcji tego interfejsu API. To jeden z powodów, dla których tak ostrożnie wybieramy nowo uwidaczniane interfejsy API, moglibyśmy bowiem w ten sposób zamknąć sobie możliwość innowacji na przyszłość. Jestem pewien, że wielu z nas mogłoby podać ciekawy przykład interfejsu API, którego wolałoby nigdy nie dodawać; ja mam takich kilka.

✓ **TAK** Przykłady dla głównego scenariusza napisz w co najmniej dwóch językach z różnych rodzin (np. C# i F#).

Najlepiej upewnić się, że wybrane języki mają znacząco różną składnię, styl i możliwości.

PAUL VICK Jeśli pisany przez Ciebie framework ma być używany w wielu językach programowania, naprawdę warto znać więcej niż jeden z nich (znajomość kliku języków z rodziny C się nie liczy). Zauważyliśmy, że czasami API działa poprawnie tylko w jednym języku, a to dlatego, że osoba projektująca dany interfejs API (lub go testująca) tak naprawdę znała tylko ten jeden język. Naucz się kilku języków platformy .NET i używaj ich zgodnie z przeznaczeniem. Oczekiwanie, że cały świat będzie mówił w Twoim języku, nie sprawdza się zbyt dobrze na wielojęzycznej platformie, jaką jest .NET Framework.

✓ **PRZEMYŚL** Pisz przykłady kodu dla głównego scenariusza w języku typowanym dynamicznie, takim jak PowerShell lub IronPython.

Nietrudno zaprojektować interfejsy API tak, że nie będą działały poprawnie w językach typowanych dynamicznie, w których często występują problemy z niektórymi metodami

uogólnionymi, oraz z interfejsami API, które wymagają nadawania atrybutów lub tworzenia typów o silnym typowaniu.

✗ **NIE** polegaj wyłącznie na standardowych metodykach projektowania podczas projektowania warstwy publicznych interfejsów API frameworka.

Standardowe metodyki projektowania (również obiektowe) zoptymalizowano pod kątem łatwości utrzymania powstałej implementacji, a nie użyteczności tworzonych interfejsów API. Dużo lepszym podejściem jest projektowanie oparte na scenariuszach wraz z tworzeniem prototypów, badaniami użyteczności i pewną dozą iteracji.

CHRIS ANDERSON Każdy programista ma własne metody pracy i choć zasadniczo nie ma nic złego w stosowaniu innych metod modelowania, to generalnie problemem są uzyskiwane wyniki. Stworzenie na początku kodu, jaki powinien pisać programista, jest prawie zawsze najlepszym podejściem — można to potraktować jako formę programowania sterowanego testami. Pisze się wzorowy kod, a potem rozpracowuje go, aby dojść do pożądanego modelu obiektów.

2.2.1.1. Badania użyteczności

Badania użyteczności prototypu frameworka prowadzone wśród szerokiego grona programistów są kluczem do projektowania opartego na scenariuszach. Interfejsy API dla głównych scenariuszy mogą się wydawać autorom proste, ale mogą wcale takie nie być dla innych programistów.

Zrozumienie sposobu, w jaki programiści podchodzą do każdego z głównych scenariuszy, pozwala głębiej spojrzeć na projekt frameworka i ocenić, jak dobrze zaspokaja on potrzeby wszystkich docelowych programistów. Z tego powodu prowadzenie badań użyteczności — formalnych czy nieformalnych — jest bardzo ważną częścią procesu projektowania frameworków.

Jeśli podczas badań projektant odkryje, że większość programistów nie potrafi zaimplementować jednego ze scenariuszy albo że ich podejście znacznie się różni od oczekiwanego, interfejs API powinien zostać przeprojektowany.

KRZYSZTOF CWALINA Przed wydaniem wersji 1.0 platformy .NET Framework nie przeprowadziliśmy testów użyteczności typów w przestrzeni nazw `System.IO`. Wkrótce potem napłynęły do nas negatywne opinie użytkowników. Byliśmy dość zaskoczeni i postanowiliśmy przeprowadzić badania użyteczności w grupie ośmiu przeciętnych programistów. Żaden z nich nie zdołał odczytać danych z pliku tekstowego w ciągu 30 minut przydzielonych na to zadanie. Uważamy, że było to częściowo spowodowane problemami z wyszukiwarką dokumentacji i niewystarczającą ilością przykładów; oczywiste jest jednak, że sam interfejs API też miał szereg problemów z użytecznością. Gdybyśmy przeprowadzili badania przed wydaniem produktu, moglibyśmy wyeliminować poważne źródło niezadowolenia klientów i uniknąć kosztów związanych z próbą naprawy interfejsu API w zasadniczym obszarze funkcji bez wprowadzania istotnych zmian.

BRAD ABRAMS Nie ma mocniejszego doświadczenia dającego projektantowi API dogłębne zrozumienie użyteczności swojego API niż siedzenie za lustrem weneckim i obserwowanie, jak programiści, jeden po drugim, są sfrustrowani zaprojektowanym przez niego API i ostatecznie żadnemu nie udaje się wykonać zadania. Osobiście odczuwałem cały wachlarz emocji, gdy obserwowałem badania użyteczności przestrzeni nazw `System.IO`, które przeprowadziliśmy zaraz po wydaniu wersji 1.0. Gdy żaden programista nie potrafił wykonać prostego zadania, moje emocje przechodziły od arogancji do niedowierzania, potem do frustracji, a w końcu pojawiła się determinacja, by naprawić problem w API.

CHRIS SELLS Przy odpowiedniej ilości czasu i pieniędzy badania użyteczności mogą być sformalizowane, jednak 80 procent potrzebnych informacji można zebrać po uruchomieniu proponowanego interfejsu API przez kilku programistów bliskich docelowej grupie użytkowników biblioteki. Niech pojęcie „badań użyteczności" nie wystraszy Cię do tego stopnia, by nic nie robić: pomyśl o nich jak o badaniach typu „hej, spójrz na to".

STEVEN CLARKE Zamiast wkładać mnóstwo wysiłku w planowanie, projektowanie i prowadzenie szerokich badań z wieloma uczestnikami i próbować uwzględnić w nich jak największą powierzchnię interfejsów API, odkryliśmy, że dużo bardziej wartościowe jest prowadzenie serii mniejszych, bardziej ukierunkowanych badań w trakcie procesu opracowywania API. W każdym studium prosimy niewielką grupę uczestników o skupienie się na jednym zagadnieniu projektowym albo obszarze API. Korzystamy z tego, czego się dowiedzieliśmy, wracamy do projektowania, a tydzień lub dwa później przeprowadzamy kolejne badania nad zaktualizowanym projektem lub innym obszarem API. Ta metoda ciągłego uczenia oznacza, że do procesu projektowania napływa ciągły strumień spostrzeżeń użytkowników, a nie jedna duża dawka wiedzy dostarczona w pewnym konkretnym momencie.

Idealnie byłoby, gdyby badania użyteczności API prowadzono z użyciem prawdziwego środowiska programistycznego, edytorów kodu i dokumentacji najczęściej używanej przez docelową grupę programistów. Badania użyteczności lepiej prowadzić jednak na wcześniejszym etapie prac nad produktem — nie warto więc odkładać organizacji badania tylko dlatego, że cały produkt nie jest jeszcze gotowy.

STEPHEN TOUB Aby uzyskać wyobrażenie o użyteczności API, niepotrzebna jest nawet realna implementacja. Choć dobrze, żeby programiści mieli coś, co mogą uruchomić, aby zobaczyć wyniki swoich eksperymentów. Jednak wczesną informację zwrotną na temat projektu mogą uzyskać, mając po prostu interfejs API, dla którego mogą napisać i skompilować kod, co oznacza, że w całej implementacji można użyć zaślepek niewykonujących żadnych operacji lub zgłaszających wyjątki; nie ma to znaczenia, bo i tak nie będą one wywoływane. Czy programiści intuicyjnie znajdowali właściwe typy? Czy potrafili rozpoznać wzorce dostępu do funkcji? Czy mechanizm IntelliSense potrafił w sensowny sposób oprowadzić ich po interfejsie API? Czy podchodzili do problemów w zakładany hipotetycznie sposób? Czy regularnie szukali rzeczy nazywających się jakoś inaczej?

W przypadku małych zespołów programistycznych i frameworków przeznaczonych dla względnie małego grona programistów formalne badania użyteczności są często niepraktyczne. Mogą one wtedy przyjąć nieformalny charakter. Pokazanie prototypu biblioteki komuś nieobeznanemu z projektem, poproszenie go o napisanie w ciągu 30 minut prostego programu i obserwowanie, jak sobie z tym radzi, jest doskonałym sposobem na wykrycie najbardziej kłopotliwych problemów z projektem API.

✓ **TAK** Organizuj badania użyteczności, by sprawdzać interfejsy API w głównych scenariuszach.

Badania te powinny być organizowane na wczesnym etapie cyklu opracowywania API, ponieważ najpoważniejsze problemy z użytecznością często wymagają znacznych zmian projektowych. Większość programistów powinna umieć napisać kod dla głównych scenariuszy bez poważniejszych problemów; jeśli nie potrafią, należy przeprojektować API. Chociaż to kosztowne rozwiązanie, zauważyliśmy, że w rzeczywistości na dłuższą metę pozwala oszczędzić zasoby, bo koszt naprawy nieprzydatnego interfejsu API tak, by nie wymagało to zmian w istniejącym kodzie, jest kolosalny.

W następnym punkcie opisano, jak ważne jest projektowanie interfejsów API tak, by nie zniechęcały przy pierwszym kontakcie. Nazywane jest to zasadą niskiego progu wejścia.

2.2.2. Zasada niskiego progu wejścia

Dzisiaj wielu programistów jest przekonanych, że bardzo szybko nauczy się podstaw nowych frameworków. Chcą tego dokonać podczas eksperymentów z elementami frameworka na zasadzie *ad hoc*, a czas na pełne zrozumienie całej architektury zamierzają przeznaczyć tylko wtedy, jeśli zainteresuje ich jakaś funkcja albo będą musieli wyjść poza proste scenariusze. Pierwsze spotkanie ze źle zaprojektowanym API może pozostawić trwałe wrażenie, że framework jest skomplikowany, i zniechęcić niektórych programistów do jego używania. Właśnie dlatego bardzo ważne jest, by frameworki miały bardzo niski próg wejścia dla programistów, którzy po prostu chcą z nimi poeksperymentować.

Zasada projektowania frameworków

Frameworki muszą mieć niski próg wejścia dla nieprofesjonalnych użytkowników dzięki łatwości przeprowadzania eksperymentów.

Wielu programistów chce poeksperymentować z interfejsami API, by odkryć, co one robią, a potem powoli dostosowywać swój kod, tak by ich program robił to, czego naprawdę chcą.

PAUL VICK Większość programistów, niezależnie od języka, w którym pracuje, uczy się przez działanie. Dokumentacja może im dać jakieś pojęcie, co powinno nastąpić, ale wszyscy wiemy, że tak naprawdę nigdy nie nauczymy się, jak coś działa, jeśli się do tego nie zabierzemy i nie zaczniemy przy tym manipulować, by zrobić coś przydatnego. Do takiego eksperymentalnego podejścia do programowania zachęca szczególnie środowisko Visual Basic. Chociaż nigdy nie unikamy przemyśleń i planowania z góry, próbujemy sprawić, by proces nauki i programowania przebiegał w sposób ciągły. Zachęca do tego pisanie interfejsów API, które są oczywiste i nie wymagają złożonej wiedzy na temat wzajemnej interakcji wielu obiektów lub interfejsów API. (Tak naprawdę wydaje się, że uwaga ta dotyczy większości języków, nie tylko programu Visual Basic).

Niektóre interfejsy API pozwalają na eksperymenty, a inne nie. Łatwy do eksperymentowania interfejs API musi spełniać następujące warunki:

- Pozwalać w typowych zadaniach programistycznych na łatwą identyfikację właściwego zestawu typów i składowych. Niełatwo eksperymentować z przestrzenią nazw utworzoną do przechowywania popularnych interfejsów API zawierającą 500 typów, z których w zwykłych scenariuszach ważna jest naprawdę tylko garstka. To samo dotyczy używanych w głównych scenariuszach typów z wieloma składowymi utworzonymi tylko do bardzo zaawansowanych zastosowań.

CHRIS ANDERSON Na taki właśnie problem natknęliśmy się w początkach projektu Windows Presentation Foundation (WPF). Mieliśmy wspólny typ bazowy o nazwie `Visual`, od którego pochodziły prawie wszystkie nasze elementy. Problem polegał na tym, że wprowadzono w nim składowe wprost niezgodne z modelem obiektowym elementów będących jego dalszymi pochodnymi. Chodziło konkretnie o ich dzieci. Typ `Visual` miał pojedynczą hierarchię elementów wizualnych do renderowania, ale w jego elementach pochodnych chcieliśmy wprowadzić elementy podrzędne specyficzne dziedzinowo (tak by np. typ `TabControl` przyjmował tylko obiekty klasy `TabPages`). Zamiast komplikować model obiektowy każdego elementu, rozwiązaliśmy problem przez utworzenie klasy `VisualOperations` ze składowymi statycznymi działającymi na typie `Visual`.

- Pozwalać programiście używać API natychmiast, niezależnie od tego, czy wykonuje to, czego programista ostatecznie chce. Nie jest łatwo eksperymentować z frameworkiem wymagającym kompleksowej inicjacji lub utworzenia instancji szeregu typów, a następnie połączenia ich ze sobą. Podobnie interfejsy API bez wygodnych przeciążeń (przeciążonych składowych z krótszymi listami parametrów) albo ze złymi wartościami domyślnymi właściwości stanowią dużą barierę dla programistów, którzy chcą po prostu poeksperymentować z tymi interfejsami.

CHRIS ANDERSON Pomyśl o modelu obiektów jak o mapie: musisz umieścić wyraźne znaki wyjaśniające, jak dostać się z jednego miejsca do drugiego. Właściwość powinna jasno wskazywać, co robi, jakie wartości przyjmuje i co się stanie po jej ustawieniu. Bardzo źle jest, jeśli nakierowuje na abstrakcyjny typ bazowy, niemający oczywistych pochodnych. Negatywnym przykładem jest sposób, w jaki uwidoczniono animacje we frameworku WPF: typ bazowy animacji nosił nazwę `Timeline` (linia czasu), ale w przestrzeni nazw nic tym słowem się nie kończyło. Okazuje się, że po typie `Timeline` dziedziczyła klasa `Animation` i istniało mnóstwo typów w rodzaju `DoubleAnimation`, `ColorAnimation` itd., ale pomiędzy typem właściwości a dozwolonymi elementami, którymi można było ją wypełnić, nie było żadnego związku.

- Pozwalać na łatwe znajdowanie i naprawianie błędów spowodowanych nieprawidłowym użyciem API. Na przykład wyjątki zgłaszane przez interfejsy API powinny jasno opisywać, co należy zrobić, by naprawić problem.

CHRIS SELLS Gdy sam programuję, bardzo lubię komunikaty o błędach, które mówią, co zrobiłem źle i jak to naprawić. Często niestety uzyskuję tylko tę pierwszą informację, gdy tak naprawdę zależy mi jedynie na tej drugiej.

Kolejne zalecenia pozwolą przystosować framework dla programistów, którzy chcą się uczyć poprzez eksperymentowanie.

✓ **TAK** Dopilnuj, by przestrzenie nazw w każdym głównym obszarze funkcji zawierały tylko typy stosowane w najczęściej spotykanych przypadkach. Typy używane w zaawansowanych scenariuszach należy umieszczać w podobszarach nazw.

Na przykład przestrzeń nazw `System.Net` zawiera najważniejsze interfejsy API związane z pracą w sieci. Bardziej zaawansowane interfejsy API dotyczące gniazd znajdują się w podobszarze `System.Net.Sockets`.

ANTHONY MOORE Działa to również w drugą stronę, co można wyrazić następująco: „Nie zagrzebuj często używanego typu w jednej przestrzeni nazw z dużo rzadziej stosowanymi". Przykładem jest klasa `StringBuilder` Żałowaliśmy później, że nie umieściliśmy jej w obszarze nazw `System`. Znajduje się w przestrzeni `System.Text`, ale jest dużo częściej używana niż pozostałe zawarte tam typy i nie jest z nimi blisko związana.

Niemniej jednak jest to jedyny przypadek niespełnienia tej odwrotnej reguły w przestrzeni `System`. Dużo bardziej zaszkodziło umieszczenie tam zbyt wielu rzadko używanych typów.

✓ **TAK** Zapewnij proste przeciążenia konstruktorów i metod. Proste przeciążenie ma bardzo małą liczbę parametrów, z których każdy jest typu podstawowego (ang. *primitive*).

✗ **NIE** Typy stosowane w często występujących sytuacjach nie powinny mieć składowych zarezerwowanych dla zaawansowanych scenariuszy.

BRAD ABRAMS Jedną z ważnych zasad stosowanych przy projektowaniu platformy .NET Framework było dodawanie przez odejmowanie. Przez usuwanie funkcji z frameworka (lub niedodawanie ich tam nigdy) możemy faktycznie poprawić wydajność programistów, gdyż będą mieli do czynienia z mniejszą liczbą koncepcji. Nieuwzględnienie wielokrotnego dziedziczenia jest klasycznym przykładem dodawania przez odejmowanie na poziomie środowiska CLR.

✗ **NIE** wymagaj od użytkowników w najprostszych scenariuszach jawnego tworzenia instancji więcej niż jednego typu.

KRZYSZTOF CWALINA Wydawcy książek powiadają, że liczba sprzedanych egzemplarzy jest odwrotnie proporcjonalna do liczby zawartych w książce równań. Wersja tego prawa dla projektantów frameworków brzmi następująco: Liczba programistów, którzy będą używać frameworka, jest odwrotnie proporcjonalna do liczby jawnych wywołań konstruktorów wymaganych do implementacji dziesięciu najprostszych scenariuszy.

✗ **NIE** wymagaj od użytkowników przeprowadzania rozbudowanych inicjacji w celu zaprogramowania podstawowych scenariuszy.

Interfejsy API stosowane w najczęściej występujących sytuacjach należy zaprojektować tak, by wymagały jak najmniej inicjacji. Najlepiej, by do rozpoczęcia pracy z typem przeznaczonym dla podstawowych scenariuszy wystarczył konstruktor domyślny lub taki, który ma jeden prosty parametr.

```
var zipCodes = new Dictionary<string,int>();
zipCodes.Add("Radom",26600);
zipCodes.Add("Siemianowice",41100);
```

Jeśli niezbędna jest jakaś inicjacja, wyjątek zgłoszony wskutek jej niedokonania powinien jasno wyjaśniać, co należy zrobić.

STEVEN CLARKE Od czasu pierwszego wydania tej książki przeprowadziliśmy w tej dziedzinie ważne badania użyteczności. Raz za razem obserwowaliśmy, że typy wymagające rozbudowanej inicjacji znacznie podnoszą próg wejściowy dla programistów. Konsekwencje są takie, że niektórzy z nich zdecydują się nie korzystać z takiego typu i poszukają czegoś innego, co przyniesie rezultaty, inni użyją typu niewłaściwie, a tylko niewielu nauczy się w końcu, jak z niego poprawnie korzystać.

Przykładem obszaru funkcji, który sprawiał trudność użytkownikom z powodu wymogu rozbudowanej inicjacji, jest zbiór bibliotek `ADO.NET`. Nawet w najprostszych scenariuszach oczekiwano od użytkowników zrozumienia złożonych interakcji i zależności pomiędzy szeregiem typów. Aby skorzystać z tej funkcji, nawet w najprostszych scenariuszach użytkownicy musieli tworzyć instancje kilku typów (`DataSet`, `DataAdapter`, `SqlConnection` i `SqlCommand`) i łączyć je ze sobą. Należy zauważyć, że wiele z tych problemów rozwiązano w .NET Framework 2.0 przez dodanie klas pomocniczych, co bardzo uprościło podstawowe scenariusze.

✓ **TAK** Jeśli to możliwe, nadaj odpowiednie wartości domyślne wszystkim właściwościom i parametrom (przy użyciu wygodnych przeciążeń).

Dobrą ilustracją tej koncepcji jest komponent `System.Messaging.MessageQueue`. Do wysłania komunikatu za jego pomocą wystarczy przekazać do konstruktora ciąg ścieżki i wywołać metodę `Send`. Priorytet, algorytmy szyfrowania i inne właściwości komunikatu można dostosować do własnych potrzeb przez dodanie kodu do prostego scenariusza.

```
var ordersQueue = new MessageQueue(path);
ordersQueue.Send(order); // Używa domyślnego priorytetu, szyfrowania itd.
```

Rekomendacji tych nie można się ślepo trzymać. Projektanci frameworków nie powinni wprowadzać wartości domyślnych, jeśli może to zwieść na manowce. Taka wartość nie powinna nigdy powodować luki w zabezpieczeniach ani przyczyniać się do fatalnego działania kodu.

STEPHEN TOUB Podczas decydowania o „wartościach domyślnych" bardzo ważne jest, by poznać główne przypadki użycia API, i — na ile to możliwe — spróbować przewidzieć, jakie będą one w przyszłości. Jeden z moich najważniejszych przypadków, które „chciałbym móc zrobić od nowa", dotyczy przestrzeni nazw `System.Threading.Tasks`. Początkowo zaprojektowaliśmy interfejsy API pod kątem przetwarzania równoległego na procesorach, ale z czasem okazało się, że podstawowe przypadki użycia dotyczą głównie asynchronicznych operacji wejścia i wyjścia. Niektóre z początkowych wartości domyślnych ułatwiały to pierwsze zastosowanie, a niekorzystnie wpływały na to drugie. W stosownym czasie zajęliśmy się tymi kwestiami przy okazji dodawania łatwiejszych w użyciu interfejsów API, ale początkowe problemy i wynikające z nich trudności wciąż przeszkadzają programistom nadal korzystającym z pierwotnych interfejsów API.

JEREMY BARTON Bardzo ważne jest tu rozważenie kontrargumentów, a zachowanie właściwej równowagi może być trudne. Kryptograficzne interfejsy API platformy .NET zawierają pewną liczbę typów, których wartości domyślne miały być zarówno przyjazne, jak i bezpieczne. Ich „przyjazność" przetrwała, ale „bezpieczeństwo" to cel, któremu trudno dotrzymać kroku. Czasem przez dostarczenie wartości domyślnych wyświadcza się użytkownikom przysługę, a czasami wyrządza się im szkodę.

✓ **TAK** Komunikuj niewłaściwe zastosowanie interfejsów API przy użyciu wyjątków.

Wyjątki powinny jasno informować, jaki był powód ich wystąpienia i jak należy zmodyfikować kod, by pozbyć się problemu. Na przykład w celu zapisywania zdarzeń w komponencie `EventLog` wymagane jest najpierw nadanie wartości jego właściwości `Source`. Jeśli nie zostanie ona ustawiona przed wywołaniem metody `WriteEntry`, zgłaszany jest wyjątek z komunikatem: *Source property was not set before writing to the event log* (Właściwość źródła nie została ustawiona przed dokonaniem zapisu w dzienniku zdarzeń).

STEVEN CLARKE Podczas naszych badań użyteczności zaobserwowaliśmy, że wielu programistów uważa takie wyjątki za najlepszy rodzaj dokumentacji dla interfejsów API. Te wskazania są zawsze zgodne z kontekstem tego, co programista stara się osiągnąć, i rzeczywiście wspierają preferowane przez wiele osób podejście polegające na uczeniu się przez działanie.

W następnym punkcie opisano, jak ważne jest doprowadzenie do tego, by model obiektów stał się jak najbardziej samodokumentujący.

2.2.3. Zasada samodokumentujących się modeli obiektów

Wiele frameworków składa się z setek, jeśli nie tysięcy, typów oraz znacznie większej liczby składowych i parametrów. Programiści używający takich frameworków potrzebują w dużym stopniu prowadzenia i częstego przypominania o przeznaczeniu i właściwym użyciu interfejsów API. Dokumenty referencyjne nie wystarczą. Konieczność szukania w nich odpowiedzi na najprostsze pytania zajmuje dużo czasu i wytrąca programistę z rytmu pracy. Ponadto, jak już wspomniano, wiele osób preferuje kodowanie metodą prób i błędów, a do czytania dokumentacji ucieka się tylko wtedy, gdy zawiedzie je intuicja.

Z tych wszystkich powodów bardzo ważne jest projektowanie interfejsów API tak, by nie wymagały od programisty czytania dokumentacji za każdym razem, gdy chce wykonać proste zadanie. Zauważyliśmy, że postępowanie zgodne z kilkoma prostymi zaleceniami pomaga tworzyć intuicyjne interfejsy API, które stosunkowo dobrze same się dokumentują.

Zasada projektowania frameworków

W prostych scenariuszach frameworki muszą nadawać się do użycia bez korzystania z dokumentacji.

CHRIS SELLS Nigdy nie lekceważ możliwości mechanizmu IntelliSense, gdy prognozujesz, w jaki sposób programiści będą się uczyć korzystać z Twojego frameworka. Jeśli interfejs API jest intuicyjny, IntelliSense zapewni 80 procent tego, co nowicjusz potrzebuje do szczęścia i pomyślności w korzystaniu z biblioteki. Zoptymalizuj ją pod kątem IntelliSense.

KRZYSZTOF CWALINA Dokumenty referencyjne nadal są bardzo ważną częścią frameworka. Niemożliwe jest zaprojektowanie całkowicie samodokumentującego się interfejsu API. Różni ludzie, w zależności od ich umiejętności i doświadczeń, nie będą potrzebowali objaśnień w różnych obszarach frameworka. Dokumentacja ma też krytyczne znaczenie dla wielu użytkowników, którzy chcą poświęcić czas na poznanie z góry całego projektu frameworka. Dla takich użytkowników pouczająca, zwięzła i kompletna dokumentacja jest równie ważna jak niewymagające wyjaśnień modele obiektowe.

- ✓ **TAK** Dopilnuj, by interfejsy API były intuicyjne i mogły być z powodzeniem używane w podstawowych scenariuszach bez konieczności uciekania się do dokumentów referencyjnych.
- ✓ **TAK** Doskonale udokumentuj wszystkie interfejsy API.
- ✓ **TAK** Podawaj przykłady kodu ilustrujące, jak używać najważniejszych interfejsów API w typowych scenariuszach.

  Nie wszystkie interfejsy API mogą być zrozumiałe same z siebie, a niektórzy programiści chcą dokładnie poznać te interfejsy, zanim zaczną ich używać.

Aby framework sam się dokumentował, należy ostrożnie dobierać nazwy i typy, starannie projektować wyjątki itd. W kolejnych podpunktach zostaną opisane niektóre najważniejsze zagadnienia dotyczące projektowania samodokumentujących się interfejsów API.

2.2.3.1. Nazewnictwo

Najłatwiejszą, jednak bardzo rzadko wykorzystywaną, możliwością sprawienia, by frameworki same się dokumentowały, jest zarezerwowanie prostych i intuicyjnych nazw dla typów, których programiści powinni używać (tworzyć ich instancje) w najczęstszych sytuacjach. Projektanci frameworków często marnują najlepsze nazwy dla rzadziej używanych typów, którymi większość użytkowników się nie zajmuje.

Na przykład nadanie abstrakcyjnej klasie bazowej nazwy `File` (plik), a potem dostarczenie konkretnego typu `NtfsFile` sprawdzi się tylko przy założeniu, że wszyscy użytkownicy przed rozpoczęciem korzystania z interfejsów API poznają hierarchię dziedziczenia. Jeśli użytkownik nie zna hierarchii, to pierwszą rzeczą, jakiej spróbuje użyć, najczęściej bez powodzenia, jest typ `File`. Choć takie nazewnictwo z punktu widzenia projektowania obiektowego jest poprawne (w końcu typ `NtfsFile` jest rodzajem klasy `File`), nie zdaje testu użyteczności, bo większość programistów intuicyjnie skorzystałaby z nazwy `File`.

> **KRZYSZTOF CWALINA** Projektanci platformy .NET Framework spędzają mnóstwo czasu na dyskusjach na temat wariantów nazw głównych typów. Większość identyfikatorów ma tu dobrze wybrane nazwy. Przypadki, w których wybór nazw okazał się mniej fortunny, wynikały ze skupienia się na koncepcjach i abstrakcjach zamiast na głównych scenariuszach.

Kolejna rekomendacja to używanie opisowych nazw identyfikatorów, które jasno określają, co robi każda z metod i co reprezentuje sobą każdy typ i parametr. Projektanci frameworków nie powinni się obawiać dość szczegółowych nazw identyfikatorów. Na przykład definicja `EventLog.DeleteEventSource(string source, string machineName)` może sprawiać wrażenie rozwlekłej, ale uważamy, że pod względem użyteczności jest w sumie korzystna.

Opisowe nazwy można nadawać tylko metodom, które mają prostą i jasną semantykę. To kolejny powód, dla którego unikanie złożonej semantyki jest zasadą, która świetnie się sprawdza przy projektowaniu.

Ogólny wniosek jest taki, żeby wyjątkowo rozważnie wybierać nazwy identyfikatorów. Wybór nazw to jedna z najważniejszych decyzji projektowych podejmowanych przez projektanta frameworków. Po wydaniu interfejsu API zmiana nazw identyfikatorów jest niezwykle trudna i kosztowna.

✓ **TAK** Niech dyskusja o wyborze nazw identyfikatorów będzie istotną częścią przeglądów specyfikacji.

Od użycia jakich typów zaczyna się większość scenariuszy? Jakie nazwy przyjdą pierwsze na myśl większości osób, które będą próbowały zaimplementować dany scenariusz? Czy odpowiadają one nazwom najczęściej używanych typów? Dla przykładu: ponieważ większości ludzi mających do czynienia z plikowymi operacjami wejścia i wyjścia przychodzi na myśl nazwa `File`, powinien ją nosić główny typ umożliwiający dostęp do plików.

Należy też omówić najczęściej używane metody najbardziej popularnych typów i wszystkie ich parametry. Czy każda osoba obeznana z Twoimi rozwiązaniami technicznymi, ale nie z tym konkretnym projektem, szybko poprawnie i z łatwością rozpozna te metody i będzie umiała je wywoływać?

✗ **NIE** obawiaj się używać szczegółowych nazw identyfikatorów, jeśli takie postępowanie sprawi, że interfejs API stanie się samodokumentujący.

Większość nazw identyfikatorów powinna jasno określać, co robi każda z metod i co reprezentuje sobą każdy typ i parametr.

BRENT RECTOR Programiści odczytują nazwy identyfikatorów setki, jeśli nie tysiące razy częściej, niż je piszą. W nowoczesnych edytorach nawet potrzeba ich wpisywania jest minimalna. Dłuższe nazwy pozwalają szybciej znaleźć odpowiedni typ lub składową dzięki mechanizmowi IntelliSense. Dodatkowo kod, w którym typy są identyfikowane przez dobrze wybrane nazwy, jest bardziej zrozumiały i na dłuższą metę łatwiejszy w utrzymaniu.

Uwaga szczególnie do programistów języków z rodziny C: uwolnijcie się z więzów zmniejszonej wydajności spowodowanej przyzwyczajeniem do stosowania enigmatycznych identyfikatorów.

✓ **PRZEMYŚL** Wcześnie angażuj w proces projektowania frameworka autorów dokumentacji technicznych. Świetnie dostrzegą oni konstrukcje o źle wybranych nazwach i koncepcje, które trudno byłoby wytłumaczyć użytkownikom.

✓ **PRZEMYŚL** Zastrzeż najlepsze nazwy dla najczęściej używanych typów.

Jeśli spodziewasz się, że w przyszłości dodasz więcej wysokopoziomowych interfejsów API, nie obawiaj się zarezerwować dla nich w pierwszej wersji frameworka najlepszej nazwy.

ANTHONY MOORE Są też inne powody, by unikać zbyt ogólnych nazw, nawet jeśli nigdy później nie miałyby zostać użyte. Bardziej konkretne nazwy ułatwiają zrozumienie interfejsu API i poprawiają jego czytelność. Ktoś, kto zobaczy w kodzie ogólną nazwę, pomyśli zapewne, że ma ona bardzo szerokie zastosowanie, tak więc używanie ogólnej nazwy dla czegoś wyspecjalizowanego jest mylące. Bardziej opisowe nazwy ułatwiają też identyfikację scenariusza lub rozwiązania technicznego, z którym związany jest dany typ.

2.2.3.2. Wyjątki

Wyjątki odgrywają kluczową rolę w projektowaniu samodokumentujących się frameworków. Komunikat o wyjątku powinien informować programistę, co zrobił niepoprawnie. Poniższy urywek kodu spowoduje na przykład zgłoszenie wyjątku z komunikatem `Source property was not set before writing to the event log.` (Właściwość źródła nie została ustawiona przed dokonaniem zapisu w dzienniku zdarzeń).

```
// C#
var log = new EventLog();
// Nie nadano jeszcze wartości właściwości źródła dziennika.
log.WriteEntry("Witaj, świecie");
```

✓ **TAK** Powiadamiaj programistę o błędnym użyciu frameworka za pomocą komunikatów o wyjątkach.

Załóżmy na przykład, że użytkownik zapomniał ustawić właściwości `Source` komponentu `EventLog`. Wtedy każde wywołanie metody wymagającej, by ta właściwość miała nadaną wartość, spowoduje zgłoszenie wyjątku. Komunikat o wyjątku powinien zawierać jasną informację na temat przyczyny błędu. Więcej zaleceń dotyczących projektowania wyjątków i zawartych w nich komunikatów znajduje się w rozdziale 7.

2.2.3.3. Silna kontrola typów

Silna kontrola typów (ang. *strong typing*) jest prawdopodobnie najważniejszym czynnikiem wyznaczającym intuicyjność interfejsów API. To oczywiste, że łatwiej dokonać wywołania `Customer.Name` niż `Customer.Properties["Name"]`. Poza tym właściwość `Name` (nazwa) jest bardziej użyteczna, gdy zwraca nazwę jako typ `String`, niż gdyby miała zwracać klasę `Object`.

Bywa, że korzystanie ze zbiorów właściwości (ang. *property bags*), wywołań z późnym wiązaniem (ang. *late-bound calls*) i innych luźno typowanych interfejsów API jest niezbędne, ale powinny to być wyjątki od reguły, a nie częsta praktyka. Ponadto projektanci powinni rozważyć zapewnienie silnie typowanych konstrukcji pomocniczych dla najczęstszych operacji, które użytkownicy wykonują w warstwie API niemającej silnej kontroli typów. Na przykład typ `Customer` (klient) może zawierać zbiór właściwości, ale powinien zapewniać również silnie typowane interfejsy API dla popularnych właściwości takich jak `Name`, `Address` itd.

✓ **TAK** Jeśli to możliwe, udostępniaj silnie typowane interfejsy API.

Nie polegaj wyłącznie na słabo typowanych interfejsach API, np. zbiorach właściwości. W przypadkach, gdy taki zbiór jest potrzebny, przygotuj typowane odpowiedniki najpopularniejszych zawartych w nim właściwości.

> **VANCE MORRISON** Silna kontrola typów (i dzięki temu dużo lepszy mechanizm IntelliSense) jest kluczowym powodem, dla którego frameworki platformy .NET łatwiej „poznać przez programowanie" niż typowe API modelu COM. Od czasu do czasu wciąż muszę korzystać z możliwości dostępnych w modelu COM i dopóki jest tam silne typowanie, dobrze sobie z tym radzę. Bardzo często jednak interfejsy API zwracają lub pobierają parametry w postaci ogólnych obiektów lub ciągów albo — gdy potrzebne jest wyliczenie — przekazują wartość DWORD, a wtedy potrzebuję dziesięć razy więcej czasu, by odkryć, co dokładnie ma być przekazane.

2.2.3.4. Spójność

Kolejną skuteczną techniką projektowania samodokumentujących się frameworków jest zachowanie spójności z istniejącymi interfejsami API, z którymi użytkownik jest już obeznany. Ważna jest spójność tak z innymi interfejsami API platformy .NET, jak i niektórymi starszymi. Mając to w pamięci, nie należy traktować korzystania ze starszych interfejsów API lub wadliwie zaprojektowanych interfejsów istniejącego frameworka jako wymówki, by w ogóle nie stosować się do wytycznych opisanych w tej książce — nie należy też jednak bez powodu arbitralnie zmieniać dobrych, ustalonych wzorców i konstrukcji.

✓ **TAK** Zapewnij spójność z platformą .NET i innymi frameworkami, z którymi użytkownicy prawdopodobnie będą mieli styczność.

Zachowanie spójności doskonale sprzyja ogólnej użyteczności. Jeśli użytkownik jest obeznany z jakimś fragmentem frameworka przypominającego Twój interfejs API, Twój projekt wyda mu się naturalny i intuicyjny. Twój interfejs API powinien odbiegać od interfejsów API platformy .NET tylko w takich miejscach, w których występuje coś unikatowego.

2.2.3.5. Ograniczanie typów abstrakcyjnych

API dla typowych scenariuszy nie powinno zawierać wielu interfejsów i klas abstrakcyjnych, ale odpowiadać fizycznym lub dobrze znanym logicznym częściom systemu.

Jak wcześniej wspomniano, celem standardowych metodyk projektowania obiektowego jest tworzenie projektów zoptymalizowanych pod kątem łatwości utrzymania bazy kodu. Ma to sens, bo koszt utrzymania stanowi największą część całych nakładów na opracowanie oprogramowania. Jedną z metod na ułatwienie utrzymania jest korzystanie z typów abstrakcyjnych — interfejsów lub klas abstrakcyjnych. Z tego powodu w nowoczesnych metodykach programowania tworzonych jest ich wiele.

Problem polega na tym, że frameworki zawierające wiele typów abstrakcyjnych (abstrakcji) wymagają, by użytkownicy stali się ekspertami od ich architektury, zanim zaczną implementować najprostsze scenariusze. Większość programistów nie ma jednak ochoty zostawać ekspertami od wszystkich interfejsów API dostarczanych przez takie frameworki ani nie ma żadnego uzasadnienia biznesowego po temu. Programiści wymagają, by w przypadku zwykłych scenariuszy interfejsy API były na tyle proste, by można ich było używać bez zrozumienia, jak współdziałają ze sobą całe obszary funkcji. Standardowe metodyki projektowania nie są zoptymalizowane pod tym kątem i nikt nigdy tego nie twierdził.

Oczywiście abstrakcje mają swoje miejsce w konstrukcji frameworka. Mogą być na przykład niezwykle użyteczne dla polepszenia testowalności i ogólnej rozszerzalności frameworków. Często taką rozszerzalność umożliwiają właściwie zaprojektowane abstrakcje. W osiągnięciu odpowiedniej równowagi pod tym względem powinien pomóc rozdział 6., w którym opisano projektowanie rozszerzalnych interfejsów API.

✗ **UNIKAJ** wielu abstrakcji w interfejsach API stosowanych w najważniejszych scenariuszach.

KRZYSZTOF CWALINA Abstrakcje są prawie zawsze niezbędne, ale zbyt duża ich liczba wskazuje, że systemy są przesadnie skomplikowane. Projektanci frameworków powinni starać się tworzyć je dla klientów, a nie dla własnej intelektualnej przyjemności.

JEFF PROSISE Konstrukcja ze zbyt dużą liczbą abstrakcji może mieć też wpływ na wydajność. Kiedyś pracowałem dla firmy, która przeprojektowała swój produkt w sposób wysoce obiektowy. Jej programiści zamodelowali *wszystko* jako klasy i w efekcie uzyskali absurdalnie głęboko zagnieżdżone hierarchie obiektów. Przeprojektowanie było częściowo podyktowane zamiarem polepszenia wydajności, ale „usprawnione" oprogramowanie działało cztery razy wolniej niż oryginał!

VANCE MORRISON Każdy, kto miał „przyjemność" podczas debugowania kodu C++ przedzierać się przez biblioteki STL, wie, że abstrakcja jest bronią obosieczną. Jeśli abstrakcji jest zbyt wiele, kod staje się bardzo trudny do zrozumienia, bo trzeba pamiętać, co te wszystkie abstrakcyjne nazwy naprawdę znaczą w bieżącym scenariuszu. Przesada w używaniu uogólnień (ang. *generics*) i dziedziczenia to często oznaka tego, że kod jest zbyt ogólny.

CHRIS SELLS Często mówi się, że każdy problem informatyczny można rozwiązać przez dodanie warstwy abstrakcji. Niestety równie często przyczyniają się one do problemów w kształceniu programistów.

2.2.4. Zasada architektury warstwowej

Nie od wszystkich programistów oczekuje się rozwiązywania tego samego rodzaju problemów. Różni programiści często wymagają od używanych przez nich frameworków i spodziewają się po nich innych poziomów abstrakcji i odmiennego stopnia kontroli. Niektórzy programiści, zazwyczaj używający języków C++ lub C#, cenią wyraziste i zaawansowane interfejsy API. Nazywamy je niskopoziomowymi, ponieważ często zapewniają niski poziom abstrakcji. Dla odmiany inni programiści, zwykle używający języków C# lub VB.NET, cenią interfejsy API zoptymalizowane pod kątem wydajności i prostoty. Nazywamy je wysokopoziomowymi, bo zapewniają wyższy poziom abstrakcji. Dzięki użyciu konstrukcji warstwowej można zbudować jeden framework zaspokajający te odmienne potrzeby.

Zasada projektowania frameworków

Konstrukcja warstwowa pozwala na to, by framework był zarówno zaawansowany, jak i łatwy w użyciu.

PAUL VICK Jeden z powodów przeniesienia środowiska Visual Basic na platformę .NET był taki, że wielu programistów VB napotykało problemy, gdy musiało użyć niskopoziomowych interfejsów API w celu zastosowania pewnych funkcji, które nie były dostępne w dostarczanych przez nas interfejsach wyższego poziomu. To, że programiści VB spędzają początkowo większość czasu na szybkim opracowywaniu aplikacji przy użyciu wysokopoziomowych interfejsów API, nie zmienia faktu, że wcześniej czy później większość z nich musi zmodyfikować lub dopracować aplikacje, co wymaga zwykle pracy z interfejsami niższego poziomu celem uzyskania tych dodatkowych elementów funkcyjnych. Przy projektowaniu niskopoziomowych interfejsów API należy więc brać też poważnie pod uwagę programistów VB.

Generalną wytyczną do zbudowania jednego frameworka skierowanego do szerokiego grona programistów jest rozłożenie zestawu API na typy niskopoziomowe, udostępniające całe bogactwo i możliwości frameworka, oraz typy wysokopoziomowe, opakowujące niższy poziom w wygodne interfejsy API.

Jest to bardzo skuteczna technika upraszczania. W przypadku jednowarstwowego API projektant jest często zmuszany do wyboru między bardziej złożoną konstrukcją frameworka a brakiem obsługi niektórych scenariuszy. Posiadanie niskopoziomowej warstwy o dużych możliwościach zapewnia swobodę określania zakresu wysokopoziomowego interfejsu API, tak by faktycznie odpowiadał najważniejszym scenariuszom.

W niektórych przypadkach jedna z warstw może nie być potrzebna, na przykład w pewnych obszarach funkcji uwidaczniane mogą być tylko niskopoziomowe interfejsy API.

Przykładem takiej konstrukcji warstwowej są interfejsy API platformy .NET zapewniające obsługę formatu JSON. Programistom ceniącym możliwości i wyrazistość typ `Utf8JsonReader` zapewnia niskopoziomowy parser pozwalający na pisanie kodu w odniesieniu do poszczególnych tokenów zawartych w ładunku JSON. Platforma .NET zawiera jednak również typy `JsonDocument` i `JsonElement`, oparte na klasie `Utf8JsonReader` i pozwalające na pisanie kodu z użyciem koncepcji wyższego poziomu, takich jak struktura dokumentu, bez przejmowania się obliczaniem głębokości obiektów ani usuwaniem sekwencji specjalnych z ciągów. Typy te mają spójne działanie i identyfikatory, ale znajdują się na różnych warstwach, przeznaczonych dla odmiennych scenariuszy i grup programistów.

Istnieją dwa główne sposoby rozmieszczenia warstw API w przestrzeniach nazw:

- w oddzielnych obszarach,
- w tym samym obszarze.

2.2.4.1. Umieszczenie warstw w oddzielnych przestrzeniach nazw

Jednym ze sposobów uporządkowania frameworka jest umieszczenie wysokopoziomowych i niskopoziomowych typów w odrębnych, ale związanych ze sobą przestrzeniach nazw. Zaletą jest to, że w najczęstszych sytuacjach typy niskopoziomowe pozostają ukryte, lecz są w bliskim zasięgu programistów, którzy muszą implementować bardziej złożone scenariusze.

W ten sposób rozmieszczone są interfejsy API platformy .NET dotyczące sieci. Typy: niskiego poziomu `System.Net.Sockets.Socket`, średniego poziomu `System.Net.Security.SslStream` i wysokiego poziomu `System.Net.Http.HttpClient` znajdują się w różnych przestrzeniach nazw. Implementacja tego ostatniego zależy ostatecznie tak od typu `Socket`, jak i `SslStream`, ale większość programistów pracujących z protokołem HTTP może korzystać z klasy `HttpClient` bez potrzeby używania typów niższych poziomów.

Ta metoda uporządkowania przestrzeni nazw powinna być stosowana w zdecydowanej większości frameworków.

2.2.4.2. Umieszczenie warstw w tej samej przestrzeni nazw

Drugim sposobem uporządkowania frameworka jest umieszczenie typów wysoko- i niskopoziomowych w tej samej przestrzeni nazw. Zaletą jest możliwość łatwego przejścia w razie potrzeby do bardziej złożonych funkcji. Wadą jest to, że występowanie w przestrzeni nazw skomplikowanych typów utrudnia w niektórych przypadkach pracę, nawet jeśli takie typy nie są używane.

To rozmieszczenie najlepiej sprawdza się przy prostych zestawach funkcji, na przykład przestrzeń nazw `System.Text` zawiera zarówno typy niskopoziomowe, takie jak `Encoder` i `Decoder`, jak i hierarchię klas wyższego rzędu `Encoding`.

> **STEVEN CLARKE** Pamiętaj, by pomyśleć o działaniu warstwowego API po uruchomieniu aplikacji. Jeśli na przykład programista pracuje z jedną z warstw, nie należy od niego wymagać przechwytywania wyjątków zgłaszanych w innej warstwie. Dopilnuj, by programiści piszący, czytający i poznający kod musieli zająć się tylko tym, co dzieje się na jednej warstwie, i mogli śmiało traktować inne warstwy jak czarne skrzynki.

✓ **PRZEMYŚL** Nadaj frameworkowi konstrukcję warstwową, w której interfejsy API wysokiego poziomu są zoptymalizowane pod kątem produktywności, a interfejsy niskiego poziomu — pod kątem możliwości i wyrazistości.

✗ **UNIKAJ** mieszania interfejsów API niskiego i wysokiego poziomu w jednej przestrzeni nazw, jeśli te pierwsze są bardzo złożone (tzn. zawierają wiele typów).

✓ **TAK** Dopilnuj, by warstwy jednego obszaru funkcji dobrze się ze sobą integrowały. Programiści powinni móc rozpocząć programowanie od jednej z warstw, a potem przejść do użycia drugiej bez konieczności przepisywania od nowa całej aplikacji.

Podsumowanie

Podczas projektowania frameworka bardzo ważne jest, by pamiętać, że jego odbiorcy są niejednorodni, tak pod względem potrzeb, jak i poziomu umiejętności. Przestrzeganie opisanych w tym rozdziale wytycznych daje gwarancję, że framework będzie przydatny dla zróżnicowanej grupy programistów.

Najlepsze wzorce tworzenia frameworków i bibliotek: poznaj i stosuj!

Projektant frameworka tworzy dla innych programistów. To odpowiedzialne zadanie: celem jest zapewnienie większości potrzebnych elementów, które po dostosowaniu i połączeniu mają się stać aplikacją. Dobrze zaprojektowany framework pozwala na wygodną i efektywną pracę. Jest prosty, łatwy do rozwijania i świetnie integruje się z innymi narzędziami programistycznymi, językami czy modelami aplikacji. Projektant musi więc dobrze się orientować w zasadach tworzenia interfejsów API, bibliotek i innych komponentów wielokrotnego użytku.

Ta książka jest trzecim, przejrzanym i zaktualizowanym wydaniem znakomitego wprowadzenia do programowania komponentów i ich bibliotek na platformie .NET. Położono w nim nacisk na zagadnienia projektowe bezpośrednio wiążące się z programowalnością frameworka. Przedstawione wytyczne, wypracowane przez lata rozwijania platformy .NET, wynikają z doświadczenia i wiedzy projektantów i ekspertów branżowych. Uwzględniają też innowacje w zakresie projektowania interfejsów API oraz programowania asynchronicznego i uproszczonego dostępu do pamięci. Poszczególne wytyczne zostały uporządkowane, wyjaśnione i bogato skomentowane. Dzięki temu można w pełni wykorzystać najlepsze wzorce języka C# 8, a także platform .NET Framework 4.8 i .NET Core.

W książce:

- Najważniejsze zasady projektowania nowoczesnych frameworków
- Typowe dla frameworków wzorce projektowe
- Wytyczne w zakresie nazw, typów, rozszerzalności i wyjątków
- Projektowanie skalowalnych bibliotek działających w chmurze
- Nowe techniki programowania asynchronicznego z wykorzystaniem typów Task i ValueTask
- Dostęp do pamięci za pomocą typów Memory<T> i Span<T>

Krzysztof Cwalina jest architektem oprogramowania w Microsofcie. Był członkiem założycielem zespołu .NET Framework. Obecnie pomaga w projektowaniu interfejsów API wielokrotnego użytku.

Jeremy Barton jest głównym inżynierem oprogramowania w Microsofcie. Pracuje w zespole .NET Core Libraries. Zdobył też doświadczenie w rozwijaniu małych frameworków w C#.

Brad Abrams jest kierownikiem grupy programów w Google i odpowiada za inkubację nowych projektów dla Asystenta Google. Jest też autorem i współautorem wielu publikacji.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

INFORMATYKA W NAJLEPSZYM WYDANIU

ISBN 978-83-283-7606-9

Cena: 119,00 zł

Pearson
Addison-Wesley