13A

Kontrakty kodu

Wprowadzone we Framework 4.0 kontrakty kodu umożliwiają metodom współpracę na podstawie zbioru wzajemnie obowiązujących zasad i szybkie reagowanie w przypadku, gdyby któraś z reguł została złamana.

Większość opisanych w tym rozdziale typów jest zdefiniowana w przestrzeniach nazw `System.Diagnostics` i `System.Diagnostics.Contracts`.

Ogólne omówienie kontraktów kodu

W rozdziale 13. książki *C# 7.0 w pigułce* wspomnieliśmy o koncepcji *asercji*, w której następuje sprawdzenie, czy w programie zostały spełnione określone warunki. W przypadku niespełnienia danego warunku oznacza to błąd w aplikacji, który zwykle może być obsłużony przez wywołanie debuggera (w przypadku kompilacji testowej) lub zgłoszenie wyjątku (w gotowym produkcie).

W asercji stosowana jest następująca reguła: jeśli wystąpi jakikolwiek problem, najlepiej będzie doprowadzić do awarii i zamknąć źródło błędu. Jest to zwykle rozwiązanie lepsze niż próba kontynuowania pracy z nieprawidłowymi danymi, co może doprowadzić do wygenerowania niewłaściwych wyników, powstania niechcianych efektów ubocznych lub skutkować później zgłoszeniem wyjątku w wydanej aplikacji (te wszystkie wymienione czynniki utrudniają postawienie właściwej diagnozy).

Mamy dwa sposoby na wymuszenie asercji:

- wywołanie metody `Assert()` w egzemplarzu klasy `Debug` lub `Trace`;
- zgłoszenie wyjątku (takiego jak `ArgumentNullException`).

W.NET Framework 4.0 wprowadzono nową funkcję o nazwie *kontrakty kodu*, która oba wymienione powyżej podejścia zastępuje ujednoliconym systemem. Ten system pozwala na przygotowanie nie tylko prostych asercji, ale również oferujących znacznie potężniejsze możliwości asercji opartych na *kontraktach*.

Kontrakty kodu wywodzą się z zasady programowania kontraktowego (ang. *design by contract*) znanego z języka programowania Eiffel, gdzie funkcje współdziałają ze sobą za pomocą systemu wzajemnych zobowiązań i korzyści. Ogólnie rzecz biorąc, funkcja wskazuje **warunki początkowe**, które muszą być spełnione przez klienty (komponenty wywołujące). W zamian gwarantuje spełnienie **warunków końcowych**, na których klienty mogą się opierać po zakończeniu działania danej funkcji.

Typy dla kontraktów kodu znajdują się w przestrzeni nazw `System.Diagnostics.Contracts`.

Typy obsługujące kontrakty kodu są wbudowane w platformę .NET Framework, natomiast na stronie: *https://github.com/Microsoft/CodeContracts* dostępny jest tzw. binarny rewriter oraz statyczne narzędzia przeznaczone do sprawdzania. Musisz zainstalować[1] te narzędzia, zanim będziesz mógł używać kontraktów kodu w Visual Studio.

Dlaczego warto używać kontraktów kodu?

Aby zilustrować koncepcję, przygotujemy metodę dodającą element do listy tylko wtedy, gdy ten jeszcze na niej nie istnieje. Zdefiniujemy dwa *warunki początkowe* i jeden *warunek końcowy*:

```
public static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  Contract.Requires (list != null);          // warunek początkowy
  Contract.Requires (!list.IsReadOnly);      // warunek początkowy
  Contract.Ensures (list.Contains (item));   // warunek końcowy
  if (list.Contains(item)) return false;
  list.Add (item);
  return true;
}
```

Warunki początkowe są definiowane przez wywołania `Contract.Requires()` i weryfikowane podczas rozpoczęcia wykonywania metody. Warunek końcowy jest zdefiniowany przez `Contract.Ensures()` i weryfikowany nie w miejscu występowania w kodzie, ale tam, *gdzie kończy się działanie metody*.

Warunki początkowe i końcowe działają podobnie jak asercje. W omawianym przykładzie pozwolą na wykrycie następujących błędów:

- wywołanie metody, gdy lista nie istnieje (`null`) lub jest tylko do odczytu;
- błąd w metodzie polegający na tym, że zapomnimy o dodaniu elementu do listy.

Warunki początkowe i końcowe muszą pojawiać się na początku metody. To dobre rozwiązanie, ponieważ w przypadku niepowodzenia spełnienia warunków kontraktu w kolejnych wersjach metody błąd zostanie wykryty

Co więcej, te warunki tworzą postać możliwego do odkrycia *kontraktu* dla danej metody. Metoda `AddIfNotPresent()` jest przedstawiana konsumentowi w poniższy sposób:

- „Musisz wywołać mnie wraz z niepustą listą pozwalającą na zapis w niej".
- „Kiedy zakończę działanie, ta lista będzie zawierała podany przez ciebie element".

[1] Gotowy do pobrania instalator wspomnianych narzędzi znajduje się na stronie: *https://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970 — przyp. tłum.*

Powyższe informacje mogą być umieszczone w pliku XML dokumentacji podzespołu (można to zrobić w Visual Studio, przechodząc do karty *Code Contracts* okna właściwości projektu, włączając dodawanie odwołań do kontraktów, a następnie zaznaczając pole wyboru *Emit Contracts into XML doc file*). Narzędzia takie jak SandCastle mogą później umieszczać w plikach dokumentacji szczegółowe informacje dotyczące kontraktu.

Kontrakty pozwalają także na przeanalizowanie poprawności programu przez statyczne narzędzia przeznaczone do weryfikacji kontraktu. Na przykład jeżeli spróbujemy wywołać metodę `AddIfNot ↪Present()` wraz z listą, której wartością może być `null`, statyczne narzędzie weryfikacji może wyświetlić odpowiednie ostrzeżenie jeszcze przed uruchomieniem programu.

Inną korzyścią płynącą z kontraktów jest łatwość użycia. W omawianym tutaj przykładzie łatwiejsze będzie wyraźne zdefiniowanie warunku końcowego w podanym miejscu zamiast w obu punktach wyjścia. Kontrakty obsługują również tzw. metody **inwariantów obiektu**, co jeszcze bardziej zmniejsza ilość powtarzającego się kodu i prowadzi do powstania bardziej niezawodnego egzekwowania założeń.

Warunki mogą być również zdefiniowane w elementach składowych interfejsu i metodach abstrakcyjnych, co jest niemożliwe w przypadku standardowych podejść w zakresie weryfikacji. Ponadto warunki w metodach wirtualnych nie będą mogły być przypadkowo pominięte w podklasach.

Kolejną korzyścią płynącą ze stosowania kontraktów kodu jest to, że zachowanie łamiące warunki kontraktu można bardzo łatwo dostosować do własnych potrzeb, i to na znacznie więcej sposobów niż jedynie przez wywołanie `Debug.Assert()` lub zgłoszenie wyjątku. Istnieje możliwość zagwarantowania, że informacje dotyczące złamania kontraktu zawsze zostaną zarejestrowane, nawet jeśli związany z tym wyjątek będzie przechwycony przez procedurę obsługi wyjątków znajdującą się na wyższych warstwach stosu wywołań.

Wadą użycia kontraktów kodu jest to, że implementacja .NET Framework opiera się na tzw. **binarnym rewriterze**, czyli narzędziu pozwalającym na modyfikację podzespołu po jego kompilacji. Spowalnia to proces kompilacji, jak również komplikuje usługi opierające się na wywołaniu kompilatora C# (niezależnie od tego, czy jawnie, czy za pomocą klasy `CSharpCodeProvider`).

Wymuszenie przestrzegania kontraktów kodu może skutkować także zmniejszeniem wydajności działania uruchomionej aplikacji, choć można to łatwo złagodzić przez zmniejszenie poziomu sprawdzania kontraktu w finałowej wersji programu.

Innym ograniczeniem związanym z kontraktami kodu jest brak możliwości ich użycia do wymuszenia operacji sprawdzania związanych z zapewnieniem bezpieczeństwa, ponieważ mogą zostać pominięte w trakcie działania aplikacji (np. przez obsługę zdarzenia `ContractFailed`).

Reguły dotyczące kontraktów

Kontrakty kodu składają się z *warunków początkowych*, *warunków końcowych*, *asercji* i metod *inwariantów obiektu*. To wszystko są możliwe do wykrycia asercje, różnice polegają jedynie na momencie, w którym przeprowadzana jest weryfikacja:

- *warunki początkowe* są weryfikowane podczas rozpoczęcia działania funkcji;
- *warunki końcowe* są weryfikowane przed zakończeniem działania funkcji;
- *asercje* są weryfikowane w miejscu, w którym występują w kodzie;
- metody *inwariantów obiektu* są weryfikowane po każdej metodzie publicznej w klasie.

Kontrakty kodu są definiowane wyłącznie przez wywoływanie metod (statycznych) w klasie `Contract`. Dzięki temu kontrakty są *niezależne od języka*.

Kontrakty mogą się pojawiać nie tylko w metodach, ale również w innych funkcjach, np.: konstruktorach, właściwościach, indeksach i operatorach.

Kompilacja

Właściwie wszystkie metody w klasie `Contract` są zdefiniowane wraz z atrybutem `[Conditional ↪("CONTRACTS_FULL")]`. Oznacza to, że dopóki nie zdefiniujemy symbolu `CONTRACTS_FULL`, większość kodu kontraktów będzie wyeliminowana. Visual Studio automatycznie definiuje ten symbol, jeśli włączymy sprawdzanie kontraktów kodu na stronie właściwości projektu. (Aby ta karta się pojawiła, konieczne jest pobranie wspomnianych wcześniej narzędzi i ich zainstalowanie).

Usunięcie symbolu `CONTRACTS_FULL` może się wydawać łatwym sposobem na całkowite wyłączenie sprawdzania wszystkich kontraktów. Jednak nie ma to zastosowania względem warunków `Requires<TException>` (do tego tematu wkrótce jeszcze powrócimy).

Jedynym sposobem całkowitego wyłączenia kontraktów w kodzie z uwzględnieniem `Requires<TException>` jest włączenie symbolu `CONTRACTS_FULL`, a następnie za pomocą binarnego rewritera pozbycie się całego kodu kontraktów przez ustalenie poziomu wymuszenia na „none” (czyli brak).

Binarny rewriter

Po przeprowadzeniu kompilacji kodu zawierającego kontrakty konieczne jest wywołanie narzędzia binarnego rewritera, czyli *ccrewrite.exe* (Visual Studio zrobi to automatycznie po zaznaczeniu opcji wskazującej na użycie kontraktów kodu). Binarny rewriter przenosi warunki końcowe (i metody *inwariantów obiektu*) we właściwe miejsca, wywołuje te warunki w nadpisanych metodach i zastępuje wywołania do `Contract` wywołaniami do *klasy kontraktów w środowisku uruchomieniowym*. Poniżej przedstawiono (uproszczoną) postać omówionego wcześniej przykładu po jego przetworzeniu przez binarny rewriter:

```
static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  __ContractsRuntime.Requires (list != null);
  __ContractsRuntime.Requires (!list.IsReadOnly);
  bool result;
  if (list.Contains (item))
    result = false;
  else
  {
    list.Add (item);
    result = true;
```

```
    }
    __ContractsRuntime.Ensures (list.Contains (item));    // warunek końcowy
    return result;
}
```

Jeżeli zapomnimy o wywołaniu binarnego rewritera, wówczas wywołania do `Contract` nie zostaną zastąpione wywołaniami `__ContractsRuntime`, co doprowadzi do zgłaszania wyjątków.

Typ `__ContractsRuntime` to domyślna klasa kontraktów w środowisku uruchomieniowym. W bardziej zaawansowanych scenariuszach można podać własną tego rodzaju klasę. W tym celu należy skorzystać z opcji `/rw` na karcie *Code Contracts* we właściwościach projektu w Visual Studio.

Typ `__ContractsRuntime` jest dostarczany wraz z binarnym rewriterem (nie jest to standardowy komponent platformy .NET Framework), więc tak naprawdę ten binarny rewriter wstrzykuje klasę `__ContractsRuntime` do skompilowanego podzespołu. Można przeanalizować ten kod przez rozłożenie dowolnego podzespołu, w którym została włączona obsługa kontraktów kodu.

Binarny rewriter oferuje również opcje pozwalające na pozbycie się sprawdzania części lub wszystkich kontraktów, co zostanie omówione dalej, w podrozdziale „Selektywne egzekwowanie kontraktów". Z reguły włącza się pełne sprawdzanie kontraktów dla konfiguracji *Debug* oraz sprawdzanie jedynie wybranych dla konfiguracji *Release*.

Asercja kontra zgłoszenie wyjątku w przypadku niepowodzenia

Binarny rewriter pozwala na dokonanie wyboru między wyświetleniem okna dialogowego i zgłoszeniem wyjątku `ContractException` w przypadku niepowodzenia spełnienia kontraktu. Pierwsza z wymienionych możliwości jest zwykle stosowana w konfiguracji *Debug*, natomiast druga — w konfiguracji *Release*. W celu włączenia możliwości w tej drugiej konfiguracji należy podczas wywoływania binarnego rewritera podać opcję `/throwonfailure` lub też usunąć zaznaczenie pola wyboru *Assert on contract failure* na karcie *Code Contracts* w oknie właściwości projektu w Visual Studio.

Do tego tematu dokładniej powrócimy w podrozdziale „Rozwiązywanie problemów z awariami podczas użycia kontraktów".

Czystość

Wszystkie funkcje wywoływane z argumentów przekazanych metodom kontraktu (`Requires()`, `Assumes()`, `Assert()` itd.) muszą być *czyste*, czyli nie mogą powodować efektów ubocznych (nie wolno im zmieniać wartości elementów składowych). Za pomocą atrybutu `[Pure]` konieczne jest zasygnalizowanie binarnemu rewriterowi, że wywoływana funkcja jest czysta:

```
[Pure]
public static bool IsValidUri (string uri) { ... }
```

W ten sposób poniższe wywołanie staje się prawidłowe:

```
Contract.Requires (IsValidUri (uri));
```

Narzędzia kontraktów niejawnie przyjmują założenie, że wszystkie akcesory `get` właściwości są czyste, podobnie jak wszystkie operatory C# (+, *, % itd.) oraz elementy składowe wybranych typów frameworka, m.in.: `string`, `Contract`, `Type`, `System.IO.Path`, a także operatory zapytań LINQ. Ponadto przyjmowane jest założenie, że metody wywoływane za pomocą delegatów oznaczonych atrybutem `[Pure]` również są czyste (atrybuty `Comparison<T>` i `Predicate<T>` są oznaczone przez `[Pure]`).

Warunki początkowe

Warunki początkowe kontraktu kodu są definiowane za pomocą metod: `Contract.Requires()`, `Contract.Requires<TException>` i `Contract.EndContractBlock()`.

Contract.Requires()

Wywołanie `Contract.Requires()` na początku funkcji wymaga zastosowania warunku początkowego:

```
static string ToProperCase (string s)
{
  Contract.Requires (!string.IsNullOrEmpty(s));
  ...
}
```

Przypomina to asercję, z wyjątkiem tego, że warunek początkowy stanowi możliwy do odkrycia fakt dotyczący funkcji i może być wyodrębniony ze skompilowanego kodu oraz używany przez dokumentację lub statyczne narzędzia sprawdzające. (Dzięki temu te narzędzia będą mogły ostrzegać, że gdzieś w programie następuje próba wywołania funkcji `ToProperCase()` wraz z wartością `null` lub z pustym ciągiem tekstowym).

Kolejną zaletą warunków początkowych jest to, że podklasy przeciążające metody wirtualne wraz z warunkami początkowymi nie mogą zabronić sprawdzenia tych warunków w klasie bazowej. Ponadto wszelkie warunki początkowe zdefiniowane w elementach składowych *interfejsu* będą niejawnie wplecione w konkretne implementacje (zob. podrozdział „Kontrakty w interfejsach i metodach abstrakcyjnych”).

Warunki początkowe powinny mieć dostęp jedynie do tych elementów składowych, które są dostępne przynajmniej na takim samym poziomie jak funkcja. Dzięki temu komponent wywołujący ma możliwość sensownego wykorzystania kontraktu. Jeżeli zachodzi potrzeba odczytu lub wywołania mniej dostępnych elementów składowych, wówczas prawdopodobnie weryfikujemy *wewnętrzny stan* zamiast egzekwować *wykonanie kontraktu*. W takim przypadku należy skorzystać z asercji.

Istnieje możliwość wywołania `Contract.Requires()` dowolną niezbędną liczbę razy na początku wykonywania metody, aby wyegzekwować wykonanie różnych warunków.

Co należy umieszczać w warunkach początkowych?

Poniżej przedstawiono udzielone przez zespół tworzący kontrakty kodu wskazówki dotyczące warunków początkowych:

- Możliwość łatwej weryfikacji do przeprowadzenia przez klienta (komponent wywołujący).
- Opieranie się na danych i funkcjach, które są przynajmniej tak samo dostępne jak metoda.
- W przypadku złamania kontraktu zawsze należy wskazać na istnienie *błędu*.

Konsekwencją ostatniego punktu jest to, że klient nigdy nie powinien celowo „przechwytywać" niepowodzenia wykonania kontraktu (tak naprawdę typ `ContractException` ma na celu pomoc w egzekwowaniu tej reguły). Zamiast tego klient powinien prawidłowo wywoływać obiekt docelowy. Jeżeli ta operacja zakończy się niepowodzeniem, wskazuje ono na błąd, który powinien zostać obsłużony za pomocą ogólnego przeznaczenia procedury obsługi wyjątków (co może oznaczać przerwanie działania aplikacji). Innymi słowy: jeżeli decyzje dotyczące kontroli przepływu działania aplikacji lub inne operacje są podejmowane na podstawie niepowodzenia warunku początkowego, w rzeczywistości nie potrzebujemy kontraktu, ponieważ aplikacja może kontynuować działanie nawet po niepowodzeniu kontraktu.

Prowadzi to do następujących wskazówek, kiedy należy się zdecydować na użycie warunków początkowych, a kiedy lepiej zgłaszać zwykłe wyjątki:

- Jeżeli niepowodzenie *zawsze* wskazuje na błąd po stronie klienta, lepszym rozwiązaniem jest warunek początkowy.
- Jeżeli niepowodzenie wskazuje na *nietypową sytuację*, która *może* oznaczać istnienie błędu po stronie klienta, wtedy najlepiej zgłosić (możliwy do przechwycenia) wyjątek.

Aby zilustrować omawiany przykład, załóżmy, że tworzymy funkcję `Int32.Parse()`. Rozsądne wydaje się przyjęcie założenia, że dane wejściowe w postaci wartości `null` wskazują na błąd po stronie komponentu wywołującego tę funkcję. Dlatego też decydujemy się na zastosowanie warunku początkowego:

```
public static int Parse (string s)
{
  Contract.Requires (s != null);
  ...
}
```

Następnie trzeba sprawdzić, czy przekazany ciąg tekstowy zawiera jedynie cyfry oraz symbole takie jak `+` i `-` (umieszczone w odpowiednich miejscach). Przeprowadzenie tego rodzaju weryfikacji po stronie komponentu wywołującego spowodowałoby jego niepotrzebne obciążenie. Dlatego też nie stosujemy tutaj warunku początkowego, ale ręczną operację sprawdzenia, która w przypadku niepowodzenia zgłasza (możliwy do przechwycenia) wyjątek `FormatException`.

Wystąpił problem związany z dostępnością elementu składowego. Spójrz na poniższy fragment kodu, który dość często pojawia się w typach implementujących interfejs `IDisposable`:

```
public void Foo()
{
  if (_isDisposed)  // _isDisposed to prywatny element składowy
    throw new ObjectDisposedException ("...");
  ...
}
```

Tego rodzaju operacja sprawdzania nie powinna mieć postaci warunku początkowego, jeżeli element składowy `_isDisposed` nie będzie dostępny dla komponentu wywołującego (np. przez przeprowadzenie jego refaktoryzacji na postać publicznie dostępnej właściwości).

Na koniec rozważmy metodę `File.ReadAllText()`. Poniższy fragment kodu pokazuje *nieprawidłowy* sposób użycia warunku początkowego:

```
public static string ReadAllText (string path)
{
  Contract.Requires (File.Exists (path));
  ...
}
```

Przed wywołaniem tej metody komponent wywołujący nie będzie miał pewności, czy plik istnieje (plik mógł zostać usunięty między operacją sprawdzenia a wywołaniem metody). Dlatego też lepiej jest tutaj zastosować tradycyjne podejście, czyli zgłoszenie możliwego do przechwycenia wyjątku `FileNotFoundException`.

Contract.Requires<TException>

Wprowadzenie kontraktów kodu stanowi wyzwanie dla przedstawionego poniżej podejścia głęboko zakorzenionego w .NET Framework, począwszy od wersji 1.0:

```
static void SetProgress (string message, int percent)  // podejście klasyczne
{
  if (message == null)
    throw new ArgumentNullException ("message");

  if (percent < 0 || percent > 100)
    throw new ArgumentOutOfRangeException ("percent");
  ...
}

static void SetProgress (string message, int percent)  // podejście nowoczesne
{
  Contract.Requires (message != null);
  Contract.Requires (percent >= 0 && percent <= 100);
  ...
}
```

Jeżeli mamy ogromny podzespół wymuszający klasyczne sprawdzanie argumentu, wówczas użycie warunków początkowych w nowo tworzonych metodach spowoduje powstanie niespójności w bibliotece. Pewne metody będą zgłaszały wyjątki dotyczące argumentów, podczas gdy inne będą zgłaszać wyjątek `ContractException`. Jednym z rozwiązań jest uaktualnienie wszystkich istniejących metod, aby wykorzystywały kontrakty kodu. Jednak takie podejście generuje dwa poniższe problemy:

- Uaktualnienie kodu to zadanie bardzo czasochłonne.
- Działanie komponentu wywołującego może *zależeć* od rodzaju zgłaszanego wyjątku, np. `ArgumentNullException`. (Ten wyjątek to niemal zawsze oznaka niewłaściwego przygotowania projektu).

Rozwiązaniem będzie wywołanie ogólnej wersji `Contract.Requires()`. W ten sposób można wskazać rodzaj wyjątku do zgłoszenia w przypadku niepowodzenia:

```
Contract.Requires<ArgumentNullException> (message != null, "message");
Contract.Requires<ArgumentOutOfRangeException>
 (percent >= 0 && percent <= 100, "percent");
```

(Drugi argument jest przekazywany konstruktorowi klasy wyjątku).

Wynikiem jest dokładnie takie samo zachowanie jak w przypadku tradycyjnego sprawdzania argumentu, choć jednocześnie otrzymujemy wszystkie korzyści wynikające ze stosowania kontraktów kodu (spójność, obsługa w interfejsach, niejawna dokumentacja, sprawdzanie statyczne i możliwość dostosowania do własnych potrzeb w trakcie działania aplikacji).

Wskazany wyjątek zostanie zgłoszony tylko wtedy, gdy podczas wywoływania binarnego rewritera podamy opcję `/throwonfailure` lub też *usuniemy* zaznaczenie pola wyboru *Assert on contract failure* na karcie *Code Contracts* w oknie właściwości projektu w Visual Studio. W przeciwnym razie nastąpi wyświetlenie okna dialogowego.

Istnieje również możliwość wskazania w binarnym rewriterze *ReleaseRequires* jako poziomu sprawdzania kontraktu (zob. podrozdział „Selektywne egzekwowanie kontraktów" w dalszej części rozdziału). Wywołania do ogólnej metody `Contract.Requires<TException>` pozostają na miejscu, podczas gdy wszystkie pozostałe są eliminowane. W takim przypadku podzespół będzie zachowywał się tak jak wcześniej.

Contract.EndContractBlock()

Metoda `Contract.EndContractBlock()` pozwala na połączenie zalet kontraktów kodu z tradycyjnym kodem sprawdzającym argumenty. W ten sposób unikamy konieczności refaktoryzacji kodu utworzonego dla platformy .NET Framework w wersji wcześniejszej niż 4.0. Nasze zadanie sprowadza się do wywołania wymienionej metody po przeprowadzeniu sprawdzenia argumentów:

```
static void Foo (string name)
{
  if (name == null) throw new ArgumentNullException ("name");
  Contract.EndContractBlock();
  ...
}
```

Binarny rewriter skonwertuje powyższy kod na postać podobną do przedstawionej poniżej:

```
static void Foo (string name)
{
  Contract.Requires<ArgumentNullException> (name != null, "name");
  ...
}
```

Kod poprzedzający `EndContractBlock()` musi się składać z prostych poleceń w następującej postaci:

```
if <warunek> throw <wyrażenie>;
```

Można łączyć tradycyjne sprawdzanie argumentów i wywołania kontraktów kodu. W tym celu wywołania kontraktów kodu powinny się znajdować za poleceniami tradycyjnego sprawdzania argumentów:

```
static void Foo (string name)
{
  if (name == null) throw new ArgumentNullException ("name");
  Contract.Requires (name.Length >= 2);
  ...
}
```

Wywołanie dowolnej z metod egzekwujących kontrakty kodu niejawnie powoduje zakończenie bloku kontraktu.

Kluczem jest zdefiniowanie na początku metody pewnego obszaru, aby binarny rewriter wiedział, że znajdujące się w nim polecenia `if` stanowią fragment kontraktu. Wywołanie dowolnej metody egzekwującej kontrakt powoduje niejawne rozszerzenie bloku kontraktu, więc nie musimy się przejmować wywołaniem `EndContractBlock()`, jeśli korzystamy z innej metody kontraktu, np. `Contract.Ensures()`.

Warunki początkowe i nadpisane metody

Podczas nadpisywania metody wirtualnej nie można dodawać warunków początkowych, ponieważ spowoduje to *zmianę kontraktu* (na skutek dalszego zaostrzania warunków) i złamanie zasad polimorfizmu.

(Technicznie rzecz biorąc, projektant mógłby otrzymać możliwość nadpisywania metod w celu *złagodzenia* warunków początkowych. Jednak nie zdecydowano się na taki krok, ponieważ potencjalne scenariusze użycia okazały się niewystarczające do uzasadnienia zwiększenia poziomu skomplikowania).

Binarny rewriter gwarantuje, że warunki początkowe metody bazowej zawsze będą egzekwowane w podklasach, niezależnie od tego, czy nadpisana metoda wywołuje metodę bazową.

Warunki końcowe

Do zdefiniowania warunku końcowego można wykorzystać metodę `Contract.Ensures()`.

Contract.Ensures()

Metoda `Contract.Ensures()` egzekwuje warunek końcowy, który musi przyjąć wartość `true` w chwili, gdy metoda kończy działanie. Przykład widzieliśmy już wcześniej:

```
static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  Contract.Requires (list != null);          // warunek początkowy
  Contract.Ensures (list.Contains (item));   // warunek końcowy
  if (list.Contains(item)) return false;
  list.Add (item);
  return true;
}
```

Binarny rewriter przenosi warunki końcowe do punktów wyjścia metody. Tego rodzaju warunki są sprawdzane, gdy działanie metody kończy się wcześniej (jak w omawianym przykładzie), ale już nie w przypadku jej zakończenia za pomocą nieobsłużonego wyjątku.

W przeciwieństwie do warunków początkowych, wykrywających ich nieprawidłowe użycie przez *komponent wywołujący*, warunki końcowe wykrywają błędy w samej funkcji (podobnie jak asercje). Dlatego też warunek końcowy może mieć dostęp do prywatnych informacji o stanie (na ten temat pisaliśmy w sekcji „Warunki początkowe i nadpisane metody”).

Warunki końcowe i zapewnienie bezpieczeństwa wątków

Scenariusze wielowątkowe (zob. rozdział 14.) stanowią wyzwanie dla użyteczności warunków końcowych. Na przykład przyjmujemy założenie o przygotowaniu zapewniającego bezpieczeństwo wątków opakowania dla `List<T>` w postaci przedstawionej poniżej metody:

```
public class ThreadSafeList<T>
{
  List<T> _list = new List<T>();
  object _locker = new object();

  public bool AddIfNotPresent (T item)
  {
    Contract.Ensures (_list.Contains (item));
    lock (_locker)
    {
      if (_list.Contains(item)) return false;
      _list.Add (item);
      return true;
    }
  }

  public void Remove (T item)
  {
    lock (_locker)
      _list.Remove (item);
  }
}
```

Warunek końcowy w metodzie `AddIfNotPresent()` jest sprawdzany *po* zwolnieniu blokady — na tym etapie element może już nie istnieć na liście, jeśli inny wątek wywołał dla niego metodę `Remove()`. Na razie nie istnieje rozwiązanie tego problemu, poza stosowaniem takich warunków jako asercji (zob. podrozdział „Asercje i metody inwariantów obiektu" w dalszej części rozdziału) zamiast warunków końcowych.

Contract.EnsuresOnThrow(TException)

Od czasu do czasu użyteczne jest zagwarantowanie, że gdy wskazany warunek będzie miał przypisaną wartość `true`, wtedy należy zgłosić wyjątek określonego rodzaju. Dokładnie na tym polega działanie metody `Contract.EnsuresOnThrow()`:

```
Contract.EnsuresOnThrow<WebException> (this.ErrorMessage != null);
```

Contract.Result<T> i Contract.ValueAtReturn<T>

Ponieważ warunki końcowe nie są oszacowane aż do zakończenia działania funkcji, rozsądne jest uzyskanie dostępu do wartości zwrotnej metody. Dokładnie na tym polega działanie metody `Contract.Result<T>`:

```
Random _random = new Random();
int GetOddRandomNumber()
{
  Contract.Ensures (Contract.Result<int>() % 2 == 1);
  return _random.Next (100) * 2 + 1;
}
```

Metoda `Contract.ValueAtReturn<T>` pełni tę samą funkcję, ale dla parametrów `ref` i `out`.

Contract.OldValue<T>

Metoda `Contract.OldValue<T>` zwraca wartość początkową parametru metody. Taka możliwość jest użyteczna w przypadku warunków końcowych, ponieważ są sprawdzane na *końcu* funkcji. Dlatego też dowolne wyrażenie w warunku końcowym zawierające parametry będzie odczytywało *zmodyfikowane* wartości parametrów.

Na przykład warunek końcowy w poniższej metodzie zawsze zakończy się niepowodzeniem:

```
static string Middle (string s)
{
  Contract.Requires (s != null && s.Length >= 2);
  Contract.Ensures (Contract.Result<string>().Length < s.Length);
  s = s.Substring (1, s.Length - 2);
  return s.Trim();
}
```

Oto poprawiona wersja poprzedniego fragmentu kodu:

```
static string Middle (string s)
{
  Contract.Requires (s != null && s.Length >= 2);
  Contract.Ensures (Contract.Result<string>().Length <
                    Contract.OldValue (s).Length);
  s = s.Substring (1, s.Length - 2);
  return s.Trim();
}
```

Warunki końcowe i nadpisane metody

Nadpisana metoda nie może obejść warunków końcowych zdefiniowanych w metodzie bazowej, ale może zdefiniować nowe. Binarny rewriter gwarantuje, że warunki końcowe w metodzie bazowej zawsze będą egzekwowane, nawet jeśli nadpisana metoda nie wywołuje tej implementacji bazowej.

Z wcześniej wspomnianych powodów warunki końcowe w metodach wirtualnych nie powinny mieć dostępu do prywatnych elementów składowych. W takim przypadku skutkiem będzie wplecenie przez binarny rewriter kodu w podklasie próbującego uzyskać dostęp do prywatnych elementów składowych w klasie bazowej, co doprowadzi do powstania błędu w trakcie działania aplikacji.

Asercje i metody inwariantów obiektu

Poza warunkami początkowymi i końcowymi API kontraktów kodu pozwala jeszcze na tworzenie asercji oraz definiowanie metod *inwariantów obiektu*.

Asercje

Do zdefiniowania asercji można wykorzystać metodę `Contract.Assert()`.

Contract.Assert()

Asercje można utworzyć w dowolnym miejscu funkcji. Wystarczy do tego wywołanie `Contract.Assert()`. Opcjonalnie definiujemy komunikat błędu wyświetlany w przypadku niepowodzenia asercji:

```
...
int x = 3;
...
Contract.Assert (x == 3);                    // niepowodzenie, jeżeli wartość x nie wynosi 3
Contract.Assert (x == 3, "Wartość x musi wynosić 3.");
...
```

Binarny rewriter nie przenosi asercji w kodzie. Istnieją dwa powody, dla których lepiej stosować metodę `Contract.Assert()` zamiast `Debug.Assert()`:

- Możliwość wykorzystania znacznie elastyczniejszego mechanizmu obsługi niepowodzenia oferowanego przez kontrakty kodu.
- Statyczne narzędzia sprawdzające mogą podjąć próbę weryfikacji `Contract.Assert()`.

Contract.Assume()

W trakcie działania aplikacji metoda `Contract.Assume()` zachowuje się dokładnie tak samo jak `Contract.Assert()`, ale wiążą się z nią nieco inne implikacje podczas użycia statycznych narzędzi sprawdzających. Ogólnie rzecz biorąc, statyczne narzędzia sprawdzające nie *kwestionują* założenia, podczas gdy mogą kwestionować asercje. Jest to rozwiązanie użyteczne, ponieważ zawsze istnieją kwestie niemożliwe do udowodnienia przez statyczne narzędzie sprawdzające, co może prowadzić do „podnoszenia fałszywego alarmu" dotyczącego prawidłowości asercji. Zmiana asercji na założenie pozwala uciszyć statyczne narzędzie sprawdzające.

Metody inwariantów obiektu

W przypadku klasy istnieje możliwość podania jednej metody lub więcej metod *inwariantów obiektu*. Tego rodzaju metody są wykonywane automatycznie po każdej funkcji *publicznej* w klasie i pozwalają na przygotowanie asercji sprawdzającej, czy wewnętrzny stan obiektu pozostaje spójny.

Obsługa wielu metod inwariantów obiektu została wprowadzona, aby zapewnić doskonałą współpracę z klasami częściowymi.

W celu zdefiniowania metody inwariantów obiektu należy utworzyć pozbawioną parametrów metodę typu `void` i oznaczyć ją atrybutem `[ContractInvariantMethod]`. Następnie w tej metodzie trzeba wywołać `Contract.Invariant()` w celu wyegzekwowania każdego warunku, który powinien przyjmować wartość `true`:

```
class Test
{
  int _x, _y;

  [ContractInvariantMethod]
  void ObjectInvariant()
  {
    Contract.Invariant (_x >= 0);
    Contract.Invariant (_y >= _x);
  }
```

```
  public int X { get { return _x; } set { _x = value; } }
  public void Test1() { _x = -3; }
  void Test2()        { _x = -3; }
}
```

Binarny rewriter konwertuje właściwość `X` oraz metody `Test1()` i `Test2()` na postać podobną do przedstawionej poniżej:

```
public void X { get { return _x; } set { _x = value; ObjectInvariant(); } }
public void Test1() { _x = -3; ObjectInvariant(); }
void Test2()        { _x = -3; }     // bez zmian, ponieważ jest to metoda prywatna
```

Metody inwariantów obiektu nie *chronią* obiektu przed przejściem do nieprawidłowego stanu, a jedynie *wykrywają* wystąpienie tego rodzaju sytuacji.

Wywołanie `Contract.Invariant()` przypomina `Contract.Assert()`, z wyjątkiem tego, że może pojawiać się jedynie w metodach oznaczonych atrybutem `[ContractInvariantMethod]`. I na odwrót: metoda inwariantów kontraktu może zawierać jedynie wywołania `Contract.Invariant()`.

Podklasa również może wprowadzać własną metodę inwariantów obiektu, która będzie sprawdzana obok metody inwariantu klasy bazowej. Jedyne zastrzeżenie to fakt, że to sprawdzenie będzie przeprowadzane tylko po wywołaniu metody publicznej.

Kontrakty w interfejsach i metodach abstrakcyjnych

Potężną funkcjonalnością oferowaną przez kontrakty kodu jest możliwość umieszczania warunków w elementach składowych interfejsów oraz w metodach abstrakcyjnych. Binarny rewriter następnie automatycznie wplecie te warunki do implementacji konkretnych elementów składowych.

Specjalny mechanizm pozwala na wskazanie oddzielnej klasy kontraktu przeznaczonej dla interfejsów i metod abstrakcyjnych. Dzięki temu można przygotować metody przeznaczone do przechowywania warunków kontraktów. Poniżej przedstawiono przykład tego rodzaju rozwiązania:

```
[ContractClass (typeof (ContractForITest))]
interface ITest
{
  int Process (string s);
}

[ContractClassFor (typeof (ITest))]
sealed class ContractForITest : ITest
{
  int ITest.Process (string s)      // konieczne jest wyraźne użycie implementacji
  {
    Contract.Requires (s != null);
    return 0;                       // przykładowa wartość w celu zadowolenia kompilatora
  }
}
```

Zwróć uwagę na konieczność zwrotu wartości podczas implementacji `ITest.Process()`, aby zadowolić kompilator. Jednak kod zwracający wartość `0` nie działa. Zamiast tego binarny rewriter wyodrębnia z tej metody warunek i wplata go do rzeczywistej implementacji `ITest.Process()`. Oznacza

to, że tak naprawdę nigdy nie zostanie utworzony egzemplarz klasy kontraktu (więc przygotowany konstruktor nie będzie mógł zostać wykonany).

Istnieje możliwość przypisania zmiennej tymczasowej wewnątrz bloku kontraktu, aby ułatwić odwołania do innych elementów składowych interfejsu. Na przykład jeśli interfejs `ITest` definiowałby także właściwość `Message` typu `string`, wówczas moglibyśmy przygotować poniższą implementację `ITest.Process()`:

```
int ITest.Process (string s)
{
  ITest test = this;
  Contract.Requires (s != test.Message);
  ...
}
```

Zastosowane w tej implementacji polecenie jest znacznie łatwiejsze niż poniższe:

```
Contract.Requires (s != ((ITest)this).Message);
```

(Użycie po prostu `this.Message` nie działa, ponieważ wymagana jest wyraźna implementacja `Message`). Proces definiowania klas kontraktów dla klas abstrakcyjnych przedstawia się dokładnie tak samo, z wyjątkiem faktu, że klasa kontraktu powinna być oznaczona jako abstrakcyjna (`abstract`) zamiast zapieczętowanej (`sealed`).

Rozwiązywanie problemów z awariami podczas użycia kontraktów

Za pomocą opcji `/throwonfailure` (lub pola wyboru *Assert on contract failure* na karcie *Code Contracts* w oknie właściwości projektu w Visual Studio) binarny rewriter pozwala na zdefiniowanie, co się stanie w przypadku niepowodzenia spełnienia warunku kontraktu.

Jeżeli nie użyjemy opcji `/throwonfailure` — lub pola wyboru *Assert on contract failure* — wówczas wspomniane niepowodzenie spowoduje wyświetlenie okna dialogowego, w którym zdecydujemy, co dalej: przerwanie pracy, debugowanie lub zignorowanie błędu.

Należy mieć świadomość istnienia kilku niuansów:

- Jeżeli środowisko uruchomieniowe CLR jest hostowane (np. w SQL Server lub Exchange), wówczas zamiast wyświetlenia okna dialogowego będziemy mieli do czynienia ze stosowaną przez host polityką eskalacji.
- W przeciwnym razie, jeżeli bieżący proces nie może wyświetlić użytkownikowi okna dialogowego, nastąpi wywołanie `Environment.FailFast()`.

Wspomniane okno dialogowe jest z kilku powodów użyteczne podczas debugowania aplikacji:

- Znacznie ułatwia diagnozowanie i debugowanie niepowodzeń kontraktów i nie ma w tym celu konieczności ponownego uruchamiania programu. Takie rozwiązanie działa niezależnie od tego, czy Visual Studio skonfigurowano do awarii przy pierwszej sposobności zgłoszenia wyjątku. W przeciwieństwie do wyjątku, niepowodzenie w wypełnieniu kontraktu niemal na pewno oznacza błąd w kodzie.

- Jesteśmy informowani o niepowodzeniu kontraktu, nawet jeśli komponent wywołujący znajduje się wyżej na stosie wywołań i „połyka" wyjątek w pokazany poniżej sposób:

```
try
{
  // wywołanie pewnej metody, której kontrakt kończy się niepowodzeniem
}
catch { }
```

Przedstawiony powyżej kod jest w większości sytuacji uznawany za antywzorzec, ponieważ *maskuje* awarię — w tym m.in. warunki, których autor się nigdy nie spodziewał.

Jeżeli użyjemy opcji `/throwonfailure` i usuniemy zaznaczenie pola wyboru *Assert on contract failure* na karcie *Code Contracts* w oknie właściwości projektu w Visual Studio, wówczas niepowodzenie kontraktu spowoduje zgłoszenie wyjątku `ContractException`. Takie rozwiązanie jest oczekiwane w kilku sytuacjach:

- Wersje finalne aplikacji — w takim przypadku chcemy przekazywać wyjątek w górę stosu, aby został potraktowany w taki sam sposób jak każdy inny nieoczekiwany wyjątek (prawdopodobnie obsłużony przez najwyższego poziomu procedurę obsługi, która zarejestruje informacje o błędzie lub zachęci użytkownika do jego zgłoszenia).
- Środowiska testów jednostkowych, gdzie przetwarzanie zarejestrowanych błędów jest zautomatyzowane.

Wyjątek `ContractException` nie może się pojawić w bloku `catch`, ponieważ ten typ nie jest publiczny. Nie istnieje też żaden powód, dla którego należałoby *specjalnie* przechwytywać wyjątek `ContractException`. Powinien zostać przechwycony tylko jako część ogólnej procedury obsługi wyjątków.

Zdarzenie ContractFailed

Kiedy kontrakt zakończy się niepowodzeniem, statyczne zdarzenie `ContractFailed` zostanie wywołane jeszcze przed podjęciem jakiejkolwiek innej akcji. Jeżeli obsłużymy to zdarzenie, wówczas możemy wykonywać zapytania do argumentów obiektu zdarzenia w celu otrzymania szczegółów dotyczących błędu. Ponadto możemy wywołać `SetHandled()`, aby uniknąć ponownego zgłoszenia wyjątku `ContractException` (lub wyświetlenia okna dialogowego).

Obsługa wymienionego zdarzenia jest szczególnie użyteczna, gdy została użyta opcja `/throwon` ↪`failure`, ponieważ zyskujemy możliwość zarejestrowania danych o *wszystkich* niepowodzeniach kontraktów, nawet jeśli kod znajdujący się wyżej na stosie wywołań przechwyci wyjątek, jak to zostało wcześniej przedstawione. Poniżej zaprezentowano przykład wykorzystujący zautomatyzowane testy jednostkowe:

```
Contract.ContractFailed += (sender, args) =>
{
  string failureMessage = args.FailureKind + ": " + args.Message;
  // rejestracja failureMessage za pomocą frameworka testów jednostkowych
  // ...
  args.SetUnwind();
};
```

Ta procedura obsługi rejestruje wszystkie niepowodzenia kontraktów, przy czym pozwala na normalne działanie wyjątku `ContractException` (lub też na wyświetlenie okna dialogowego z informacją o niepowodzeniu) po zakończeniu działania procedury obsługi zdarzenia. Zwróć uwagę na wywołanie `SetUnwind()` neutralizujące efekt wszelkich wywołań do `SetHandled()` z poziomu innych komponentów subskrybujących powiadomienia o tym zdarzeniu. Innymi słowy: mamy pewność zgłoszenia wyjątku `ContractException` (lub wyświetlenia okna dialogowego) po każdym wykonaniu wszystkich procedur obsługi zdarzeń.

Jeżeli zgłoszenie wyjątku nastąpi w omawianej procedurze obsługi zdarzenia, wszelkie pozostałe procedury obsługi zdarzenia mimo wszystko nadal będą wykonywane. Zgłoszony wyjątek wypełni właściwość `InnerException` ostatecznie zgłoszonego wyjątku `ContractException`.

Wyjątki wewnątrz warunków kontraktu

Jeżeli wyjątek zostanie zgłoszony wewnątrz samego warunku kontraktu, wówczas taki wyjątek będzie propagowany podobnie jak inne, niezależnie od tego, czy została użyta opcja `/throwonfailure`. Przedstawiona poniżej metoda zgłasza wyjątek `NullReferenceException` w przypadku wywołania jej z wartością `null` zamiast ciągu tekstowego:

```
string Test (string s)
{
  Contract.Requires (s.Length > 0);
  ...
}
```

Warunek początkowy jest w zasadzie nieprawidłowy i powinien mieć następującą postać:

```
Contract.Requires (!string.IsNullOrEmpty (s));
```

Selektywne egzekwowanie kontraktów

Binarny rewriter oferuje dwie opcje pozwalające na wyeliminowanie sprawdzania części lub wszystkich kontraktów: `/publicsurface` i `/level`. Można je kontrolować z poziomu Visual Studio za pomocą karty *Code Contracts* we właściwościach projektu. Opcja `/publicsurface` nakazuje rewriterowi sprawdzanie kontraktów jedynie publicznych elementów składowych. Natomiast opcja `/level` ma wymienione poniżej warianty:

None (poziom 0.)

Całkowita eliminacja sprawdzania kontraktów.

ReleaseRequires (poziom 1.)

Włączenie jedynie wywołań do ogólnej wersji `Contract.Requires<TException>`.

Preconditions (poziom 2.)

Włączenie wszystkich warunków początkowych (poziom 1. plus standardowe warunki początkowe).

Pre and Post (poziom 3.)

Włączenie sprawdzania poziomu 2. plus warunki końcowe.

Full (poziom 4.)

Włączenie sprawdzania poziomu 3. plus metody inwariantów obiektu oraz asercje (czyli wszystko).

W konfiguracji *Debug* zwykle włącza się pełne sprawdzanie wszystkich kontraktów.

Kontrakty w konfiguracji Release

W przypadku kompilowania aplikacji w konfiguracji *Release* najczęściej stosowane są dwie wymienione poniżej ogólne filozofie:

- postawienie na bezpieczeństwo i włączenie pełnego sprawdzania kontraktów;
- postawienie na wydajność i całkowite wyłączenie sprawdzania kontraktów.

Jednak jeżeli tworzymy bibliotekę przeznaczoną do publicznego użycia, wówczas podejście drugie okaże się problematyczne. Wyobraźmy sobie kompilację i dystrybucję biblioteki *L* w konfiguracji *Release* wraz z wyłączonym sprawdzaniem kontraktów. Następnie klient kompiluje projekt *C* w konfiguracji *Debug* zawierający odwołania do biblioteki *L*. W ten sposób podzespół *C* może nieprawidłowo wywoływać elementy składowe *L*, co nie spowoduje złamania kontraktu! W takiej sytuacji naprawdę będziemy oczekiwali wyegzekwowania fragmentów kontraktów *L* gwarantujących prawidłowe użycie *L*, innymi słowy: *warunków początkowych* w *publicznych* elementach składowych *L*.

Najprostszym rozwiązaniem jest włączenie sprawdzania `/publicsurface` w *L* wraz z poziomem *Preconditions* lub *ReleaseRequires*. W ten sposób mamy gwarancję, że istotne warunki początkowe będą egzekwowane dla dobra konsumentów, podczas gdy spadek wydajności będzie niewielki i związany jedynie ze wspomnianymi warunkami początkowymi.

W przypadkach skrajnych możemy nie chcieć zgodzić się nawet na tak niewielki spadek wydajności i wówczas należy zastosować bardziej zaawansowane podejście określane mianem *sprawdzenia w miejscu wywołania* (ang. *call-site checking*).

Sprawdzenie w miejscu wywołania

Sprawdzenie w miejscu wywołania powoduje przeniesienie weryfikacji warunku początkowego z metod *wywoływanych* do metod *wywołujących* (czyli miejsc wywoływania). W ten sposób otrzymujemy rozwiązanie omówionego powyżej problemu — umożliwiamy konsumentom biblioteki *L* przeprowadzanie sprawdzenia warunków początkowych *L* podczas kompilacji w konfiguracji *Debug*.

Aby włączyć sprawdzenie w miejscu wywołania, najpierw trzeba przygotować oddzielny *podzespół odwołań do kontraktów*, czyli podzespół uzupełniający zawierający po prostu warunki początkowe dla podzespołu, do którego będziemy się odwoływać. W tym celu można wykorzystać narzędzie *ccrefgen* działające z poziomu wiersza poleceń lub też przeprowadzić operację w Visual Studio, jak opisujemy poniżej:

1. W konfiguracji *Release* projektu *biblioteki* (*L*) należy przejść do karty *Code Contracts* we właściwościach projektu i wyłączyć sprawdzanie kontraktów w trakcie działania aplikacji oraz zaznaczyć opcję *Build a Contract Reference Assembly*. W ten sposób zapewniamy utworzenie wspomnianego wcześniej podzespołu uzupełniającego (wraz z rozszerzeniem *.contracts.dll*).

2. W konfiguracji *Release* podzespołów korzystających z biblioteki *L* należy całkowicie wyłączyć sprawdzanie kontraktów.
3. W konfiguracji *Debug* podzespołów korzystających z biblioteki *L* należy zaznaczyć opcję *Call-site Requires Checking*.

Krok 3. jest odpowiednikiem wywołania *ccrewrite* wraz z opcją `/callsiterequires`. Odczytuje warunki początkowe z podzespołu kontraktów i wplata je w miejscach wywołań w podzespołach korzystających z biblioteki.

Statyczne sprawdzenie kontraktu

Kontrakty kodu umożliwiają przeprowadzenie *statycznego sprawdzenia kontraktu*, dzięki czemu narzędzie analizujące warunki kontraktu odszukuje potencjalne błędy w programie, zanim zostanie on uruchomiony. Na przykład statyczne sprawdzenie przedstawionego poniżej kodu wygeneruje ostrzeżenie:

```
static void Main()
{
  string message = null;
  WriteLine (message);      // narzędzie sprawdzania statycznego wygeneruje ostrzeżenie
}

static void WriteLine (string s)
{
  Contract.Requires (s != null);
  Console.WriteLine (s);
}
```

Opracowane przez Microsoft narzędzie przeznaczone do statycznego sprawdzania kontraktów można uruchomić z poziomu wiersza poleceń (*cccheck*) lub przez włączenie odpowiedniej opcji we właściwościach projektu w Visual Studio.

Aby sprawdzenie statyczne mogło być przeprowadzone, konieczne może być dodanie do metod warunków początkowych i końcowych. Poniżej przedstawiamy przykład, który spowoduje wygenerowanie ostrzeżenia:

```
static void WriteLine (string s, bool b)
{
  if (b)
    WriteLine (s);      // ostrzeżenie: nieudowodnione spełnienie warunku
}

static void WriteLine (string s)
{
  Contract.Requires (s != null);
  Console.WriteLine (s);
}
```

Ponieważ wywołujemy metodę wymagającą, aby parametr nie miał wartości `null`, konieczne jest udowodnienie, że argument ma wartość inną niż `null`. W tym celu w pierwszej metodzie można dodać warunek początkowy, jak pokazano w poniższym fragmencie kodu:

```
static void WriteLine (string s, bool b)
{
  Contract.Requires (s != null);
  if (b)
    WriteLine (s);    // OK
}
```

Atrybut [ContractVerification]

Statyczne sprawdzanie kontraktów będzie łatwiejsze, jeśli zostanie rozpoczęte na początku cyklu życiowego projektu. W przeciwnym razie będziemy przytłoczeni ogromną liczbą ostrzeżeń.

Aby zastosować statyczne sprawdzanie kontraktów w istniejącej bazie kodu, pomocne może być użycie ich na początku względem wybranych fragmentów programu za pomocą atrybutu `[Contract ↪Verification]` (zdefiniowanego w przestrzeni nazw `System.Diagnostics.Contracts`). Ten atrybut może być zastosowany na poziomie podzespołu, typu lub elementu składowego. Jeżeli zastosujemy go na wielu poziomach, wykorzystany będzie ten na poziomie najbardziej szczegółowym. Dlatego też w celu włączenia statycznej weryfikacji kontraktów dla określonej klasy należy zacząć od wyłączenia weryfikacji na poziomie podzespołu, jak pokazano poniżej:

```
[assembly: ContractVerification (false)]
```

Następnie trzeba włączyć weryfikację dla wybranej klasy:

```
[ContractVerification (true)]
class Foo { ... }
```

Opcja Baseline

Inną taktyką podczas stosowania statycznej weryfikacji kontraktów w istniejącej bazie kodu jest uruchomienie narzędzia wraz z zaznaczoną opcją *Baseline* w Visual Studio. Wszystkie wygenerowane ostrzeżenia zostaną zapisane we wskazanym pliku XML. Kiedy następnym razem rozpoczniemy weryfikację statyczną, wszystkie ostrzeżenia istniejące w tym pliku będą zignorowane — otrzymamy jedynie komunikaty wygenerowane dla *nowego* kodu w projekcie.

Atrybut [SupressMessage]

Narzędziu przeznaczonemu do statycznego sprawdzania kontraktów można nakazać zignorowanie określonego rodzaju ostrzeżeń, co wymaga użycia atrybutu `[SuppressMessage]` (zdefiniowanego w przestrzeni nazw `System.Diagnostics.CodeAnalysis`):

```
[SuppressMessage ("Microsoft.Contracts", rodzajOstrzeżenia)]
```

Wartość *`rodzajOstrzeżenia`* należy zastąpić jedną z poniższych:

```
Requires Ensures Invariant NonNull DivByZero MinValueNegation
ArrayCreation ArrayLowerBound ArrayUpperBound
```

Wymieniony atrybut można zastosować na poziomie podzespołu lub typu.