Olaf Spinczyk
Daniel Lohmann
Matthias Urban

AspectC++: aspektowe rozszerzenie C++

Na CD:

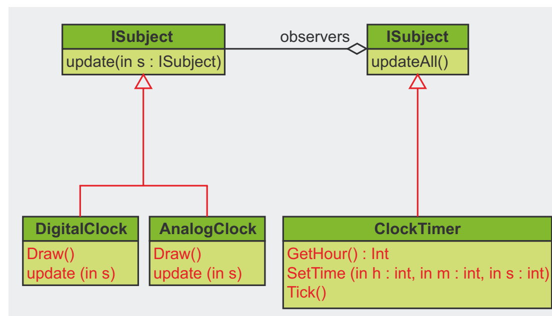
Na płycie dołączonej do numeru znajduje się wersja próbna dodatku AspectC++ dla Visual Studio .NET, darmowy zestaw narzędzi AspectC++ Development Tools (ACDT) dla Eclipse, a także listingi przedstawione w tym artykule.

Coraz więcej programistów poznaje programowanie aspektowe (AOP, od ang. *Aspect-Oriented Programming*) – metodologię umożliwiającą modularyzację implementacji problemów przekrojowych, ułatwiającą ponowne wykorzystywanie kodu, zapewniającą słabsze sprzężenie między modułami i lepsze rozdzielenie problemów. AOP posiada obecnie solidną bazę narzędziową w postaci produktów JBossa (JBoss AOP), BEA (AspectWerkz) czy IBM-a (AspectJ i AJDT dla Eclipse). Jednak wszystkie te produkty bazują na Javie. Z punktu widzenia programistów C/C++, żaden z głównych graczy na rynku nie oferuje jak dotąd obsługi programowania aspektowego.

W tym artykule poznamy AspectC++ – aspektowe rozszerzenie języka C++. Kompilator przekształcający kod źródłowy AspectC++ do postaci zwykłego kodu C++ jest rozwijany jako projekt open source i w porównaniu ze stworzonym w 2001 roku prototypem badawczym jest już projektem dojrzałym. Obecnie język i parser AspectC++ mają już za sobą udane występy w wielu prawdziwych projektach komercyjnych i akademickich, a dzięki jego integracji ze środowiskami programistycznymi Eclipse i Microsoft Visual Studio .NET, pierwsze aspektowe kroki stają się dziecinną igraszką.

Nasze wprowadzenie do AspectC++ rozpoczniemy od przykładu, który traktować można jako *Hello World* programowania aspektowego. Ilustruje on podstawowe elementy języka takie jak aspekty, punkty przekroju (ang. *pointcuts*) oraz rady (ang. *advice*), które część z czytelników zna już być może z języka AspectJ. Następnie szybko wyjdziemy poza konstrukcje typowe dla AspectJ i przyjrzymy się popularnemu wzorcowi Obserwator w wersji AspectC++, jak również radom ogólnym (ang. *Generic Advice*) – unikalnemu elementowi AspectC++, łączącemu potęgę aspektów z generycznym i generatywnym programowaniem w C++.

Dr Olaf Spinczyk jest założycielem i prowadzącym projektu AspectC++. Daniel Lohmann korzysta z AspectC++ w swoich badaniach nad rozwojem systemów operacyjnych oraz zajmuje się projektowaniem i rozwijaniem koncepcji języka AspectC++. Obaj pracują na Friedrich-Alexander University Erlangen-Nuremberg. Matthias Urban jest głównym deweloperem parsera AspectC++. Pracuje dla pure-systems GmbH, gdzie odpowiada za obsługę AspectC++ oraz rozwój dodatku AspectC++ dla VisualStudio.NET.
Kontakt z autorami: *os@aspectc.org*, *dl@aspectc.org*, *mu@aspectc.org*



Rysunek 1. *Przekroje we wzorcu Obserwator*

Tracing – aspektowy Hello World

Wprowadzenie do programowania aspektowego w AspectC++ zaczniemy od bardzo prostego aspektu o nazwie *Tracing*, czyli *śledzenie*, umożliwiającego ujednolicenie implementacji operacji wyjścia używanych do śledzenia wykonania programu. Aspekt ten wyświetla nazwę każdej wykonywanej funkcji:

```
#include <cstdio>
// Przykład śledzenia wykonania programu
aspect Tracing {
   // wyświetl nazwę funkcji przed jej wykonaniem
   advice execution ("% ...::%(...)") : before () {
      std::printf ("in %s\n", JoinPoint::signature ());
   }
};
```

Nawet bez dokładnego zrozumienia składni wszystkich pokazanych elementów, można od razu dostrzec kilka ogromnych zalet programowania aspektowego. Przedstawiony aspekt ma tylko kilka linijek, a oszczędza programiście ręcznego dodawania polecenia `printf` w kodzie każdej funkcji w całym programie. W dużym projekcie wymagałoby to umieszczenia takiego wymagania w poradniku stylu programowania, który wszyscy programiści musieliby przeczytać i którego musieliby przestrzegać. Aspektowe rozwiązanie problemu pozwala zaoszczędzić mnóstwo czasu i wysiłku organizacyjnego oraz gwarantuje, że żadna funkcja nie zostanie pominięta. Jednocześnie kod podlegający działaniu aspektu jest całkowicie oddzielony od właściwego kodu śledzącego, czyli polecenia `printf`. Nie trzeba nawet dołączać `<cstdio>`, gdyż tym zajmuje się sam aspekt.

Aspekty, rady i punkty przekroju

Przykład Tracing ilustruje większość spośród elementów aspektowych implementujących wspomniane zalety. Na początek przyjrzyjmy się samemu aspektowi, pełniącemu funkcję modułu implementującego problem przekrojowy. Z punktu widzenia składni aspekt

w AspectC++ (deklarowany słowem kluczowym `aspect`) bardzo przypomina klasę C++, ale oprócz funkcji i danych składowych aspekt może dodatkowo zawierać definicję rady. Podawane po słowie kluczowym `advice` wyrażenie punktu przekroju określa, w których miejscach dana rada ma wpływać na program – są to punkty złączenia. Po dwukropku podajemy, jakie konkretnie ma być działanie programu w tych punktach. Jest to ogólna zasada dotycząca formułowania rad w AspectC++.

Wyrażenia punktów przekroju

Wyrażeniem punktu przekroju w podanym wcześniej przykładzie jest `execution("% ...::%(...)")`, co oznacza, że rada ma wpływać na wykonanie wszystkich funkcji pasujących do wzorca `"% ...::%(...)"`. W takich wyrażeniach regularnych, znaki `%` i `...` są symbolami wieloznacznymi. Znak `%` pasuje do każdego typu (na przykład `"% *"` pasuje do każdego typu wskaźnikowego) oraz do dowolnego ciągu znaków w identyfikatorach (na przykład `"xdr_%"` pasuje do wszystkich klas o nazwach zaczynających się od `xdr_`). Wielokropek (`...`) pasuje do dowolnej sekwencji typów lub przestrzeni nazw (na przykład `"int foo(...)"` pasuje do każdej globalnej funkcji o nazwie `foo` zwracającej wartość typu `int`). Tym samym wykorzystane w przykładzie wyrażenie `"% ...::%(...)"` pasuje do każdej funkcji z każdej klasy lub przestrzeni nazw.

Wyrażenia regularne odpowiadają zbiorom konkretnych funkcji lub klas programu, więc już same w sobie są prostymi opisami punktów przekroju, wskazującymi zbiory punktów złączenia w statycznej strukturze programu (tzw. statyczne punkty złączeń). W naszym przykładzie chcemy jednak przypisać radę zdarzeniu wykonania funkcji, związanemu z dynamicznym przebiegiem programu. Stąd też musimy skorzystać z funkcji punktu przekroju o nazwie `execution()`, zwracającej punkty złączeń dla wykonania wszystkich funkcji przekazanych jako jej argumenty.

Rady dla dynamicznych punktów złączeń

W przypadku dynamicznych punktów złączeń, AspectC++ oferuje trzy typy rad: `before()` (przed), `after()` (po) i `around()` (wokół). Implementują one kolejny element zachowania programu. W naszym przykładowym aspekcie Tracing, zachowanie to jest implementowane przez wyrażenie `printf()`, poprzedzone radą `before()`. Z punktu widzenia składni przypomina to ciało funkcji – i nieprzypadkowo, gdyż treść rady można faktycznie traktować jako anonimową funkcję składową aspektu. W naszym przykładzie moglibyśmy zamiast `before()` użyć rady `after()` – wtedy treść rady zostałaby wykonana po zakończeniu wykonywania funkcji. Można też użyć i jednej, i drugiej. Treść rady `around()` jest wykonywana zamiast fragmentu kodu programu, który w normalnych warunkach znajdowałby się za dynamicznym punktem złączenia.

Łączenie wyrażeń punktów przekroju

Wyrażenia punktów przekroju można łączyć za pomocą operacji na zbiorach: `&&` (część wspólna), `||` (suma) i `!` (odwrotność). Na przykład wyrażenie `"% foo(int, ...)" || "int bar(...)"` pasuje do każdej globalnej funkcji o nazwie `foo`, której pierwszym parametrem jest `int`, a także do dowolnej globalnej funkcji o nazwie `bar` zwracającej `int`. Łącząc te operacje z funkcjami punktów przekroju, otrzymujemy możliwość tworzenia złożonych wyrażeń określających miejsca wpływu rady na program. Moglibyśmy na przykład w następujący sposób zmodyfikować wyrażenie punktu przekroju w naszym aspekcie Tracing:

```
advice call ("% ...::%(...)")
       && within ("Client") : before () {
  std::printf ("calling %s\n", JoinPoint::signature ());
}
```

Funkcja `call()` zwraca punkty złączeń dla wywołania wszystkich funkcji przekazanych jako jej argumenty. W przeciwieństwie do punktów wykonania, punkty złączeń wywołania znajdują się po stronie wywołującego, czyli przed, po lub zamiast samego wywołania funkcji. Funkcja punktu przekroju `within()` zwraca po prostu wszystkie punkty przekroju dla zadanych klas lub funkcji. Rada ma dotyczyć części wspólnej `call ("% ...::%(...)")` (wywołania dowolnej funkcji) i `within ("Client")` (wszystkich punktów złączenia w klasie `Client`), więc w efekcie aspekt będzie śledzić wyłącznie wywołania funkcji wykonane w metodach klasy `Client`.

API punktów złączenia

Pozostaje jeszcze wyjaśnić wyrażenie `JoinPoint::signature()` w treści rady. Z przykładu widać, że zwraca ono nazwę funk-

Listing 1. *Abstrakcyjny aspekt ObserverPattern*

```
aspect ObserverPattern {
   // Struktury danych do zarządzania podmiotami
   // i obserwatorami
   ...
public:
   // Interfejsy dla obu ról
   struct ISubject {};
   struct IObserver {
      virtual void update(ISubject *) = 0;
   };
   // Do zdefiniowania przez konkretny aspekt pochodny
   // subjectChange() pasuje do wszystkich
   // metody innych niż const
   pointcut virtual observers() = 0;
   pointcut virtual subjects() = 0;
   pointcut virtual subjectChange() =
      execution("% ...::%(...)" && !"% ...::%(...) const")
         && within(subjects());
   // Dodaj nową klasę bazową do każdej klasy podmiotu
   // lub obserwatora i wstaw kod
   // do informowania obserwatorów
   advice observers() : baseclass(IObserver);
   advice subjects() : baseclass(ISubject);
      advice subjectChange() : after() {
      ISubject* subject = tjp->that();
   updateObservers(subject);
   }
   // Operacje dodawania, usuwania
   // i powiadamiania obserwatorów
   void updateObservers(ISubject* sub) { ... }
   void addObserver(ISubject* sub, IObserver* ob) { ... }
   void remObserver(ISubject* sub, IObserver* ob) { ... }
};
```

Listing 2. *Implementacja konkretnego obserwatora*

```
#include "ObserverPattern.ah"
#include "ClockTimer.h"
aspect ClockObserver : public ObserverPattern {
   // zdefiniuj punkty przekroju
   pointcut subjects()   = "ClockTimer";
   pointcut observers()  = "DigitalClock"||"AnalogClock";
public:
   advice observers() :
         void update( ObserverPattern::ISubject* sub ) {
      Draw(); }
};
```

cji, wypisywaną przed rozpoczęciem wykonania danej funkcji. Statyczna funkcja składowa `signature()` jest zdefiniowana w ramach API punktów złączeń. Jest to API AspectC++ pozwalające kodowi aspektu pozyskiwać bieżące informacje z punktu złączenia, dla którego jest on wywoływany (lub tego punktu dotyczące). Jak się później przekonamy, tego rodzaju informacje są niezbędną częścią aspektów w rzeczywistych zastosowaniach.

Wnioski z przykładu Tracing

Przedstawiony wcześniej przykładowy aspekt `Tracing` mieści się w kilku liniach kodu, a mimo to wprowadza wiele kluczowych koncepcji AspectC++. Podsumujmy je:

- *Problem przekrojowy*: zagadnienie implementacyjne dotykające wielu różnych części programu.
- *Aspekt*: umożliwia modularną implementację problemu przekrojowego poprzez podawanie rad.
- *Punkt złączenia*: wydarzenie podczas wykonania programu (dynamiczny punkt złączenia) lub element jego statycznej struktury (statyczny punkt złączenia), przy którym stosowana jest rada.
- *Punkty przekroju*: zbiór punktów złączenia.
- *Wyrażenie regularne*: wzorzec, z którym porównywane są sygnatury wszystkich nazwanych elementów, składających się na strukturę statyczną. Wyrażenia regularne stanowią w efekcie prymitywne wyrażenia punktów przekroju, wskazujące statyczne punkty złączenia.
- *Wyrażenie punktu przekroju*: definiuje punkt przekroju. Wyrażenie takie składa się z wyrażeń regularnych i funkcji punktów przekroju, które wspólnie wskazują, w którym miejscu rada powinna wpłynąć na program.
- *Rada*: definiuje wpływ aspektu na program w danym punkcie przekroju. W przypadku dynamicznych punktów łączeń, można skorzystać z rad `before()`, `after()` lub `around()`, implementujących dodatkowe zachowania.
- *API punktów złączenia*: pozwala kodowi rady na pobieranie informacji o kontekście z bieżącego punktu złączenia poprzez wbudowany wskaźnik `JoinPoint *tjp`.

Kolejny etap

`Tracing` to typowy *aspekt programistyczny*. W przeciwieństwie do *aspektów produkcyjnych*, aspekty tego rodzaju używane są tylko podczas tworzenia programu, np. na potrzeby debugowania, zapewniania jakości czy optymalizacji. Z kolei aspekty produkcyjne stanowią część końcowej wersji oprogramowania, która dostarczana jest użytkownikom. Zalecamy zatem rozpoczęcie przygody z programowaniem aspektowym od aspektów programistycznych, by nabrać niezbędnej wprawy. Jednak w tym artykule nie zamierzamy oczywiście zatrzymać się na aspektowym odpowiedniku *Hello World*! Naszym kolejnym przykładem będzie aspekt produkcyjny, pokazujący bardziej zaawansowane możliwości AspectC++, a w szczególności przekroje w statycznej strukturze programu.

Wzorzec Obserwator w AspectC++

Powszechnie przyjętą praktyką przy obiektowym tworzeniu oprogramowania jest obecnie stosowanie klasycznych wzorców projektowych autorstwa „Bandy Czterech" (Gamma, Helm, Johnson, Vlissides). Jednym z najpopularniejszych wzorców jest *Observer*, czyli *Obserwator*, przedstawiony na Rysunku 1. Wzorzec ten można stosować wszędzie, gdzie pewien obiekt zarządza stanem (*Podmiot* – tu obiekt `ClockTimer`), zaś dowolna liczba innych obiektów (*Obserwatorów* – tu instancje `DigitalClock` i `AnalogClock`) ma być informowana o zmianach tego stanu. Jak widać na diagramie klas, relacja podmiot-obserwator pomiędzy naszymi trzema klasami w aplikacji może zostać wyrażona poprzez dziedziczenie `ClockTimer` z klasy wielokrotnego użytku o nazwie `ISubject`, zarządzającej listą obiektów-obserwatorów, oraz dziedziczenie obserwatorów z abstrakcyjnej klasy `IObserver`. Ponadto wszystkie funkcje zmieniające stan (`SetTime()` i `Tick()`) muszą zostać rozszerzone o wywołanie `updateAll()`, by wszyscy obserwatorzy zostali poinformowani o zmianie. Po stronie obserwatorów, `DigitalClock` i `AnalogClock` muszą być rozszerzone o funkcję `update()`, wymaganą przez ich abstrakcyjną klasę bazową. Kod wszystkich trzech klas wymaga zatem sporej ilości podatnych na błędy modyfikacji. Na Rysunku 1. czerwonym kolorem zaznaczono części implementacji, na które zmiany będą miały wpływ. Ujmując rzecz aspektowo, problem implementacji protokołu obserwatora statycznie i dynamicznie przecina związane z nim klasy `ClockTimer`, `DigitalClock` i `AnalogClock`, więc znacznie wygodniej będzie zaimplementować zmiany w postaci aspektu.

Przekroje dynamiczne

Znamy już wszystkie elementy języka AspectC++ niezbędne do zaimplementowania w tym przykładzie przekroju dynamicznego. Protokół obserwatora wymaga, by wszystkie metody klasy podmiotu zmieniające stan wywoływały `updateAll()` przed zakończeniem działania. W C++ jako metody zmieniające stan można potraktować wszystkie funkcje składowe niezadeklarowane jako `const`. Poniższa definicja rady wstawia do naszej klasy `ClockTimer` niezbędne wywołania funkcji `updateAll()`:

```
advice execution("% ClockTimer::%(...)") &&
      !execution("% ClockTimer::%(...) const") : after () {
   updateAll ();
}
```

Wyrażenie punktu przekroju tej rady można odczytać tak: punkt łączeń jest elementem ostatecznego punktu przekroju, jeżeli jest on wykonaniem funkcji składowej klasy `ClockTimer`, ale *nie jest* wykonaniem funkcji składowej klasy `ClockTimer` zadeklarowanej jako `const`. Dodatkowe zastrzeżenie jest konieczne, gdyż w tym przypadku obecność `const` w wyrażeniu regularnym jest traktowana jak ograniczenie. Bez dodatkowego zastrzeżenia wyrażenie pasowałoby zarówno do funkcji `const`, jak i pozostałych.

Wprowadzenia – implementacja statycznego przecinania

Statyczny przekrój można w tym przykładzie zaimplementować za pomocą wprowadzeń. Wprowadzenia są szczególnym rodzajem rady AspectC++, której lokalizację stanowi wyrażenie punktu przekroju reprezentujące pewien zbiór klas, a zawartością jest deklaracja wprowadzana do tych klas. Na przykład funkcję `update()` można wprowadzić do klasy obserwatora w następujący sposób:

```
advice "DigitalClock"||"AnalogClock" : void update() {
      Draw(); }
```

Należy zauważyć, że wprowadzone w ten sposób składowe są widoczne nie tylko dla aspektu. Funkcja `update()` mogłaby np. zostać wywołana przez inną składową `DigitalClock` lub `AnalogClock`, zupełnie jakby była zwykłą funkcją składową. Wprowadzenia nie są jednak ograniczone do funkcji składowych – można je wykorzystywać do wprowadzania składowych zawierających dane, klas wewnętrznych lub dowolnej innej konstrukcji składniowo dopuszczalnej w ramach definicji klasy.

Szczególnym rodzajem wprowadzeń są wprowadzenia klas bazowych, pozwalające dodawać nowe klasy do listy klas bazowych. W naszym przykładzie będą one bardzo przydatne, gdyż podmiot i obserwatorzy mają dziedziczyć odpowiednio z ról `ISubject` i `IObserver`:

```
advice "DigitalClock"||"AnalogClock" : baseclass(IObserver);
advice "ClockTimer" : baseclass(ISubject);
```

Wirtualne punkty przekroju i aspekty abstrakcyjne

Teraz mamy już wszystkie elementy niezbędne do stworzenia aspektu `ObserverPattern` na potrzeby naszego przykładu. Dodawanie protokołu obserwatora do zestawu klas jest jednak częstym zadaniem, więc przydałoby się stworzyć implementację nadającą się do wielokrotnego wykorzystania. Do tego celu przydadzą nam się dwa kolejne elementy języka AspectC++.

Pierwszym z nich jest możliwość nadawania nazw punktom przekroju. Na przykład wyrażenie `"DigitalClock" || "AnalogClock"`, z którego korzystaliśmy już kilkakrotnie, może stać się nazwanym punktem przekroju o nazwie `observers()`:

```
pointcut observers() = "DigitalClock"||"AnalogClock";
```

Jeszcze ciekawszą właściwością nazwanych punktów przekroju jest możliwość deklarowania ich jako wirtualnych (`virtual`) lub czysto wirtualnych (`pure virtual`). Czysto wirtualne punkty przekroju mogą być wykorzystywane jako rady dla zwykłych punktów przekroju. Aspekt korzystający z czysto wirtualnych punktów przekroju definiuje jedynie sposób implementacji problemu przekrojowego, ale nie konkretne miejsce jego oddziaływania na program:

```
pointcut virtual observers() = 0;
pointcut virtual subjects() = 0;
advice observers() : baseclass(IObserver);
advice subjects() : baseclass(ISubject);
```

Oznacza to, że taki aspekt jest niekompletny – jest aspektem abstrakcyjnym. Konstrukcja ta bardzo przypomina mechanizm klas abstrakcyjnych zawierających czysto wirtualne funkcje składowe, co uniemożliwia tworzenie ich instancji. Abstrakcyjne aspekty nie mają wpływu na program, dopóki nie jest dostępny aspekt pochodny dostarczający definicję czysto wirtualnego punktu przekroju swojego aspektu bazowego.

Dziedziczenie aspektów

Po zebraniu wszystkiego razem otrzymujemy aspekt wielokrotnego zastosowania o nazwie `ObserverPattern`, przedstawiony na Listingu 1. Jest on całkowicie niezależny od trzech klas występujących w przykładzie, gdyż definiuje jedynie sposób, w jaki wzorzec obserwatora przecina implementację, ale nie określa konkretnego miejsca. Faktyczny punkt przekroju wskazuje aspekt pochodny `ClockObserver`, przedstawiony na Listingu 2. Dostarcza on definicje dwóch odziedziczonych, czysto wirtualnych punktów przekroju, a także implementuje wprowadzoną funkcję `update()` w sposób specyficzny dla danego obserwatora.

Dalsze wnioski

Ostatecznie stworzyliśmy dwa aspekty. Abstrakcyjny aspekt bazowy wielokrotnego zastosowania o nazwie `ObserverPattern` hermetyzuje implementację protokołu obserwatora. Jest to ogromna zaleta, gdyż wprowadzenie tego założenia projektowego w sposób tradycyjny mogłoby wymagać ręcznych zmian w dziesiątkach klas. Modyfikując sam aspekt moglibyśmy na przykład łatwo przełączać się między implementacją składującą listę obserwatorów w każdej instancji podmiotu a implementacją wykorzystującą w tym celu wspólną strukturę danych.

Aspekt `ClockObserver` dziedziczy po `ObserverPattern`, stanowiąc w ten sposób połączenie między abstrakcyjnymi rolami obserwatora i podmiotu a ich konkretnymi implementacjami w postaci klas `ClockTimer`, `DigitalClock` i `AnalogClock`. Jest to kolejna zaleta zastosowania aspektu, gdyż nie ma już potrzeby zaśmiecania samych klas aplikacji kodem specyficznym dla wzorca. Kod aplikacji pozostaje nietknięty, przez co zachowuje czytelność i uniwersalność.

Przykład `ObserverPattern` pozwolił wprowadzić kilka nowych, ważnych pojęć AspectC++. Pora na kolejne podsumowanie:

- *Wprowadzenie*: rada dla statycznych punktów złączenia. Wprowadzenia mogą służyć rozszerzeniu klasycznej struktury klas o dodatkowe elementy – metody, dane składowe lub klasy lokalne.
- *Wprowadzenie klasy bazowej*: szczególna forma wprowadzenia, rozszerzająca listę klas bazowych, po których dana klasa dziedziczy.
- *Nazwany punkt przekroju*: wyrażenie punktu przekroju dostępne poprzez identyfikator. Nazwane punkty przekroju można deklarować jako wirtualne lub czysto wirtualne, co pozwala je przedefiniować w aspekcie pochodnym.
- *Aspekt abstrakcyjny*: aspekt zawierający przynajmniej jeden czysto wirtualny punkt przekroju lub metodę. Aspekty abstrakcyjne są niekompletne, w efekcie czego nie wpływają one na program do momentu dopełnienia ich przez aspekt pochodny.
- *Dziedziczenie aspektów:* podobnie jak klasy, aspekty mogą dziedziczyć po innych aspektach.

Inne zastosowania

Zastosowania aspektów nie są ograniczone do śledzenia czy wzorców. Problemy przekrojowe są niemal wszędzie, a po opanowaniu podstaw programowania aspektowego programiści szybko zaczynają je identyfikować i szukać narzędziami umożliwiających ich szybką i modularną implementację. Przyjrzymy się teraz ostatniemu już przykładowi, ilustrującemu kolejny aspekt produkcyjny. Z technicznego punktu widzenia pozwoli on zobaczyć, jakie wpłyną kożyści z implementacji aspektów, bazujących na informacjach dostarczanych przez API punktów złączenia, w połączeniu z technikami programowania generycznego i generatywnego.

Aspekty w zaawansowanym C++

Programiści C++ często muszą pracować z istniejącymi bibliotekami C w rodzaju API Win32. Pomijając już fakt, że API to nie jest obiektowe, obsługa błędów jego funkcji bibliotecznych kompletnie nie pasuje do powszechnie obecnie stosowanego modelu bazującego na wyjątkach. Ręczne przekształcanie obsługi błędów w stylu C na obsługę za pomocą wyjątków byłoby zadaniem bardzo trudnym i podatnym na błędy. Jest to również problem przekrojowy, gdyż konieczne byłoby opakowanie każdej funkcji API Win32 w funkcję sprawdzającą wynik operacji i zgłaszającą wyjątek w razie wykrycia błędu.

Aspekt `ThrowWin32Errors` przedstawiony na Listingu 3. robi dokładnie to samo, ale wymaga od programisty znacznie mniej pracy. Po skompilowaniu aplikacji z tym aspektem API Win32 zaczyna informować o błędach przez zgłaszanie wyjątków. Implementacja tego aspektu jest już jednak nieco bardziej skomplikowana od poprzednich przykładów, więc omówimy ją szczegółowo.

Wykrywanie błędów Win32

Pierwszym krokiem ku zaimplementowaniu propagacji błędów opartej na wyjątkach jest sprawdzenie, czy wywołanie funkcji Win32 zakończyło się błędem. Funkcje API Win32 informują o sytuacji wystąpienia błędu, zwracając specjalną wartość magiczną (ang. *magic value*), zatem wykrywanie nieudanych odwołań do API jest kolejnym dynamicznym problemem przekrojowym. Jego rozwiązanie polega na dodaniu rad po wszystkich wywołaniach funkcji Win32. Wartość zwrócona przez funkcję jest sprawdzana wewnątrz rady i w razie stwierdzenia błędu zgłaszany jest wyjątek:

```
aspect ThrowWin32Errors
   {
   advice call(win32::Win32API()) : after() {
      if(<wartość_magiczna> == *tjp->result())  throw ...
   }
};
```

Tak sformułowana rada dotyczy wszystkich funkcji API opisanych przez zewnętrznie zdefiniowany punkt przekroju `win32::Win32API()`, zawierający wszystkie funkcje API Win32 (patrz Listing 4.). Wartość zwróconą przez wywołaną funkcję Win32 odczytujemy w treści rady za pomocą metody API punktów złączenia o nazwie `tjp->result()`. Zwraca ona wskaźnik do faktycznego wyniku, co w razie potrzeby pozwala nawet na jego modyfikację, ale w tym przypadku wystarczy nam porównanie go z wartością magiczną zwracaną przez API w celu zasygnalizowania błędu.

Tabela 1. *Fragment API punktów złączenia AspectC++*

Typy i wyliczenia czasu kompilacji	
`That`	typ danej klasy
`Target`	typ klasy docelowej (dla punktów złączenia wywołania)
`Arg<i>::Type` `Arg<i>::ReferredType`	typ i-tego argumentu, gdzie 0 ≤ i < ARGS
`Result`	Typ wartości wynikowej
`ARGS`	liczba argumentów
Statyczne metody czasu uruchomienia	
`const char *signature()`	sygnatura danej funkcji
Niestatyczne metody czasu uruchomienia	
`void proceed()`	wykonaj oryginalny kod (rada `around()`)
`That *that()`	instancja obiektu spod referencji `this`
`Target *target()`	instancja docelowego obiektu wywołania (dla punktów złączenia wywołania)
`Arg<i>::ReferredType *arg()`	instancja wartości i-tego argumentu
`Result *result()`	instancja wartości wynikowej

Jednak tak sformułowana definicja aspektu jeszcze nie działa. Problem polega na tym, że porównywana z wynikiem wartość magiczna nie zawsze jest taka sama, gdyż zależy ona od typu wywołanej funkcji API. Wiele funkcji Win32 jest typu `BOOL` i informuje o wystąpieniu błędu, zwracając wartość `FALSE`. Inne funkcje API korzystają z innych typów, na przykład `HWND`, `ATOM`, `HDC` lub `HANDLE`. Każdemu z tych typów odpowiada pewna wartość magiczna, zgłaszana w razie wystąpienia błędu – dla funkcji typu `ATOM` jest nią na przykład `0`, ale już dla funkcji typu `HANDLE` może to być `NULL` lub `INVALID_HANDLE_VALUE`.

Rada generyczna

Moglibyśmy spróbować rozwiązać ten problem, filtrując funkcje z punktu przekroju `win32::Win32API()` i dla każdego typu funkcji zwracając stosowną radę:

```
aspect ThrowWin32Errors {
   advice call(win32::Win32API() && "BOOL %(...)") : after() {
      if(FALSE == *tjp->result())  throw ...
   }
   advice call(win32::Win32API() && "HANDLE %(...)") : after() {
      if((NULL == *tjp->result())
            || (INVALID_HANDLE_VALUE == *tjp->result()))
            throw ...
   } // i tak dalej
};
```

Rozwiązanie to ma jednak kilka wad. Pierwszą jest to, że musimy wielokrotnie pisać niemal identyczną definicję rady. Znacznie poważniejszą wadą jest fakt, że jeśli jakiś typ pominiemy lub w przyszłości Microsoft wprowadzi nowy typ, korzystające z takiego typu funkcje API nie zostaną dopasowane do żadnej z istniejących definicji rad i w efekcie będą bez

słowa pomijane przez aspekt. Z tego też względu poszukamy lepszego, bardziej stabilnego rozwiązania:

```
aspect ThrowWin32Errors {
   advice call(win32::Win32API()) : after() {
      if(win32::IsErrorResult(*tjp->result()))  throw ...
   }
};
```

Poprzez porównanie z zależną od typu wartością magiczną umieściliśmy cały zależny od typu kod w osobnej funkcji `win32::IsErrorResult()`, którą teraz wystarczy przeciążyć dla każdego typu wartości zwracanej (Listing 4.). W zależności od statycznego typu zwracanego przez `*tjp->result()` kompilator odnajdzie stosowną wersję `win32::IsErrorResult()` lub, co ważniejsze, zgłosi protest w przypadku braku takowej, dzięki czemu nie będzie już możliwe ciche pominięcie niektórych funkcji tylko dlatego, że zapomnieliśmy o zwracanym przez nie typie.

Powyższa definicja rady jest dobrym przykładem rady generycznej. Dopasowuje ona właściwą implementację (w tym przypadku sprawdzaną wartość magiczną) do informacji na temat typu (tutaj typu wartości zwracanej przez funkcję) według kontekstu bieżącego punktu złączenia. Rozwiązanie to bardzo przypomina techniki stosowane w bibliotekach szablonów tworzonych na potrzeby programowania generycznego, jak na przykład STL.

Listing 3. *Aspekt zgłaszający błędy Win32 jako wyjątki*

```
#include <strstream>
#include "win32-helper.h"
aspect ThrowWin32Errors {
   using namespace std;
   // Szablonowy metaprogram generujący
   // kod do strumieniowania
   // sekwencji argumentów oddzielonych przecinkami
   template< class TJP, int N >
   struct stream_params {
      static void process( ostream& os, TJP* tjp ) {
         os << *tjp->arg< TJP::ARGS - N >() << ", ";
         stream_params< TJP, N - 1 >::process( os, tjp );
      }
   };
   // Specjalizacja dla zakończenia rekurencji
   template< class TJP >
   struct stream_params< TJP, 1 > {
      static void process( ostream& os, TJP* tjp ) {
         os << *tjp->arg< TJP::ARGS - 1 >(); }
   };
   advice call( win32::Win32API() ) : after() {
      if( win32::IsErrorResult( *tjp->result() ) ) {
         ostringstream os;
         DWORD code = GetLastError();
         os << "WIN32 ERROR " << code << ": "
               << win32::GetErrorText(code) << endl;
         os << "WHILE CALLING: "
               << tjp->signature() << endl;
         os << "WITH: " << "(";
         // Wygeneruj sekwencję operacji dla konkretnego
         // punktu złączenia w celu zestrumieniowania
         // wszystkich wartości argumentów
         stream_params< JoinPoint, JoinPoint::ARGS >::
               process( os, tjp );
         os << ")";
         throw win32::Exception( os.str(), code );
      }
   }
};
```

Listing 4. *Plik win32helper.h*

```
namespace win32 {
   struct Exception {
      Exception( const std::string& w, DWORD c ) { ... }
   };
   // Sprawdź "wartość magiczną" oznaczającą błąd
   inline bool IsErrorResult( HANDLE res ) {
      return res == NULL || res == INVALID_HANDLE_VALUE; }
   inline bool IsErrorResult( HWND res ) {
      return res == NULL; }
   inline bool IsErrorResult( BOOL res ) {
      return res == FALSE; }
   ...

   // Tłumaczy kod błędu Win32 na zrozumiały komunikat
   std::string GetErrorText( DWORD code ) { ... }
   pointcut  Win32API() =  "% CreateWindow%(...)"
         || "% BeginPaint(...)"
         || "% CreateFile%(...)"
         || ...
} // namespace Win32
```

Zgłaszanie błędów Win32

Skoro już wiemy, jak w naszym aspekcie poprawnie wykrywać nieudane odwołania do API Win32, kolejnym krokiem będzie ich zgłaszanie w formie wyjątków. Obiekt wyjątku powinien zawierać wszystkie informacje na temat kontekstu, jakie mogą być przydatne przy ustalaniu przyczyny wystąpienia błędu. Oprócz kodu błędu Win32 wśród tych informacji powinien się znaleźć zrozumiały dla człowieka opis błędu, sygnatura wywoływanej funkcji (pobrana za pomocą `JoinPoint::signature()`) oraz wartości parametrów, z jakimi funkcja została wywołana.

Rada generatywna

Najtrudniejszą częścią jest wygenerowanie ciągu wartości parametrów, wymagające przekazania każdego parametru do strumienia klasy `std::ostringstream`. Rada będzie miała wpływ na funkcje o bardzo różnych sygnaturach, więc jej implementacja musi być na tyle ogólna, by obsłużyć różne liczby i rodzaje argumentów funkcji. W praktyce oznacza to, że sekwencja wywołań `operator <<(std::ostream&, T)` musi być wygenerowana na podstawie sygnatury rozpatrywanej funkcji. Implementacja (przedstawiona na Listingu 3.) polega na przekazaniu informacji dostarczonych przez API punktów złączenia (Tabela 1.) do małego, szablonowego metaprogramu. Instancje tego metaprogramu są inicjalizowane przez kod rady typem `JoinPoint`,

po czym każda instancja przechodzi (za pomocą rekurencyjnego tworzenia kolejnych instancji) przez właściwą danemu punktowi złączenia listę typów argumentów `JoinPoint::Arg<I>`. Dla każdego typu argumentu generowana jest klasa `stream_params` z metodą `process()`, która w czasie wykonania strumieniowo przekaże wartość argumentu określonego typu (pobraną poprzez `tjp->arg<I>()`) i rekurencyjnie wywoła `stream_params::process()` dla kolejnego argumentu.

Czego się nauczyliśmy

Jak pokazaliśmy na przykładzie `ThrowWin32Errors`, aspekty mogą być wykorzystywane nie tylko w aplikacjach tworzonych obiektowo, ale mogą też przydać się na przykład do ulepszania istniejącego kodu w stylu C. W połączeniu z szablonami C++ możliwe jest implementowanie aspektów o bardzo wysokim stopniu ogólności. Z takim połączeniem wiążą się kluczowe pojęcia AspectC++:

- *Rada generyczna:* rada wykorzystująca statyczne informacje o typach, uzyskane z kontekstu bieżącego punktu złączenia w celu utworzenia lub przypisania instancji kodu generycznego.
- *Rada generatywna:* rada o implementacji częściowo generowanej poprzez tworzenie instancji szablonów na podstawie statycznych informacji o typach, pobranych z kontekstu bieżącego punktu złączenia.

Co dalej?

Przyjrzeliśmy się już wszystkim najważniejszym konstrukcjom językowym AspectC++. Aspekty `Tracing`, `ObserverPattern` i `ThrowWin32Error` są rzecz jasna jedynie przykładami ilustrującymi różnorodność problemów przekrojowych, które można rozwiązać za pomocą podejścia aspektowego. Zresztą pewnie masz już pomysły na wykorzystanie technik aspektowych we własnych projektach C++. Zobaczmy jeszcze, jakie narzędzia dla AspectC++ pozwolą ten cel osiągnąć.

Wsparcie narzędziowe

Programowanie aspektowe polega na modularyzacji implementacji problemów przekrojowych w postaci aspektów. Zbudowanie ostatecznej wersji programu wymaga zatem wplecenia kodu aspektowego w modyfikowane elementy kodu C++. Zadanie to realizuje specjalny parser, zwany tkaczem (ang. *weaver*).

Niekoniecznym, ale bardzo wygodnym dodatkiem jest narzędzie do wizualizacji punktów złączenia. Aspekty mają możliwość wprowadzania modyfikacji w dowolnym miejscu programu, co przy dużych projektach wiąże się z ryzykiem zaskakującego działania programu w sytuacji, gdzie twórcy kodu komponentów nic nie wiedzą o kodzie aspektów. Oznacza to, że wszystkie punkty złączenia podlegające działaniu aspektu powinny być automatycznie oznaczane w kodzie źródłowym, co wskaże programistom miejsca, w których aspekty wpływają na ich kod.

Generowanie kodu w AspectC++

Dołączony do AspectC++ tkacz *ac++* jest parserem typu *źródło na źródło*, przekształcającym kod AspectC++ w kod C++, który następnie można kompilować dowolnie wybranym kompilatorem C++ przestrzegającym standardu. Szczególnie dobrze obsługiwane są g++ (3.x) oraz Microsoft C++ (Visual Studio .NET).

W celu poprawnej identyfikacji punktów łączeń ac++ przeprowadza kompletną syntaktyczną i semantyczną analizę wejściowego kodu AspectC++. Biorąc pod uwagę stopień złożoności języka C++ projekt ten można by uznać za nazbyt ambitny, lecz ac++ jest już teraz w stanie przetwarzać prawdziwy kod C++, w tym nawet złożone szablony zdefiniowane w STL lub bibliotekę Microsoft ATL. W niedalekiej przyszłości obsługiwane będą również bardziej zaawansowane biblioteki szablonów w rodzaju biblioteki Boost.

Wtyczka ACDT dla Eclipse

AspectC++ Development Tool dla Eclipse (ACDT) to wtyczka Eclipse oparta na kodzie projektu CDT. Rozszerza ona narzędzia programistyczne C++ o podświetlanie składni dla słów kluczowych AspectC++, rozbudowany konspekt pokazujący aspekty, rady i punkty cięć (patrz Rysunek 2.), budowniczego projektów *Managed Make* oraz wizualizację punktów łączeń w widoku konspektu i edytorze kodu źródłowego, nawet w projektach *Standard Make* opartych na istniejących plikach Makefile.

Moduł AspectC++ dla Visual Studio .NET

Komercyjne rozszerzenie VisualStudio.NET dla AspectC++ oferuje firma pure-systems. Obsługuje ono rozmaite rozszerzenia języka specyficzne dla Visual C++, dzięki czemu jest najlepszym wyborem dla programistów przyzwyczajonych do IDE Visual Studio oraz kompilatora Microsoft Visual C++. Dostępna jest też darmowa wersja próbna.

Podsumowanie i wnioski

Choć programowanie aspektowe kojarzy się przede wszystkim z Javą, możliwe jest również jego stosowanie w projektach C++. W tym artykule poznaliśmy najważniejsze koncepcje i konstrukcje językowe AspectC++. Programowanie aspektowe może programistom przynieść wiele korzyści. Aspekty programistyczne w rodzaju `Tracing` są dobrym wstępem do korzystania z AOP i mogą niekiedy zaoszczędzić programistom mnóstwo pracy. Z naszych badań wynika, że w niektórych prawdziwych projektach aż 25% kodu było poświęcone na śledzenie, profilowanie i sprawdzanie ograniczeń. Aspekty produkcyjne można znaleźć niemal wszędzie. Jak pokazują nasze przykłady, mogą one uprościć projektowanie, implementację, a nawet korzystanie ze starszych bibliotek. Mamy nadzieję, że lektura tego artykułu będzie Twoim pierwszym krokiem ku aspektowości! ■

W Sieci

- Strona domowa projektu AspectC++
  http://www.aspectc.org/
- AspectC++ Development Tools (ACDT) dla Eclipse'a
  http://acdt.aspectc.org/
- Portal WWW o programowaniu aspektowym
  http://www.aosd.net/
- Firma oferująca moduł AspectC++ dla Visual Studio .NET oraz komercyjne wsparcie dla użytkowników AspectC++
  http://www.pure-systems.com/