C++. 50 efektywnych sposobów na udoskonalenie Twoich programów

Autor: Scott Meyers
Tłumaczenie: Mikołaj Szczepaniak
ISBN: 83-7361-345-5
Tytuł oryginału: Effective C++: 50 Specific Ways to Improve Your Programs and Design
Format: B5, stron: 248

Pierwsze wydanie książki „C++. 50 efektywnych sposobów na udoskonalenie twoich programów” zostało sprzedane w nakładzie 100 000 egzemplarzy i zostało przetłumaczone na cztery języki. Nietrudno zrozumieć, dlaczego tak się stało. Scott Meyers w charakterystyczny dla siebie, praktyczny sposób przedstawił wiedzę typową dla ekspertów – czynności, które niemal zawsze wykonują lub czynności, których niemal zawsze unikają, by tworzyć prosty, poprawny i efektywny kod. Każda z zawartych w tej książce pięćdziesięciu wskazówek jest streszczeniem metod pisania lepszych programów w C++, zaś odpowiednie rozważania są poparte konkretnymi przykładami. Z myślą o nowym wydaniu, Scott Meyers opracował od początku wszystkie opisywane w tej książce wskazówki. Wynik jego pracy jest wyjątkowo zgodny z międzynarodowym standardem C++, technologią aktualnych kompilatorów oraz najnowszymi trendami w świecie rzeczywistych aplikacji C++.

Do najważniejszych zalet książki „C++. 50 efektywnych sposobów na udoskonalenie twoich programów” należą:

- Eksperckie porady dotyczące projektowania zorientowanego obiektowo, projektowania klas i właściwego stosowania technik dziedziczenia
- Analiza standardowej biblioteki C++, włącznie z wpływem standardowej biblioteki szablonów oraz klas podobnych do string i vector na strukturę dobrze napisanych programów
- Rozważania na temat najnowszych możliwości języka C++: inicjalizacji stałych wewnątrz klas, przestrzeni nazw oraz szablonów składowych
- Wiedza będąca zwykle w posiadaniu wyłącznie doświadczonych programistów

Książka „C++. 50 efektywnych sposobów na udoskonalenie twoich programów” pozostaje jedną z najważniejszych publikacji dla każdego programisty pracującego z C++.

Scott Meyers jest znanym autorytetem w dziedzinie programowania w języku C++; zapewnia usługi doradcze dla klientów na całym świecie i jest członkiem rady redakcyjnej pisma C++ Report. Regularnie przemawia na technicznych konferencjach na całym świecie, jest także autorem książek „More Effective C++” oraz „Effective C++ CD”. W 1993. roku otrzymał tytuł doktora informatyki na Brown University.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Dziedziczenie i projektowanie zorientowane obiektowo

Wielu programistów wyraża opinię, że możliwość dziedziczenia jest jedyną korzyścią płynącą z programowania zorientowanego obiektowo. Można mieć oczywiście różne zdanie na ten temat, jednak liczba zawartych w innych częściach tej książki sposobów poświęconych efektywnemu programowaniu w C++ pokazuje, że masz do dyspozycji znacznie więcej rozmaitych narzędzi, niż tylko określanie, które klasy powinny dziedziczyć po innych klasach.

Projektowanie i implementowanie hierarchii klas różni się od zasadniczo od wszystkich mechanizmów dostępnych w języku C. Problem dziedziczenia i projektowania zorientowanego obiektowo z pewnością zmusza do ponownego przemyślenia swojej strategii konstruowania systemów oprogramowania. Co więcej, język C++ udostępnia bardzo szeroki asortyment bloków budowania obiektów, włącznie z publicznymi, chronionymi i prywatnymi klasami bazowymi, wirtualnymi i niewirtualnymi klasami bazowymi oraz wirtualnymi i niewirtualnymi funkcjami składowymi. Każda z wymienionych własności może wpływać także na pozostałe komponenty języka C++. W efekcie, próby zrozumienia, co poszczególne własności oznaczają, kiedy powinny być stosowane oraz jak można je w najlepszy sposób połączyć z nieobiektowymi częściami języka C++ może niedoświadczonych programistów zniechęcić.

Dalszą komplikacją jest fakt, że różne własności języka C++ są z pozoru odpowiedzialne za te same zachowania. Oto przykłady:

- Potrzebujesz zbioru klas zawierających wiele elementów wspólnych. Powinieneś wykorzystać mechanizm dziedziczenia i stworzyć klasy potomne względem jednej wspólnej klasy bazowej czy powinieneś wykorzystać szablony i wygenerować wszystkie potrzebne klasy ze wspólnym szkieletem kodu?

- Klasa A ma zostać zaimplementowana w oparciu o klasę B. Czy A powinna zawierać składową reprezentującą obiekt klasy B czy też powinna prywatnie dziedziczyć po klasie B?
- Potrzebujesz projektu bezpiecznej pod względem typu i homogenicznej klasy pojemnikowej, która nie jest dostępna w standardowej bibliotece C++ (listę pojemników *udostępnianych* przez tę bibliotekę podano, prezentując sposób 49.). Czy lepszym rozwiązaniem będzie skonstruowanie szablonów czy budowa bezpiecznych pod względem typów interfejsów wokół tej klasy, która sama byłaby zaimplementowana za pomocą ogólnych (`void*`) wskaźników?

W sposobach prezentowanych w tej części zawarłem wskazówki, jak należy znajdować odpowiedzi na powyższe pytania. Nie mogę jednak liczyć na to, że uda mi się znaleźć właściwe rozwiązania dla wszystkich aspektów projektowania zorientowanego obiektowo. Zamiast tego skoncentrowałem się więc na wyjaśnianiu, co *faktycznie* oznaczają poszczególne własności języka C++ i co tak *naprawdę* sygnalizujesz, stosując poszczególne dyrektywy czy instrukcje. Przykładowo, publiczne dziedziczenie oznacza relację „jest" lub specjalizacji-generalizacji (ang. *isa*, patrz sposób 35.) i jeśli musisz nadać mu jakikolwiek inny sens, możesz napotkać pewne problemy. Podobnie, funkcje wirtualne oznaczają, że „interfejs musi być dziedziczony", natomiast funkcje niewirtualne oznaczają, że „dziedziczony musi być zarówno interfejs, *jak i* implementacja". Brak rozróżnienia tych znaczeń doprowadził już wielu programistów C++ do trudnych do opisania nieszczęść.

Jeśli rozumiesz znaczenia rozmaitych własności języka C++, odkryjesz, że Twój pogląd na projektowanie zorientowane obiektowo powoli ewoluuje. Zamiast przekonywać Cię o istniejących różnicach pomiędzy konstrukcjami językowymi, treść poniższych sposobów ułatwi Ci ocenę jakości opracowanych dotychczas systemów oprogramowania. Będziesz potem w stanie przekształcić swoją wiedzę w swobodne i właściwe operowanie własnościami języka C++ celem tworzenia jeszcze lepszych programów.

Wartości wiedzy na temat znaczeń i konsekwencji stosowania poszczególnych konstrukcji nie da się przecenić. Poniższe sposoby zawierają szczegółową analizę metod efektywnego stosowania omawianych własności języka C++. W sposobie 44. podsumowałem cechy i znaczenia poszczególnych konstrukcji obiektowych tego języka. Treść tego sposobu należy traktować jak zwieńczenie całej części, a także zwięzłe streszczenie, do którego warto zaglądać w przyszłości.

Sposób 35. Dopilnuj, by publiczne dziedziczenie modelowało relację „jest"

William Dement w swojej książce pt. *Some Must Watch While Some Must Sleep* (W. H. Freeman and Company, 1974) opisał swoje doświadczenia z pracy ze studentami, kiedy próbował utrwalić w ich umysłach najistotniejsze tezy swojego wykładu.

Mówił im, że przyjmuje się, że świadomość historyczna przeciętnego brytyjskiego dziecka w wieku szkolnym wykracza poza wiedzę, że bitwa pod Hastings odbyła się w roku 1066. William Dement podkreśla, że jeśli dziecko pamięta więcej szczegółów, musi także pamiętać o tej historycznej dla Brytyjczyków dacie. Na tej podstawie autor wnioskuje, że w umysłach *jego* studentów zachowuje się tylko kilka istotnych i najciekawszych faktów, włącznie z tym, że np. tabletki nasenne powodują bezsenność. Namawiał studentów, by zapamiętali przynajmniej tych kilka najważniejszych faktów, nawet jeśli mają zapomnieć wszystkie pozostałe zagadnienia dyskutowane podczas wykładów. Autor książki przekonywał do tego swoich studentów wielokrotnie w czasie semestru.

Ostatnie pytanie testu w sesji egzaminacyjnej brzmiało: „wymień jeden fakt, który wyniosłeś z moich wykładów, i który na pewno zapamiętasz do końca życia”. Po sprawdzeniu egzaminów Dement był zszokowany — niemal wszyscy napisali „1066”.

Jestem teraz pełen obaw, że jedynym istotnym wnioskiem, który wyniesiesz z tej książki na temat programowania zorientowanego obiektowo w C++ będzie to, że mechanizm publicznego dziedziczenia oznacza relację „jest”. Zachowaj jednak ten fakt w swojej pamięci.

Jeśli piszesz klasę D (od ang. *Derived*, czyli klasę potomną), która publicznie dziedziczy po klasie B (od ang. *Base*, czyli klasy bazowej), sygnalizujesz kompilatorom C++ (i przyszłym czytelnikom Twojego kodu), że każdy obiekt typu D jest także obiektem typu B, *ale nie odwrotnie*. Sygnalizujesz, że B reprezentuje bardziej ogólne pojęcia niż D, natomiast D reprezentuje bardziej konkretne pojęcia niż B. Utrzymujesz, że wszędzie tam, gdzie może być użyty obiekt typu B, może być także wykorzystany obiekt typu D, ponieważ każdy obiekt typu D *jest* także obiektem typu B. Z drugiej strony, jeśli potrzebujesz obiektu typu D, obiekt typu B nie będzie mógł go zastąpić — publiczne dziedziczenie oznacza relację D „jest” B, ale nie odwrotnie.

Taką interpretację publicznego dziedziczenia wymusza język C++. Przeanalizujmy poniższy przykład:

```
class Person { ... };
class Student: public Person { ... };
```

Oczywiste jest, że każdy student jest osobą, nie każda osoba jest jednak studentem. Dokładnie takie samo znaczenie ma powyższa hierarchia. Oczekujemy, że wszystkie istniejące cechy danej osoby (np. to, że ma jakąś datę urodzenia) istnieją także dla studenta; nie oczekujemy jednak, że wszystkie dane dotyczące studenta (np. adres szkoły, do której uczęszcza) będą istotne dla wszystkich ludzi. Pojęcie osoby jest bowiem bardziej ogólne, niż pojęcie studenta — student jest specyficznym „rodzajem” osoby.

W języku C++ każda funkcja oczekująca argumentu typu `Person` (lub wskaźnika do obiektu klasy `Person` bądź referencji do obiektu klasy `Person`) może zamiennie pobierać obiekt klasy `Student` (lub wskaźnik do obiektu klasy `Student` bądź referencję do obiektu klasy `Student`):

```
void dance(const Person& p);     // każdy może tańczyć

void study(const Student& s);    // tylko studenci mogą studiować
```

```
Person p;                       // p reprezentuje osobę (obiekt klasy Person)
Student s;                      // s reprezentuje studenta (obiekt klasy Student)

dance(p);                       // dobrze, p reprezentuje osobę
dance(s);                       // dobrze, s reprezentuje studenta,
                                // a więc także osobę

study(s);                       // dobrze
study(p);                       // błąd! p nie reprezentuje studenta
```

Powyższe komentarze są prawdziwe tylko dla *publicznego* dziedziczenia. C++ będzie się zachowywał w opisany sposób tylko w przypadku, gdy klasa `Student` będzie publicznie dziedziczyła po klasie `Person`. Dziedziczenie prywatne oznacza coś zupełnie innego (patrz sposób 42.), natomiast znaczenie dziedziczenia chronionego jest nieznane.

Równoważność dziedziczenia publicznego i relacji „jest" wydaje się oczywista, w praktyce jednak właściwe modelowanie tej relacji nie jest już takie proste. Niekiedy nasza intuicja może się okazać zawodna. Przykładowo, faktem jest, że pingwin to ptak; faktem jest także, że ptaki mogą latać. Gdybyśmy w swojej naiwności spróbowali wyrazić to w C++, nasze wysiłki przyniosłyby efekt podobny do poniższego:

```
class Bird {
public:
  virtual void fly();             // ptaki mogą latać
  ...
};

class Penguin: public Bird {      // pingwiny są ptakami
  ...
};
```

Mamy teraz problem, ponieważ z powyższej hierarchii wynika, że pingwiny mogą latać, co jest oczywiście nieprawdą. Co stało się z naszą strategią?

W tym przypadku padliśmy ofiarą nieprecyzyjnego języka naturalnego (polskiego). Kiedy mówimy, że ptaki mogą latać, w rzeczywistości nie mamy na myśli tego, że *wszystkie* ptaki potrafią latać, a jedynie, że w ogólności ptaki mają możliwość latania. Gdybyśmy byli bardziej precyzyjni, wyrazilibyśmy się inaczej, by podkreślić fakt, że istnieje wiele gatunków ptaków, które nie latają — otrzymalibyśmy wówczas poniższą, znacznie lepiej modelującą rzeczywistość, hierarchię klas:

```
class Bird {
  ...                             // brak deklaracji funkcji fly
};

class FlyingBird: public Bird {
public:
  virtual void fly();
  ...
};

class NonFlyingBird: public Bird {
  ...                             // brak deklaracji funkcji fly
};
```

```
class Penguin: public NonFlyingBird {
  ...                          // brak deklaracji funkcji fly
};
```

Powyższa hierarchia jest znacznie bliższa naszej rzeczywistej wiedzy na temat ptaków, niż ta zaprezentowana wcześniej.

Nasze rozwiązanie nie jest jednak jeszcze skończone, ponieważ w niektórych systemach oprogramowania, proste stwierdzenie, że pingwin jest ptakiem, będzie całkowicie poprawne. W szczególności, jeśli nasza aplikacja dotyczy wyłącznie dziobów i skrzydeł, a w żadnym stopniu nie wiąże się z lataniem, oryginalna hierarchia będzie w zupełności wystarczająca. Mimo że jest to dosyć irytujące, omawiana sytuacja jest prostym odzwierciedleniem faktu, że nie istnieje jedna doskonała metoda projektowania dowolnego oprogramowania. Dobry projekt musi po prostu uwzględniać wymagania stawiane przed tworzonym systemem, zarówno te w danej chwili oczywiste, jak i te, które mogą się pojawić w przyszłości. Jeśli nasza aplikacja nie musi i nigdy nie będzie musiała uwzględniać możliwości latania, rozwiązaniem w zupełności wystarczającym będzie stworzenie klasy `Penguin` jako potomnej klasy `Bird`. W rzeczywistości taki projekt może być nawet lepszy niż rozróżnienie ptaków latających od nielatających, ponieważ takie rozróżnienie może w ogóle nie istnieć w modelowanym świecie. Dodawanie do hierarchii niepotrzebnych klas jest błędną decyzją projektową, ponieważ narusza prawidłowe relacje dziedziczenia pomiędzy klasami.

Istnieje jeszcze inna strategia postępowania w przypadku omawianego problemu: „wszystkie ptaki mogą latać, pingwiny są ptakami, pingwiny nie mogą latać”. Strategia polega na wprowadzeniu takich zmian w definicji funkcji `fly`, by dla pingwinów generowany był błąd wykonania:

```
void error(const string& msg);     // funkcja zdefiniowana w innym miejscu

class Penguin: public Bird {
public:
  virtual void fly() { error("Pingwiny nie mogą latać!"); }
  ...
};
```

Do takiego rozwiązania dążą twórcy języków interpretowanych (jak Smalltalk), jednak istotne jest prawidłowe rozpoznanie rzeczywistego znaczenia powyższego kodu, które jest zupełnie inne, niż mógłbyś przypuszczać. Ciało funkcji *nie* oznacza bowiem, że „pingwiny nie mogą latać”. Jej faktyczne znaczenie to: „pingwiny mogą latać, jednak kiedy próbują to robić, powodują błąd”. Na czym polega różnica pomiędzy tymi znaczeniami? Wynika przede wszystkim z możliwości wykrycia błędu — ograniczenie „pingwiny nie mogą latać” może być egzekwowane przez kompilatory, natomiast naruszenie ograniczenia „podejmowana przez pingwiny próba latania powoduje błąd” może zostać wykryte tylko podczas wykonywania programu.

Aby wyrazić ograniczenie „pingwiny nie mogą latać”, wystarczy nie definiować odpowiedniej funkcji dla obiektów klasy `Penguin`:

```
class Bird {
  ...                          // brak deklaracji funkcji fly
};
```

```
class NonFlyingBird: public Bird {
  ...                           // brak deklaracji funkcji fly
};

class Penguin: public NonFlyingBird {
  ...                           // brak deklaracji funkcji fly
};
```

Jeśli spróbujesz teraz wywołać funkcję `fly` dla obiektu reprezentującego pingwina, kompilator zasygnalizuje błąd:

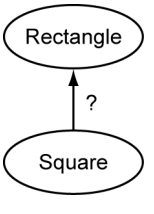
```
Penguin p;
p.fly();                        // błąd!
```

Zaprezentowane rozwiązanie jest całkowicie odmienne od podejścia stosowanego w języku Smalltalk. Stosowana tam strategia powoduje, że kompilator skompilowałby podobny kod bez przeszkód.

Filozofia języka C++ jest jednak zupełnie inna niż filozofia języka Smalltalk, dopóki jednak programujesz w C++, powinieneś stosować się wyłącznie do reguł obowiązujących w tym języku. Co więcej, wykrywanie błędów w czasie kompilacji (a nie w czasie wykonywania) programu wiąże się z pewnymi technicznymi korzyściami — patrz sposób 46.

Być może przyznasz, że Twoja wiedza z zakresu ornitologii ma raczej intuicyjny charakter i może być zawodna, zawsze możesz jednak polegać na swojej biegłości w dziedzinie podstawowej geometrii, prawda? Nie martw się, mam na myśli wyłącznie prostokąty i kwadraty.

Spróbuj więc odpowiedzieć na pytanie: czy reprezentująca kwadraty klasa `Square` publicznie dziedziczy po reprezentującej prostokąty klasie `Rectangle`?



Powiesz pewnie: „Też coś! Każde dziecko wie, że kwadrat jest prostokątem, ale w ogólności prostokąt nie musi być kwadratem”. Tak, to prawda, przynajmniej na poziomie gimnazjum. Nie sądzę jednak, byśmy kiedykolwiek wrócili do nauki na tym poziomie.

Przeanalizuj więc poniższy kod:

```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);

  virtual int height() const;          // funkcje zwracają bieżące
  virtual int width() const;           // wartości
  ...
};
```

```
void makeBigger(Rectangle& r)          // funkcja zwiększająca pole
{                                      // powierzchni prostokąta r
  int oldHeight = r.height();

  r.setWidth(r.width() + 10);          // dodaje 10 do szerokości r

  assert(r.height() == oldHeight);     // upewnia się, że wysokość
}                                      // prostokąta r nie zmieniła się
```

Jest oczywiste, że ostatnia instrukcja nigdy nie zakończy się niepowodzeniem, ponieważ funkcja `makeBigger` modyfikuje wyłącznie szerokość prostokąta reprezentowanego przez `r`.

Rozważ teraz poniższy fragment kodu, w którym wykorzystujemy publiczne dziedziczenie umożliwiające traktowanie kwadratów jak prostokątów:

```
class Square: public Rectangle { ... };
Square s;
...
assert(s.width() == s.height());       // warunek musi być prawdziwy
                                       // dla wszystkich kwadratów

makeBigger(s);                         // na skutek dziedziczenia, s
                                       // jest w relacji "jest" z klasą
                                       // Rectangle, możemy więc zwiększyć
                                       // pole jego powierzchni

assert(s.Width() == s.height());       // warunek nadal musi być prawdziwy
                                       // dla wszystkich kwadratów
```

Także teraz oczywiste jest, że ostatni warunek nigdy nie powinien być fałszywy. Zgodnie z definicją, szerokość kwadratu jest przecież taka sama jak jego wysokość.

Tym razem mamy jednak problem. Jak można pogodzić poniższe twierdzenia?

- Przed wywołaniem funkcji `makeBigger` wysokość kwadratu reprezentowanego przez obiekt `s` jest taka sama jak jego szerokość.
- Wewnątrz funkcji `makeBigger` modyfikowana jest szerokość kwadratu, jednak wysokość pozostaje niezmieniona.
- Po zakończeniu wykonywania funkcji `makeBigger` wysokość kwadratu `s` ponownie jest taka sama jak jego szerokość (zauważ, że obiekt `s` jest przekazywany do funkcji `makeBigger` przez referencję, zatem funkcja modyfikuje ten sam obiekt `s`, nie jego kopię).

Jak to możliwe?

Witaj w cudownym świecie publicznego dziedziczenia, w którym Twój instynkt — sprawdzający się do tej pory w innych dziedzinach, włącznie z matematyką — może nie być tak pomocny, jak tego oczekujesz. Zasadniczym problemem jest w tym przypadku to, że operacja, którą można stosować dla prostokątów (jego szerokość może być zmieniana niezależnie od wysokości), nie może być stosowana dla kwadratów (definicja figury wymusza równość jej szerokości i wysokości). Mechanizm publicznego dziedziczenia zakłada jednak, że absolutnie *wszystkie* operacje stosowane z powodzeniem dla obiektów klasy bazowej mogą być stosowane także dla obiektów klasy

potomnej. W przypadku prostokątów i kwadratów (podobny przykład dotyczący zbiorów i list omawiam w sposobie 40.) to założenie się nie sprawdza, zatem stosowanie publicznego dziedziczenia do modelowania występującej między nimi relacji jest po prostu błędne. Kompilatory oczywiście umożliwią Ci zaprogramowanie takiego modelu, jednak — jak się już przekonaliśmy — nie mamy gwarancji, że nasz program będzie się zachowywał prawidłowo. Od czasu do czasu każdy programista musi się przekonać (niektórzy częściej, inni rzadziej), że poprawne skompilowanie programu nie oznacza, że będzie on działał zgodnie z oczekiwaniami.

Nie denerwuj się, że rozwijana przez lata intuicja dotycząca tworzonego oprogramowania traci moc w konfrontacji z projektowaniem zorientowanym obiektowo. Twoja wiedza jest nadal cenna, jednak dodałeś właśnie do swojego arsenału rozwiązań projektowych silny mechanizm dziedziczenia i będziesz musiał rozszerzyć swoją intuicję w taki sposób, by prowadziła Cię do właściwego wykorzystywania nowych umiejętności. Z czasem problem klasy `Penguin` dziedziczącej po klasie `Bird` lub klasie `Square` dziedziczącej po klasie `Rectangle` będzie dla Ciebie równie zabawny jak prezentowane Ci przez niedoświadczonych programistów funkcje zajmujące wiele stron. *Możliwe*, że proponowane podejście do tego typu problemów jest właściwe, nadal jednak nie jest to bardzo prawdopodobne.

Relacja „jest” nie jest oczywiście jedyną relacją występującą pomiędzy klasami. Dwie pozostałe powszechnie stosowane relacje między klasami to relacja „ma” (ang. *has-a*) oraz relacja implementacji z wykorzystaniem (ang. *is-implemented-in-terms-of*). Relacje te przeanalizujemy podczas prezentacji sposobów 40. i 42. Nierzadko projekty C++ ulegają zniekształceniom, ponieważ któraś z pozostałych najważniejszych relacji została błędnie zamodelowana jako „jest”, powinniśmy więc być pewni, że właściwie rozróżniamy te relacje i wiemy, jak należy je najlepiej modelować w C++.

Sposób 36. Odróżniaj dziedziczenie interfejsu od dziedziczenia implementacji

Po przeprowadzeniu dokładnej analizy okazuje się, że pozornie oczywiste pojęcie (publicznego) dziedziczenia składa się w rzeczywistości z dwóch rozdzielnych części — dziedziczenia interfejsów funkcji oraz dziedziczenia implementacji funkcji. Różnica pomiędzy wspomnianymi rodzajami dziedziczenia ściśle odpowiada różnicy pomiędzy deklaracjami a definicjami funkcji (omówionej we wstępie do tej książki).

Jako projektant klasy potrzebujesz niekiedy takich klas potomnych, które dziedziczą wyłącznie interfejs (deklarację) danej funkcji składowej; czasem potrzebujesz klas potomnych dziedziczących zarówno interfejs, jak i implementację danej funkcji, jednak masz zamiar przykryć implementację swoim rozwiązaniem; zdarza się także, że potrzebujesz klas potomnych dziedziczących zarówno interfejs, jak i implementację danej funkcji, ale bez możliwości przykrywania czegokolwiek.

Aby lepiej zrozumieć różnicę pomiędzy zaproponowanymi opcjami, przeanalizuj poniższą hierarchię klas reprezentującą figury geometryczne w aplikacji graficznej:

```
class Shape {
public:
  virtual void draw() const = 0;
  virtual void error(const string& msg);
  int objectID() const;
  ...
};

class Rectangle: public Shape { ... };
class Ellipse: public Shape { ... };
```

`Shape` jest klasą abstrakcyjną. Można to poznać po obecności czystej funkcji wirtualnej `draw`. W efekcie klienci nie mogą tworzyć egzemplarzy klasy `Shape`, mogą to robić wyłącznie klasy potomne. Mimo to klasa `Shape` wywiera ogromny nacisk na wszystkie klasy, które (publicznie) po niej dziedziczą, ponieważ:

- *Interfejsy* funkcji składowych *zawsze są dziedziczone*. W sposobie 35. wyjaśniłem, że dziedziczenie publiczne oznacza faktycznie relację „jest”, zatem wszystkie elementy istniejące w klasie bazowej muszą także istnieć w klasach potomnych. Jeśli więc daną funkcję można wykonać dla danej klasy, musi także istnieć sposób jej wykonania dla jej podklas.

W funkcji `Shape` zadeklarowaliśmy trzy funkcje. Pierwsza, `draw`, rysuje na ekranie bieżący obiekt. Druga, `error`, jest wywoływana przez inne funkcje składowe w momencie, gdy konieczne jest zasygnalizowanie błędu. Trzecia, `objectID`, zwraca unikalny całkowitoliczbowy identyfikator bieżącego obiektu (przykład wykorzystania tego typu funkcji znajdziesz w sposobie 17.). Każda z wymienionych funkcji została zadeklarowana w inny sposób: `draw` jest czystą funkcją wirtualną, `error` jest prostą (nieczystą?) funkcją wirtualną, natomiast `objectID` jest funkcją niewirtualną. Jakie jest znaczenie tych trzech różnych deklaracji?

Rozważmy najpierw czystą funkcję wirtualną `draw`. Dwie najistotniejsze cechy czystych funkcji wirtualnych to *konieczność* ich ponownego zadeklarowania w każdej dziedziczącej je konkretnej klasie oraz brak ich definicji w klasach abstrakcyjnych. Jeśli połączymy te własności, uświadomimy sobie, że:

- Celem deklarowania czystych funkcji wirtualnych jest otrzymanie klas potomnych dziedziczących *wyłącznie interfejs*.

Jest to idealne rozwiązanie dla funkcji `Shape::draw`, ponieważ naturalne jest udostępnienie możliwości rysowania wszystkich obiektów klasy `Shape`, jednak niemożliwe jest opracowanie jednej domyślnej implementacji dla takiej funkcji. Algorytm rysowania np. elips różni się przecież znacznie od algorytmu rysowania prostokątów. Właściwym sposobem interpretowania znaczenia deklaracji funkcji `Shape::draw` jest instrukcja skierowana do projektantów podklas: „musicie stworzyć funkcję `draw`, jednak nie mam pojęcia, jak moglibyście ją zaimplementować”.

Istnieje niekiedy możliwość opracowania definicji czystej funkcji wirtualnej. Oznacza to, że możesz stworzyć taką implementację dla funkcji `Shape::draw`, że kompilatory C++ nie zgłoszą żadnych zastrzeżeń, jednak jedynym sposobem jej wywołania byłoby wykorzystanie pełnej nazwy włącznie z nazwą klasy:

```
Shape *ps = new Shape;          // błąd! Shape jest klasą abstrakcyjną

Shape *ps1 = new Rectangle;     // dobrze
ps1->draw();                    // wywołuje Rectangle::draw

Shape *ps2 = new Ellipse;       // dobrze
ps2->draw();                    // wywołuje Ellipse::draw

ps1->Shape::draw();             // wywołuje Shape::draw
ps2->Shape::draw();             // wywołuje Shape::draw
```

Poza faktem, że powyższe rozwiązanie może zrobić wrażenie na innych programistach podczas imprezy, w ogólności znajomość zaprezentowanego fenomenu jest w praktyce mało przydatna. Jak się jednak za chwilę przekonasz, może być wykorzystywana jako mechanizm udostępniania bezpieczniejszej domyślnej implementacji dla prostych (nieczystych) funkcji wirtualnych.

Niekiedy dobrym rozwiązaniem jest zadeklarowanie klasy zawierającej *wyłącznie* czyste funkcje wirtualne. Takie *klasy protokołu* udostępniają klasom potomnym jedynie interfejsy funkcji, ale nigdy ich implementacje. Klasy protokołu opisałem, prezentując sposób 34., i wspominam o nich ponownie w sposobie 43.

Znaczenie prostych funkcji wirtualnych jest nieco inne niż znaczenie czystych funkcji wirtualnych. W obu przypadkach klasy dziedziczą interfejsy funkcji, jednak proste funkcje wirtualne zazwyczaj udostępniają także swoje implementacje, które mogą (ale nie muszą) być przykryte w klasach potomnych. Po chwili namysłu powinieneś dojść do wniosku, że:

- Celem deklarowania prostej funkcji wirtualnej jest otrzymanie klas potomnych dziedziczących zarówno *interfejs, jak i domyślną implementację* tej funkcji.

W przypadku funkcji `Shape::error` interfejs określa, że każda klasa musi udostępniać funkcję wywoływaną w momencie wykrycia błędu, jednak obsługa samych błędów jest dowolna i zależy wyłącznie od projektantów klas potomnych. Jeśli nie przewidują oni żadnych specjalnych działań w przypadku znalezienia błędu, mogą wykorzystać udostępniany przez klasę `Shape` domyślny mechanizm obsługi błędów. Oznacza to, że rzeczywistym znaczeniem deklaracji funkcji `Shape::error` dla projektantów podklas jest zdanie: „musisz obsłużyć funkcję `error`, jednak jeśli nie chcesz tworzyć własnej wersji tej funkcji, możesz wykorzystać jej domyślną wersję zdefiniowaną dla klasy `Shape`”.

Okazuje się, że zezwalanie prostym funkcjom wirtualnym na precyzowanie zarówno deklaracji, jak i domyślnej implementacji może być niebezpieczne. Aby przekonać się dlaczego, przeanalizuj zaprezentowaną poniżej hierarchię samolotów należących do linii lotniczych XYZ. Linie XYZ posiadają tylko dwa typy samolotów, Model A i Model B, z których oba latają w identyczny sposób. Linie lotnicze XYZ zaprojektowały więc następującą hierarchię klas:

```
class Airport { ... };       // reprezentuje lotniska

class Airplane {
public:
  virtual void fly(const Airport& destination);
  ...
};

void Airplane::fly(const Airport& destination)
{
  domyślny kod modelujący przelot samolotu
  do danego celu
}

class ModelA: public Airplane { ... };
class ModelB: public Airplane { ... };
```

Aby wyrazić fakt, że wszystkie samoloty muszą obsługiwać jakąś funkcję `fly` oraz z uwagi na możliwe wymagania dotyczące innych implementacji tej funkcji generowane przez nowe modele samolotów, funkcja `Airplane::fly` została zadeklarowana jako wirtualna. Aby uniknąć pisania identycznego kodu w klasach `ModelA` i `ModelB`, domyślny model latania został jednak zapisany w formie ciała funkcji `Airplane::fly`, które jest dziedziczone zarówno przez klasę `ModelA`, jak i klasę `ModelB`.

Jest to klasyczny projekt zorientowany obiektowo. Dwie klasy współdzielą wspólny element (sposób implementacji funkcji `fly`), zatem element ten zostaje przeniesiony do klasy bazowej i jest dziedziczony przez te dwie klasy. Takie rozwiązanie ma wiele istotnych zalet: pozwala uniknąć powielania tego samego kodu, ułatwia przyszłe rozszerzenia systemu i upraszcza konserwację w długim okresie czasu — wszystkie wymienione własności są charakterystyczne właśnie dla technologii obiektowej. Linie lotnicze XYZ powinny więc być dumne ze swojego systemu.

Przypuśćmy teraz, że firma XYZ rozwija się i postanowiła pozyskać nowy typ samolotu — Model C. Nowy samolot różni się nieco od Modelu A i Modelu B, a w szczególności ma inne właściwości lotu.

Programiści omawianych linii lotniczych dodają więc do hierarchii klasę reprezentującą samoloty Model C, jednak w pośpiechu zapomnieli ponownie zdefiniować funkcję `fly`:

```
class ModelC: public Airplane {
  ...                        // brak deklaracji funkcji fly
};
```

Ich kod zawiera więc coś podobnego do poniższego fragmentu:

```
Airport Okecie(...);         // Okęcie to lotnisko w Warszawie

Airplane *pa = new ModelC;
...
pa->fly(Okecie);             // wywołuje funkcję Airplane::fly!
```

Mamy do czynienia z prawdziwą katastrofą, a mianowicie z próbą obsłużenia lotu obiektu klasy `ModelC`, jakby był obiektem klasy `ModelA` lub klasy `ModelB`. Z pewnością nie wzbudzimy w ten sposób zaufania u klientów linii lotniczych.

Problem nie polega tutaj na tym, że zdefiniowaliśmy domyślne zachowanie funkcji `Airplane::fly`, tylko na tym, że klasa `ModelC` mogła przypadkowo (na skutek nieuwagi programistów) dziedziczyć to zachowanie. Na szczęście istnieje możliwość przekazywania domyślnych zachowań funkcji do podklas wyłącznie w przypadku, gdy ich twórcy wyraźnie tego zażądają. Sztuczka polega na przerwaniu połączenia pomiędzy *interfejsem* wirtualnej funkcji a jej domyślną *implementacją*. Oto sposób realizacji tego zadania:

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;
  ...

protected:
  void defaultFly(const Airport& destination);
};

void Airplane::defaultFly(const Airport& destination)
{
  domyślny kod modelujący przelot samolotu
  do danego celu
}
```

Zwróć uwagę na sposób, w jaki przekształciliśmy `Airplane::fly` w *czystą* funkcję wirtualną. Zaprezentowana klasa udostępnia tym samym interfejs funkcji obsługującej latanie samolotów. Klasa `Airplane` zawiera także jej domyślną implementację, jednak tym razem w formie niezależnej funkcji, `defaultFly`. Klasy podobne do `ModelA` i `ModelB` mogą wykorzystać domyślną implementację, zwyczajnie wywołując funkcję `defaultFly` wbudowaną w ciało ich funkcji `fly` (jednak zanim to zrobisz, przeczytaj sposób 33., gdzie przeanalizowałem wzajemne oddziaływanie atrybutów `inline` i `virtual` dla funkcji składowych):

```
class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }
  ...
};

class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { defaultFly(destination); }
  ...
};
```

W przypadku klasy `ModelC` nie możemy już przypadkowo dziedziczyć niepoprawnej implementacji funkcji `fly`, ponieważ czysta funkcja wirtualna w klasie `Airplane` wymusza na projektantach nowej klasy stworzenie własnej wersji funkcji `fly`:

```
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);
  ...
};

void ModelC::fly(const Airport& destination)
{
  kod modelujący przelot samolotu typu ModelC do danego celu
}
```

Powyższy schemat postępowania nie jest oczywiście całkowicie bezpieczny (programiści nadal mają możliwość popełniania fatalnych w skutkach błędów), jednak jest znacznie bardziej niezawodny od oryginalnego projektu. Funkcja `Airplane::defaultFly` jest chroniona, ponieważ w rzeczywistości jest szczegółem implementacyjnym klasy `Airplane` i jej klas potomnych. Klienci wykorzystujący te klasy powinni zajmować się wyłącznie własnościami lotu reprezentowanych samolotów, a nie sposobami implementowania tych własności.

Ważne jest także to, że funkcja `Airplane::defaultFly` jest *niewirtualna*. Wynika to z faktu, że żadna z podklas nie powinna jej ponownie definiować — temu zagadnieniu poświęciłem sposób 37. Gdyby funkcja `defaultFly` była wirtualna, mielibyśmy do czynienia ze znanym nam już problemem: co stanie się, jeśli projektant którejś z podklas zapomni ponownie zdefiniować funkcję `defaultFly` w sytuacji, gdzie będzie to konieczne?

Niektórzy programiści sprzeciwiają się idei definiowania dwóch osobnych funkcji dla interfejsu i domyślnej implementacji (jak `fly` i `defaultFly`). Z jednej strony zauważają, że takie rozwiązanie zaśmieca przestrzeń nazw klasy występującymi wielokrotnie zbliżonymi do siebie nazwami funkcji. Z drugiej strony zgadzają się z tezą, że należy oddzielić interfejs od domyślnej implementacji. Jak więc powinniśmy radzić sobie z tą pozorną sprzecznością? Wystarczy wykorzystać fakt, że czyste funkcje wirtualne muszą być ponownie deklarowane w podklasach, ale mogą także zawierać własne implementacje. Oto sposób, w jaki możemy wykorzystać w hierarchii klas reprezentujących samoloty możliwość definiowania czystej funkcji wirtualnej:

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;
  ...
};

void Airplane::fly(const Airport& destination)
{
  domyślny kod modelujący przelot samolotu
  do danego celu
}

class ModelA: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }
  ...
};
```

```
class ModelB: public Airplane {
public:
  virtual void fly(const Airport& destination)
  { Airplane::fly(destination); }
  ...
};

class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);
  ...
};

void ModelC::fly(const Airport& destination)
{
  kod modelujący przelot samolotu typu ModelC do danego celu
}
```

Powyższy schemat niemal nie różni się od wcześniejszego projektu z wyjątkiem ciała czystej funkcji wirtualnej `Airplane::fly`, która zastąpiła wykorzystywaną wcześniej niezależną funkcję `Airplane::defaultFly`. W rzeczywistości funkcja `fly` została rozbita na dwa najważniejsze elementy. Pierwszym z nich jest deklaracja określająca jej interfejs (który *musi* być wykorzystywany przez klasy potomne), natomiast drugim jest definicja określająca domyślne zachowanie funkcji (która *może* być wykorzystana w klasach domyślnych, ale tylko na wyraźne żądanie ich projektantów). Łącząc funkcje `fly` i `defaultFly`, straciliśmy jednak możliwość nadawania im różnych ograniczeń dostępu — kod, który wcześniej był chroniony (funkcja `defaultFly` była zadeklarowana w bloku `protected`) będzie teraz publiczny (ponieważ znajduje się w zadeklarowanej w bloku `public` funkcji `fly`).

Wróćmy do należącej do klasy `Shape` niewirtualnej funkcji `objectID`. Kiedy funkcja składowa jest niewirtualna, w zamierzeniu nie powinna zachowywać się w klasach potomnych inaczej niż w klasie bazowej. W rzeczywistości niewirtualne funkcje składowe opisują zachowanie *niezależne od specjalizacji*, ponieważ implementacja funkcji nie powinna ulegać żadnym zmianom, niezależnie od specjalizacji kolejnych poziomów w hierarchii klas potomnych. Oto płynący z tego wniosek:

- Celem deklarowania niewirtualnej funkcji jest otrzymanie klas potomnych dziedziczących zarówno *interfejs, jak i wymaganą implementację* tej funkcji.

Możesz pomyśleć, że deklaracja funkcji `Shape::objectID` oznacza: „każdy obiekt klasy `Shape` zawiera funkcję zwracającą identyfikator obiektu, który zawsze jest wyznaczany w ten sam sposób (opisany w definicji funkcji `Shape::objectID`), którego żadna klasa potomna nie powinna próbować modyfikować”. Ponieważ niewirtualna funkcja opisuje zachowanie *niezależne* od specjalizacji, nigdy nie powinna być ponownie deklarowana w żadnej podklasie (to zagadnienie szczegółowo omówiłem w sposobie 37.).

Różnice pomiędzy deklaracjami czystych funkcji wirtualnych, prostych funkcji wirtualnych oraz funkcji niewirtualnych umożliwiają dokładne precyzowanie właściwych dla danego mechanizmów dziedziczenia tych funkcji przez klasy potomne: dziedziczenia samego interfejsu, dziedziczenia interfejsu i domyślnej implementacji lub dziedziczenia interfejsu i wymaganej implementacji. Ponieważ wymienione różne

typy deklaracji oznaczają zupełnie inne mechanizmy dziedziczenia, podczas deklarowania funkcji składowych musisz bardzo ostrożnie wybrać jedną z omawianych metod.

Pierwszym popularnym błędem jest deklarowanie wszystkich funkcji jako niewirtualnych. Eliminujemy w ten sposób możliwość specjalizacji klas potomnych; szczególnie kłopotliwe są w tym przypadku także niewirtualne destruktory (patrz sposób 14.). Jest to oczywiście dobre rozwiązanie dla klas, które w założeniu nie będą wykorzystywane w charakterze klas bazowych. W takim przypadku, zastosowanie zbioru wyłącznie niewirtualnych funkcji składowych jest całkowicie poprawne. Zbyt często jednak wynika to wyłącznie z braku wiedzy na temat różnić pomiędzy funkcjami wirtualnymi a niewirtualnymi lub nieuzasadnionych obaw odnośnie wydajności funkcji wirtualnych. Należy więc pamiętać o fakcie, że niemal wszystkie klasy, które w przyszłości mają być wykorzystane jako klasy bazowe, powinny zawierać funkcje wirtualne (ponownie patrz sposób 14.).

Jeśli obawiasz się kosztów związanych z funkcjami wirtualnymi, pozwól, że przypomnę Ci o regule 80-20 (patrz także sposób 33.), która mówi, że 80 procent czasu działania programu jest poświęcona wykonywaniu 20 procent jego kodu. Wspomniana reguła jest istotna, ponieważ oznacza, że średnio 80 procent naszych wywołań funkcji może być wirtualnych i będzie to miało niemal niezauważalny wpływ na całkowitą wydajność naszego programu. Zanim więc zaczniesz się martwić, czy możesz sobie pozwolić na koszty związane z wykorzystaniem funkcji wirtualnych, upewnij się, czy Twoje rozważania dotyczą tych 20 procent programu, gdzie decyzja będzie miała istotny wpływ na wydajność całego programu.

Innym powszechnym problemem jest deklarowanie *wszystkich* funkcji jako wirtualne. Niekiedy jest to oczywiście właściwe rozwiązanie — np. w przypadku klas protokołu (patrz sposób 34.). Może jednak świadczyć także o zwykłej niewiedzy projektanta klasy. Niektóre deklaracje funkcji *nie* powinny umożliwiać ponownego ich definiowania w klasach potomnych — w takich przypadkach jedynym sposobem osiągnięcia tego celu jest deklarowanie tych funkcji jako niewirtualnych. Nie ma przecież najmniejszego sensu udostępnianie innym programistom klas, które mają być dziedziczone przez inne klasy i których wszystkie funkcje składowe będą ponownie definiowane. Pamiętaj, że jeśli masz klasę bazową `B`, klasę potomną `D` oraz funkcję składową `mf`, wówczas każde z poniższych wywołań funkcji `mf` *musi* być prawidłowe:

```
D *pd = new D;
b *pb = pd;

pb->mf();              // wywołuje funkcję mf za pomocą
                       // wskaźnika do klasy bazowej

pd->mf();              // wywołuje funkcję mf za pomocą
                       // wskaźnika do klasy potomnej
```

Niekiedy musisz zadeklarować funkcję `mf` jako niewirtualną, by upewnić się, że wszystko będzie działało zgodnie z Twoimi oczekiwaniami (patrz sposób 37.). Jeśli działanie funkcji powinno być niezależne od specjalizacji, nie obawiaj się takiego rozwiązania.

Sposób 37. Nigdy nie definiuj ponownie dziedziczonych funkcji niewirtualnych

Istnieją dwa podejścia do tego problemu: teoretyczne i pragmatyczne. Zacznijmy od podejścia pragmatycznego (teoretycy są w końcu przyzwyczajeni do cierpliwego czekania).

Przypuśćmy, że powiem Ci, że klasa `D` publicznie dziedziczy po klasie `B` i istnieje publiczna funkcja składowa `mf` zdefiniowana w klasie `B`. Parametry i wartość zwracane przez funkcję `mf` są dla nas na tym etapie nieistotne, załóżmy więc, że mają postać `void`. Innymi słowy, możemy to wyrazić w następujący sposób:

```
class B {
public:
  void mf();
  ...
};

class D: public B { ... };
```

Nawet gdybyśmy nic nie wiedzieli o `B`, `D` i `mf`, mając dany obiekt `x` klasy `D`:

```
D x;                    // x jest obiektem typu D
```

bylibyśmy bardzo zaskoczeni, gdyby instrukcje:

```
B *pB = &x;             // otrzymuje wskaźnik do x
pB->mf();               // wywołuje funkcję mf za pomocą wskaźnika
```

powodowały inne działanie, niż instrukcje:

```
D *pD = &x;             // otrzymuje wskaźnik do x
pD->mf();               // wywołuje funkcję mf za pomocą wskaźnika
```

Wynika to z faktu, że w obu przypadkach wywołujemy funkcję składową `mf` dla obiektu `x`. Ponieważ w obu przypadkach jest to ta sama funkcja i ten sam obiekt, efekt wywołania powinien być identyczny, prawda?

Tak, powinien, ale nie jest. W szczególności, rezultaty wywołania będą inne, jeśli `mf` będzie funkcją niewirtualną, a klasa `D` będzie zawierała definicję własnej wersji tej funkcji:

```
class D: public B {
public:
  void mf();            // ukrywa definicję B::mf; patrz sposób 50.
  ...
};

pB->mf();               // wywołuje funkcję B::mf
pD->mf();               // wywołuje funkcję D::mf
```

Powodem takiego dwulicowego zachowania jest fakt, że *niewirtualne* funkcje `B::mf` i `D::mf` są wiązane statycznie (patrz sposób 38.). Oznacza to, że ponieważ zmienna `pB` została zadeklarowana jako wskaźnik do `B`, niewirtualne funkcje wywoływane za pośrednictwem tej zmiennej *zawsze* będą tymi zdefiniowanymi dla klasy `B`, nawet jeśli `pB` wskazuje na obiekt klasy pochodnej względem `B` (jak w powyższym przykładzie).

Z drugiej strony, funkcje *wirtualne* są wiązane dynamicznie (ponownie patrz sposób 38.), co oznacza, że opisywany problem ich nie dotyczy. Gdyby `mf` była funkcją wirtualną, jej wywołanie (niezależnie od tego, czy z wykorzystaniem wskaźnika do `pB` czy do `pD`) spowodowałoby wywołanie wersji `D::mf`, ponieważ `pB` i `pD` *w rzeczywistości* wskazują na obiekt klasy `D`.

Należy pamiętać, że jeśli tworzymy klasę `D` i ponownie definiujemy dziedziczoną po klasie `B` niewirtualną funkcję `mf`, obiekty klasy `D` będą się prawdopodobnie okazywały zachowania godne schizofrenika. W szczególności dowolny obiekt klasy `D` może — w odpowiedzi na wywołanie funkcji `mf` — zachowywać się albo jak obiekt klasy `B`, albo jak obiekt klasy `D`; czynnikiem rozstrzygającym nie będzie tutaj sam obiekt, ale zadeklarowany typ wskazującego na ten obiekt wskaźnika. Równie zdumiewające zachowanie zaprezentowałyby w takim przypadku referencje do obiektów.

To już wszystkie argumenty wysuwane przez praktyków. Chcesz pewnie teraz poznać jakieś teoretyczne uzasadnienie, dlaczego nie należy ponownie definiować dziedziczonych funkcji niewirtualnych. Wyjaśnię to z przyjemnością.

W sposobie 35. pokazałem, że publiczne dziedziczenie oznacza w rzeczywistości relację „jest"; w sposobie 36. opisałem, dlaczego deklarowanie niewirtualnych funkcji w klasie powoduje niezależność od ewentualnych specjalizacji tej klasy. Jeśli właściwie wykorzystasz wnioski wyniesione z tych sposobów podczas projektowania klas `B` i `D` oraz podczas tworzenia niewirtualnej funkcji składowej `B::mf`, wówczas:

- Wszystkie funkcje, które można stosować dla obiektów klasy `B`, można stosować także dla obiektów klasy `D`, ponieważ każdy obiekt klasy `D` „jest" obiektem klasy `B`.
- Podklasy klasy `B` muszą dziedziczyć zarówno interfejs, *jak i* implementację funkcji `mf`, ponieważ funkcja ta została zadeklarowana w klasie `B` jako niewirtualna.

Jeśli w klasie `D` ponownie zdefiniujemy teraz funkcję `mf`, w naszym projekcie powstanie sprzeczność. Jeśli klasa `D` *faktycznie* potrzebuje własnej implementacji funkcji `mf`, która będzie się różniła od implementacji dziedziczonej po klasie `B`, oraz jeśli każdy obiekt klasy `B` (niezależnie od poziomu specjalizacji) *rzeczywiście* musi wykorzystywać implementacji tej funkcji z klasy `B`, wówczas stwierdzenie, że `D` „jest" `B` jest zwyczajnie nieprawdziwe. Klasa `D` nie powinna w takim przypadku publicznie dziedziczyć po klasie `B`. Z drugiej strony, jeśli `D` *naprawdę* musi publicznie dziedziczyć po `B` oraz jeśli `D` *naprawdę* musi implementować funkcję `mf` inaczej, niż implementuje ją klasa `B`, wówczas nieprawdą jest, że `mf` odzwierciedla niezależność od specjalizacji klasy `B`. W takim przypadku funkcja `mf` powinna zostać zadeklarowana jako wirtualna. Wreszcie, jeśli każdy obiekt klasy `D` *naprawdę* musi być w relacji „jest" z obiektem klasy `B` oraz jeśli funkcja `mf` rzeczywiście reprezentuje niezależność od specjalizacji klasy `B`, wówczas klasa `D` nie powinna potrzebować własnej implementacji funkcji `mf` i jej projektant nie powinien więc podejmować podobnych prób.

Niezależnie od tego, który argument najbardziej pasuje do naszej sytuacji, oczywiste jest, że ponowne definiowanie dziedziczonych funkcji niewirtualnych jest całkowicie pozbawione sensu.

Sposób 38. Nigdy nie definiuj ponownie dziedziczonej domyślnej wartości parametru

Spróbujmy uprościć nasze rozważania od samego początku. Domyślny parametr może istnieć wyłącznie jako część funkcji, a nasze klasy mogą dziedziczyć tylko dwa rodzaje funkcji — wirtualne i niewirtualne. Jedynym sposobem ponownego zdefiniowania wartości domyślnej parametru jest więc ponowne zdefiniowanie całej dziedziczonej funkcji. Ponowne definiowanie dziedziczonej niewirtualnej funkcji jest jednak zawsze błędne (patrz sposób 37.), możemy więc od razu ograniczyć naszą analizę do sytuacji, w której dziedziczymy funkcję *wirtualną* z domyślną wartością parametru.

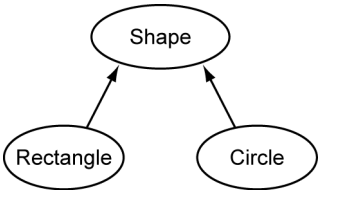
W takim przypadku wyjaśnienie sensu umieszczania tego sposobu w książce jest bardzo proste — funkcje wirtualne są wiązane dynamicznie, ale domyślne wartości ich parametrów są wiązane statycznie.

Co to oznacza? Być może nie posługujesz się biegle najnowszym żargonem związanym z programowaniem obiektowym lub zwyczajnie zapomniałeś, jakie są różnice pomiędzy wiązaniem statycznym a wiązaniem dynamicznym. Przypomnijmy więc sobie, o co tak naprawdę chodzi.

Typem statycznym obiektu jest ten typ, który wykorzystujemy w deklaracji obiektu w kodzie programu. Przeanalizujmy poniższą hierarchię klas:

```
class ShapeColor { RED, GREEN, BLUE };

// klasa reprezentująca figury geometryczne
class Shape {
public:
  // wszystkie figury muszą udostępniać rysujące je funkcje
  virtual void draw(ShapeColor color = RED) const = 0;
  ...
};

class Rectangle: public Shape {
public:
  // zwróć uwagę na inna domyślną wartość parametru - źle!
  virtual void draw(ShapeColor color = GREEN) const;
  ...
};

class Circle: public Shape {
public:
  virtual void draw(ShapeColor color) const;
  ...
};
```

Powyższą hierarchię można przedstawić graficznie:



Rozważmy teraz poniższe wskaźniki:

```
Shape *ps;                    // statyczny typ = Shape*
Shape *pc = new Circle;       // statyczny typ = Shape*
Shape *pr = new Rectangle;    // statyczny typ = Shape*
```

W powyższym przykładzie `ps`, `pc` i `pr` są zadeklarowane jako zmienne typu wskaźnikowego do obiektów klasy `Shape`, zatem wszystkie należą do typu statycznego. Zauważ, że nie ma w tym przypadku znaczenia, na co wymienione zmienne faktycznie wskazują — ich statycznym typem jest `Shape*`.

Typ dynamiczny obiektu zależy od typu obiektu aktualnie przez niego wskazywanego. Oznacza to, że od dynamicznego typu zależy zachowanie obiektu. W powyższym przykładzie typem dynamicznym zmiennej `pc` jest `Circle*`, zaś typem dynamicznym zmiennej `pr` jest `Rectangle*`. Inaczej jest w przypadku zmiennej `ps`, która nie ma dynamicznego typu, ponieważ w rzeczywistości nie wskazuje na żaden obiekt.

Typy dynamiczne (jak sama nazwa wskazuje) mogą się zmieniać w czasie wykonywania programu, tego rodzaju zmiany odbywają się zazwyczaj na skutek wykonania operacji przypisania:

```
ps = pc;                      // typem dynamicznym wskaźnika ps
                              // jest teraz Circle*

ps = pr;                      // typem dynamicznym wskaźnika ps
                              // jest teraz Rectangle*
```

Wirtualne funkcje są *wiązane dynamicznie*, co oznacza, że konkretna wywoływana funkcja zależy od dynamicznego typu obiektu wykorzystywanego do jej wywołania:

```
pc->draw(RED);                // wywołuje funkcję Circle::draw(RED)
pr->draw(RED);                // wywołuje funkcję Rectangle::draw(RED)
```

Uważasz pewnie, że nie ma powodów wracać do tego tematu — z pewnością rozumiesz już znaczenie funkcji wirtualnych. Problemy ujawniają się dopiero w momencie, gdy analizujemy funkcje wirtualne z domyślnymi wartościami parametrów, ponieważ — jak już wspomniałem — funkcje wirtualne są wiązane dynamicznie, a domyślne parametry C++ wiąże statycznie. Oznacza to, że możesz wywołać wirtualną funkcję zdefiniowaną w *klasie potomnej*, ale z domyślną wartością parametru z *klasy bazowej*:

```
pr->draw();                   // wywołuje Rectangle::draw(RED);
```

W tym przypadku typem dynamicznym zmiennej wskaźnikowej `pr` jest `Rectangle*`, zatem zostanie wywołana (zgodnie z naszymi oczekiwaniami) funkcja wirtualna `draw` zdefiniowana w klasie `Rectangle`. Domyślną wartością parametru funkcji `Rectangle::draw` jest `GREEN`. Ponieważ jednak typem statycznym zmiennej `pr` jest `Shape*`, domyślna wartość parametru dla tego wywołania funkcji będzie pochodziła z definicji klasy `Shape`, a nie `Rectangle`! Otrzymujemy w efekcie wywołanie składające się z nieoczekiwanej kombinacji dwóch deklaracji funkcji `draw` — z klas `Shape` i `Rectangle`. Możesz mi wierzyć, tworzenie oprogramowania zachowującego się w taki sposób jest ostatnią rzeczą, którą chciałbyś robić; jeśli to Cię nie przekonuje, zaufaj mi — na pewno z takiego zachowania Twojego oprogramowania nie będą zadowoleni Twoi *klienci*.

Nie muszę chyba dodawać, że nie ma w tym przypadku żadnego znaczenia fakt, że `ps`, `pc` i `pr` są wskaźnikami. Gdyby były referencjami, problem nadal by istniał. Jedynym istotnym źródłem naszego problemu jest to, że `draw` jest funkcją wirtualną i jedna z jej domyślnych wartości parametrów została ponownie zdefiniowana w podklasie.

Dlaczego C++ umożliwia tworzenie oprogramowania zachowującego się w tak nienaturalny sposób? Odpowiedzią jest efektywność wykonywania programów. Gdyby domyślne wartości parametrów były wiązane dynamicznie, kompilatory musiałyby stosować dodatkowe mechanizmy określania właściwych domyślnych wartości parametrów funkcji wirtualnych podczas wykonywania programów, co prowadziłoby do spowolnienia i komplikacji stosowanego obecnie mechanizmu ich wyznaczania w czasie kompilacji. Decyzję podjęto więc wyłącznie z myślą o szybkości i prostocie implementacji. Efektem jest wydajny mechanizm wykonywania programów, ale także — jeśli nie będziesz stosował zaleceń zawartych w tym sposobie — potencjalne nieporozumienia.

Sposób 39. Unikaj rzutowania w dół hierarchii dziedziczenia

W dzisiejszych niespokojnych czasach warto mieć na oku poczynania instytucji finansowych, rozważmy więc klasę protokołu (patrz sposób 34.) dla kont bankowych:

```
class Person { ... };

class BankAccount {
public:
  BankAccount(const Person *primaryOwner,
              const Person *jointOwner);
  virtual ~BankAccount();

  virtual void makeDeposit(double amount) = 0;
  virtual void makeWithdrawal(double amount) = 0;

  virtual double balance() const = 0;
  ...
};
```

Wiele banków przedstawia dzisiaj swoim klientom niezwykle szeroką ofertę typów kont bankowych, załóżmy jednak (dla uproszczenia), że istnieje tylko jeden typ konta bankowego, zwykłe konto oszczędnościowe:

```
class SavingsAccount: public BankAccount {
public:
  SavingsAccount(const Person *primaryOwner,
                 const Person *jointOwner);
  ~SavingsAccount();

  void creditInterest();       // dodaj odsetki do konta
  ...
};
```

Nie jest to może zbyt zaawansowane konto oszczędnościowe, jest jednak w zupełności wystarczające dla naszych rozważań.

Bank prawdopodobnie przechowuje listę wszystkich swoich kont, która może być zaimplementowana za pomocą szablonu klasy `list` ze standardowej biblioteki C++ (patrz sposób 49.). Przypuśćmy, że w naszym banku taka lista nosi nazwę `allAccounts`:

```
list<BankAccount*> allAccounts;      // wszystkie konta obsługiwane
                                     // przez dany bank
```

Jak wszystkie standardowe pojemniki, listy przechowują jedynie *kopie* umieszczanych w nich obiektów, zatem, aby uniknąć przechowywania wielu kopii poszczególnych obiektów klasy `BankAccount`, programiści zdecydowali, że lista `allAccounts` powinna składać się jedynie ze *wskaźników* do tych obiektów, a nie samych obiektów reprezentujących konta.

Przypuśćmy, że naszym zadaniem jest kolejne przejście przez wszystkie konta i doliczenie do nich należnych odsetek. Możemy zrealizować tę usługę w następujący sposób:

```
// pętla nie zostanie skompilowana (jeśli nigdy wcześniej nie miałeś
// do czynienia z kodem wykorzystującym iteratory, patrz niżej)
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
  (*p)->creditInterest();      // błąd!
}
```

Nasze kompilatory szybko zasygnalizują, że lista `allAccounts` zawiera wskaźniki do obiektów klasy `BankAccount`, a nie obiektów klasy `SavingsAccount`, zatem w każdej kolejnej iteracji zmienna `p` będzie wskazywała na obiekt klasy `BankAccount`. Oznacza to, że wywołanie funkcji `creditInterest` jest nieprawidłowe, ponieważ została ona zadeklarowana wyłącznie dla obiektów klasy `SavingsAccount`, a nie `BankAccount`.

Jeśli wiersz `list<BankAccount*>::iterator p = allAccounts.begin()` jest dla Ciebie niezrozumiały i nie przypomina Ci kodu C++, z którym miałeś do czynienia do tej pory, oznacza to, że prawdopodobnie nie miałeś wcześniej przyjemności korzystać z szablonów klas pojemnikowych ze standardowej biblioteki C++. Tę część biblioteki nazywa się często Standardową Biblioteką Szablonów (ang. *Standard Template Library — STL*), więcej informacji na jej temat znajdziesz w sposobie 49. Na tym etapie wystarczy Ci

wiedza, że zmienna `p` zachowuje się jak wskaźnik wskazujący na przeglądane w pętli kolejne elementy listy `allAccounts` (od jej początku do końca). Oznacza to, że zmienna `p` jest traktowana tak, jakby jej typem był `BankAccount**`, a elementy przeglądanej listy były przechowywane w tablicy.

Powyższy kod nie zostanie niestety skompilowany. Wiemy oczywiście, że lista `allAccounts` została zdefiniowana jako pojemnik na wskaźniki typu `BankAccount*`, ale *mamy także świadomość*, że w powyższej pętli faktycznie przechowuje wskaźniki typu `SavingsAccount*`, ponieważ `SavingsAccount` jest jedyną klasą, dla której możemy tworzyć obiekty. Głupie kompilatory! Zdecydowaliśmy się przekazać im wiedzę, która jest dla nas zupełnie oczywista i okazało się, że są zbyt tępe, by zauważyć, że lista `allAccounts` przechowuje w rzeczywistości wskaźniki typu `SavingsAccount*`:

```
// pętla zostanie skompilowana, chociaż nie jest to dobre rozwiązanie
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
  static_cast<SavingsAccount*>(*p)->creditInterest();
}
```

Rozwiązaliśmy wszystkie nasze problemy! Zrobiliśmy to przejrzyście, elegancko i zwięźle — wystarczyło jedno proste rzutowanie. Wiemy, jakiego typu wskaźniki faktycznie znajdują się na liście `allAccounts`, nasze ogłupiałe kompilatory tego nie wiedzą, więc zastosowaliśmy rzutowanie, by przekazać im naszą wiedzę. Czy można znaleźć bardziej logiczne rozwiązanie?

Chciałbym w tym momencie przedstawić pewną biblijną analogię. Operacje rzutowania są dla programistów C++ jak jabłko dla biblijnej Ewy.

Zaprezentowany w powyższym kodzie rodzaj rzutowania (ze wskaźnika do klasy bazowej do wskaźnika do klasy potomnej) nosi nazwę *rzutowania w dół*, ponieważ rzutujemy w dół hierarchii dziedziczenia. W powyższym przykładzie operacja rzutowania w dół zadziałała, jednak — jak się za chwilę przekonasz — takie rozwiązanie prowadzi do ogromnych problemów podczas konserwacji oprogramowania.

Wróćmy jednak do naszego banku. Zachęcony sukcesem, jakim było rozwiązanie problemu kont oszczędnościowych, bank postanawia zaoferować swoim klientom także konta czekowe. Co więcej, załóżmy, że do tego typu kont także dolicza się należne odsetki (podobnie, jak w przypadku kont oszczędnościowych):

```
class CheckingAccount: public BankAccount {
public:
  void creditInterest();       // dodaj odsetki do konta
  ...
};
```

Nie muszę chyba dodawać, że lista `allAccounts` będzie teraz zawierała wskaźniki zarówno do obiektów reprezentujących konta oszczędnościowe, jak i do tych reprezentujących konta czekowe. Okazuje się, że stworzona przed chwilą pętla naliczająca odsetki przestaje działać.

Pierwszy problem polega na tym, że pętla nadal będzie poprawnie kompilowana, mimo że nie wprowadziliśmy jeszcze zmian uwzględniających istnienie obiektów nowej klasy `CheckingAccount`. Wynika to z faktu, że nasze kompilatory nadal będą pochopnie zakładały, że kiedy sygnalizujemy (za pomocą `static_cast`), że `*p` w rzeczywistości wskazuje na `SavingsAccount*`, tak jest w istocie. W końcu to do nas należy przewidywanie skutków wykonywania poszczególnych instrukcji. Drugi problem wynika z typowego sposobu radzenia sobie z pierwszym problemem, co zwykle skutkuje tworzeniem podobnego kodu:

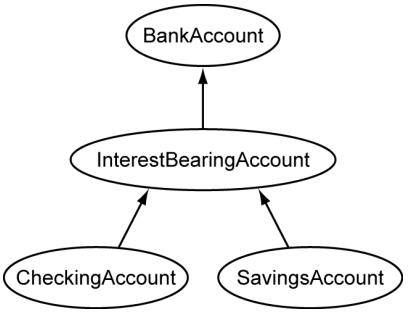
```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
  if (*p wskazuje na obiekt klasy SavingsAccount)
    static_cast<SavingsAccount*>(*p)->creditInterest();
  else
    static_cast<CheckingAccount*>(*p)->creditInterest();
}
```

Jeśli kiedykolwiek stwierdzisz, że Twój kod zawiera instrukcję warunkową w postaci: „jeśli obiekt jest typu T1, zrób coś, jeśli jednak jest typu T2, zrób coś innego”, natychmiast uderz się w pierś. Tego rodzaju instrukcje są nienaturalne dla języka C++; podobną strategię można stosować w C lub Pascalu, ale nigdy w C++, w którym możemy przecież użyć funkcji wirtualnych.

Pamiętaj, że zastosowanie funkcji wirtualnych sprawia, że za zapewnianie prawidłowych wywołań funkcji — w zależności od typu wykorzystywanego obiektu — odpowiadają *kompilatory*. Nie zaśmiecaj więc swojego kodu instrukcjami warunkowymi ani przełącznikami; zamiast tego wykorzystuj możliwości swoich kompilatorów. Oto przykład:

```
class BankAccount { ... };     // jak wyżej

// nowa klasa reprezentująca konta przynoszące klientom odsetki
class InterestBearingAccount: public BankAccount {
public:
  virtual void creditInterest() = 0;
  ...
};

class SavingsAccount: public InterestBearingAccount {
  ...                          // jak wyżej
};

class CheckingAccount: public InterestBearingAccount {
  ...                          // jak wyżej
};
```

Powyższą hierarchię można przedstawić graficznie:



Ponieważ zarówno konta oszczędnościowe, jak i konta czekowe wymagają naliczania odsetek, naturalnym rozwiązaniem jest przeniesienie wspólnej operacji na wyższy poziom hierarchii klas — do wspólnej klasy bazowej. Jednak przy założeniu, że nie wszystkie konta w danym banku muszą mieć naliczane odsetki (jak wynika z moich doświadczeń, takie założenie jest sensowne), nie możemy przenieść wspomnianej operacji do najwyższej (w hierarchii) klasy `BankAccount`. Wprowadziliśmy więc nową podklasę klasy `BankAccount`, którą nazwaliśmy `InterestBearingAccount`, i która jest klasą bazową dla klas `SavingsAccount` i `CheckingAccount`.

Wymaganie, by zarówno dla konta oszczędnościowego, jak i dla konta czekowego naliczać odsetki, uwzględniliśmy, deklarując w klasie `InterestBearingAccount` czystą funkcję wirtualną `creditInterest`, która zostanie przypuszczalnie zdefiniowana w podklasach `SavingsAccount` i `CheckingAccount`.

Nowa hierarchia klas umożliwia nam napisanie omawianej wcześniej pętli od początku:

```
// rozwiązanie lepsze, ale nadal niedoskonałe
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
  static_cast<InterestBearingAccount*>(*p)->creditInterest();
}
```

Mimo że powyższa pętla nadal zawiera skrytykowane wcześniej rzutowanie, zaproponowane rozwiązanie jest znacznie lepsze od omawianych do tej pory, ponieważ będzie poprawnie funkcjonowało nawet po dodaniu do naszej aplikacji nowych podklas do klasy `InterestBearingAccount`.

Aby całkowicie pozbyć się rzutowania, musimy wprowadzić do naszego projektu kilka dodatkowych modyfikacji. Jedną z nich jest doprecyzowanie specyfikacji wykorzystywanej listy kont. Gdybyśmy mogli wykorzystać listę obiektów klasy `InterestBearingAccount`, zamiast obiektów klasy `BankAccount`, rozwiązanie byłoby trywialne:

```
// wszystkie obsługiwane przez bank konta z odsetkami
list<InterestBearingAccount*> allIBAccounts;

// pętla zostanie skompilowana i będzie działała prawidłowo
for (list<InterestBearingAccount*>::iterator p = allIBAccounts.begin();
     p != allIBAccounts.end();
     ++p) {
  (p*)->creditInterest();
}
```

Jeśli tworzenie bardziej specjalizowanej listy nie jest brane pod uwagę, można stwierdzić, że operację `creditInterest` można stosować dla wszystkich kont bankowych; jednak w przypadku kont, dla których nie nalicza się odsetek, wspomniana operacja jest pusta. To samo możemy wyrazić za pomocą poniższego fragmentu kodu:

```
class BankAccount {
public:
  virtual void creditInterest() {}
  ...
};

class SavingsAccount: public BankAccount { ... };

class CheckingAccount: public BankAccount { ... };

list<BankAccount*> allAccounts;

// patrz, nie ma rzutowania!
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
  (p*)->creditInterest();
}
```

Zauważ, że wirtualna funkcja `BankAccount::creditInterest` udostępnia pustą domyślną implementację. Jest to wygodny sposób definiowania funkcji, która domyślnie jest operacją pustą, może jednak w przyszłości doprowadzić do nieprzewidywalnych trudności. Omówienie przyczyn takiego niebezpieczeństwa i sposobów jego eliminowania znajdziesz w sposobie 36. Zauważ także, że funkcja `creditInterest` jest (niejawnie) wbudowana. Oczywiście nie ma w tym nic złego, jednak z uwagi na fakt, że omawiana funkcja jest także wirtualna, dyrektywa `inline` będzie prawdopodobnie zignorowana przez kompilator (patrz sposób 33.).

Jak się przekonałeś, rzutowanie w dół hierarchii klas może być eliminowane na wiele sposobów. Najlepszym z nich jest zastąpienie tego typu operacji wywołaniami funkcji wirtualnych — można wówczas zdefiniować domyślne implementacje tych funkcji jako operacje puste, co pozwoli na ich prawidłową obsługę przez obiekty klasy, dla których dane działania nie mają sensu. Drugą metodą jest uściślenie wykorzystywanych typów, co pozwala wyeliminować nieporozumienia wynikające z rozbieżności pomiędzy typami deklarowanymi a typami reprezentowanymi przez używane obiekty w rzeczywistości. Wysiłek związany z eliminowaniem rzutowania w dół nie idzie na marne, ponieważ kod zawierający operacje tego typu jest brzydki i może być źródłem błędów, jest także trudny do zrozumienia, rozszerzania i konserwacji.

To, co napisałem do tej pory, jest prawdą i tylko prawdą. Nie jest jednak całą prawdą. Istnieją bowiem sytuacje, w których *naprawdę* musisz wykonać operację rzutowania w dół.

Przykładowo, przypuśćmy, że mamy do czynienia z rozważaną na początku tego sposobu sytuacją, w której lista `allAccounts` przechowuje wskaźniki do obiektów klasy `BankAccount`, funkcja `creditInterest` jest zdefiniowana wyłącznie dla obiektów klasy `SavingsAccount` i musimy napisać pętlę naliczającą odsetki dla wszystkich kont. Przypuśćmy także, że wszystkie wspomniane elementy są poza naszą kontrolą, co

oznacza, że nie możemy modyfikować definicji `BankAccount`, `SavingsAccount` ani `allAccounts` (jest to sytuacja charakterystyczna, kiedy korzystamy z elementów zdefiniowanych w bibliotece, do której mamy dostęp tylko do odczytu). W takim przypadku *musielibyśmy* zastosować rzutowanie w dół, niezależnie od katastrofalnych skutków takiego posunięcia.

Niezależnie od tego istnieje lepszy sposób niż stosowane wcześniej surowe rzutowanie. Tym sposobem jest coś, co nazywamy „bezpiecznym rzutowaniem w dół" — wymaga zastosowania zaimplementowanego w C++ operatora `dynamic_cast`. Kiedy stosujemy ten operator dla wskaźnika, wykonywane jest rzutowanie, które — w przypadku powodzenia (tzn. jeśli dynamiczny typ wskaźnika, patrz sposób 38., jest zgodny z typem, do którego jest rzutowany) — powoduje zwrócenie poprawnego wskaźnika nowego typu; w przypadku niepowodzenia operacji, zwracany jest wskaźnik pusty.

Oto przykład z kontami bankowymi po dodaniu mechanizmu bezpiecznego rzutowania w dół:

```
class BankAccount { ... };          // jak na początku tego sposobu

class SavingsAccount:
  public BankAccount { ... };       // jak wyżej

class CheckingAccount:              // jak wyżej
  public BankAccount { ... };

list<BankAccount*> allAccounts;     // wygląda znajomo...

void error(const string& msg);      // funkcja obsługująca błędy (patrz niżej)

// cóż, przynajmniej rzutowanie jest teraz bezpieczne
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {
  // próba bezpiecznego rzutowania w dół wskaźnika *p do
  // SavingsAccount; informacje na temat definicji psa
  // znajdziesz poniżej
  if (SavingsAccount *psa =
        dynamic_cast<SavingsAccount*>(*p)) {
    psa->creditInterest();
  }
  // próba bezpiecznego rzutowania w dół do CheckingAccount
  else if (CheckingAccount *pca =
        dynamic_cast<CheckingAccount*>(*p)) {
    pca->creditInterest();
  }
  // niestety, nieznany typ konta
  else {
    error("Nieznany typ konta!");
  }
}
```

Powyższy schemat daleki jest od ideału, ale przynajmniej umożliwia wykrywanie operacji rzutowania w dół zakończonych niepowodzeniem, co było niemożliwe, kiedy wykorzystywaliśmy operator `static_cast` zamiast operatora `dynamic_cast`. Zauważ

jednak, że rozsądek nakazuje nam także sprawdzenie przypadku, w którym *wszystkie* operacje rzutowania zakończyły się niepowodzeniem. Realizujemy to zadanie w powyższym kodzie za pomocą ostatniej klauzuli `else`. Taka weryfikacja była zbędna w przypadku wirtualnych funkcji, ponieważ każde wywołanie takiej funkcji musiało dotyczyć *jakiejś* jej wersji. Kiedy jednak decydujemy się na rzutowanie w dół, nasza sytuacja zmienia się diametralnie — kiedy ktoś doda do hierarchii np. nowy typ konta, ale zapomni o aktualizacji powyższego kodu, wszystkie rzutowania w dół zakończą się niepowodzeniem. Dlatego właśnie tak ważna jest obsługa opisywanej sytuacji. Jest bardzo mało prawdopodobne, by wszystkie operacje rzutowania zakończyły się niepowodzeniem, jeśli jednak decydujemy się na rzutowanie w dół, nawet najlepszym programistom może się przytrafić coś niedobrego.

Czy przecierasz z niedowierzania oczy, widząc, jak wyglądają definicje zmiennych w warunkach powyższych instrukcji `if`? Jeśli tak, nie martw się, dobrze widzisz. Możliwość definiowania takich zmiennych została dodana do języka C++ w tym samym momencie, co operator `dynamic_cast`. Możemy dzięki temu pisać elegancki kod, ponieważ wskaźniki `psa` i `pca` w rzeczywistości nie są nam potrzebne aż do momentu wywołania pomyślnie inicjalizującego je operatora `dynamic_cast`. Nowa składnia sprawia, że nie musimy definiować tych zmiennych poza instrukcjami warunkowymi zawierającymi operacje rzutowania (w sposobie 32. wyjaśniłem, dlaczego powinniśmy unikać niepotrzebnych definicji zmiennych). Jeśli Twoje kompilatory jeszcze nie obsługują takiego sposobu definiowania zmiennych, możesz wykorzystać starą metodę:

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  SavingsAccount *psa;         // definicja tradycyjna
  CheckingAccount *pca;        // definicja tradycyjna

  if (psa = dynamic_cast<SavingsAccount*>(*p)) {
    psa->creditInterest();
  }

  else if (pca = dynamic_cast<CheckingAccount*>(*p)) {
    pca->creditInterest();
  }

  else {
    error("Nieznany typ konta!");
  }
}
```

W tym przypadku miejsce definiowania zmiennych podobnych do `psa` i `pca` nie jest oczywiście najważniejsze. Istotne jest coś zupełnie innego — rzutowanie w dół prowadzi do stylu programowania opartego na klauzulach `if-then-else`, co jest najgorszym możliwym rozwiązaniem w ciałach funkcji wirtualnej i powinno być zarezerwowane dla sytuacji, w których naprawdę nie istnieje rozwiązanie alternatywne. Na szczęście, przy odrobinie szczęścia nigdy nie będziesz musiał zmagać się z tak ponurym obliczem programowania.

Sposób 40. Modelując relacje posiadania („ma”) i implementacji z wykorzystaniem, stosuj podział na warstwy

Dzielenie na warstwy jest procesem budowania pewnych klas ponad innymi klasami w taki sposób, że część klas zawiera — w postaci składowych reprezentujących dane — obiekty klas znajdujących się na innych warstwach. Przykładowo:

```
class Address { ... };          // reprezentuje adres zamieszkania

class PhoneNumber { ... };

class Person {
public:
  ...

private:
  string name;                  // obiekt klasy z innej warstwy
  Address address;              // j.w.
  PhoneNumber voiceNumber;      // j.w.
  PhoneNumber faxNumber;        // j.w.
};
```

W powyższym przykładzie klasa `Person` znajduje się w warstwie ponad klasami `string`, `Address` i `PhoneNumber`, ponieważ zawiera składowe reprezentujące dane trzech wymienionych typów. Pojęcia *dzielenia na warstwy* ma wiele synonimów, używa się także określeń *składania*, *zawierania*, *agregacji* i *osadzania* klas.

W sposobie 35. wyjaśniłem, że publiczne dziedziczenie faktycznie oznacza relację „jest”. Inaczej jest w przypadku podziału na warstwy — ten sposób wiązania klas oznacza w rzeczywistości albo relację „ma”, albo relację implementacji z wykorzystaniem.

Zaprezentowana powyżej klasa `Person` jest przykładem relacji „ma”. Obiekty tej klasy zawierają dane o nazwisku, adresie oraz numerze zwykłego telefonu i faksu. Nie możemy powiedzieć, że dana osoba *jest* nazwiskiem lub *jest* adresem. Powiedzielibyśmy raczej, że osoba ta *ma* nazwisko lub *ma* adres etc. Dla większości programistów takie rozróżnienie nie stanowi większego problemu, zatem nieporozumienia odnośnie znaczeń relacji „jest” i „ma” są stosunkowo rzadkie.

Znacznie trudniejsze jest jasne określenie różnicy pomiędzy relacją „jest” a relacją implementacji z wykorzystaniem. Przykładowo przypuśćmy, że potrzebujemy szablonu dla klas reprezentujących zbiory dowolnych obiektów, czyli kolekcje bez powtórzeń. Ponieważ zdolność ponownego wykorzystywania gotowych rozwiązań jest jedną z najbardziej pożądanych cech każdego programisty i ponieważ pewnie zapoznałeś się już z treścią sposobu 49. poświęconego standardowej bibliotece C++, Twoim pierwszym pomysłem będzie wykorzystanie dostępnego w tej bibliotece szablonu `set`. Po co miałbyś pisać nowy szablon, jeśli możesz wykorzystać dobrej jakości szablon napisany przez kogoś innego?

Kiedy zagłębisz się w dokumentacji szablonu `set`, odkryjesz jednak pewne ograniczenia tej struktury, które są nie do przyjęcia w Twojej aplikacji — struktura `set` wymaga, by przechowywane w niej elementy były *całkowicie uporządkowane*, co oznacza, że dla każdej pary obiektów `a` i `b` należących do struktury `set` musi istnieć możliwość określenia, czy `a < b` i czy `b < a`. W przypadku wielu typów spełnienie takiego wymagania jest bardzo łatwe, a posiadanie całkowicie uporządkowanych obiektów pozwala strukturze `set` na zapewnianie bardzo korzystnych warunków w zakresie efektywności działania (więcej szczegółów na temat wydajności struktur udostępnianych przez standardową bibliotekę C++ znajdziesz w sposobie 49.). Potrzebujesz jednak struktury bardziej ogólnej, klasy podobnej do `set`, w której przechowywane obiekty nie muszą spełniać relacji całkowitego porządku, a jedynie takie, dla których można wyznaczyć relację równości (dla których istnieje możliwość określenia, czy `a==b` dla obiektów `a` i `b` tego samego typu). To skromniejsze wymaganie znacznie lepiej pasuje do typów reprezentujących np. kolory. Czy czerwony jest mniejszy od zielonego, czy też zielony jest mniejszy od czerwonego? Wygląda na to, że w takim przypadku będziemy musieli ostatecznie opracować własny szablon.

Ponowne wykorzystywanie gotowych rozwiązań jest nadal świetnym rozwiązaniem. Jeśli jesteś ekspertem w dziedzinie struktur danych, z pewnością wiesz, że niemal nieograniczone możliwości implementowania zbiorów daje (stosunkowo prosta) struktura listy jednokierunkowej. Co więcej, szablon `list` (generujący klasy list jednokierunkowych) *jest dostępny* w standardowej bibliotece C++! Decydujesz się więc na jego (ponowne) wykorzystanie.

W szczególności postanawiasz, że Twój nowy szablon `Set` będzie dziedziczył po szablonie `list`. Oznacza to, że struktura `Set<T>` będzie dziedziczyła po `list<T>`. Twoja implementacja zakłada w końcu, że obiekt `Set` w rzeczywistości *będzie* obiektem `list`. Deklarujesz więc swój szablon `Set` w następujący sposób:

```
// niewłaściwy sposób wykorzystania listy dla szablonu Set
template<class T>
class Set: public list<T> { ... };
```

Być może wszystko na tym etapie wygląda prawidłowo, jednak w rzeczywistości zaprezentowane rozwiązanie zawiera zasadnicze usterki. W sposobie 35. wyjaśniłem, że jeśli D „jest” B, wszystkie poprawne własności klasy B są także poprawne dla klasy D. Obiekt klasy `list` może jednak zawierać duplikaty, zatem jeśli wartość 3051 zostanie wstawiona do struktury `list<int>` dwukrotnie, reprezentowana lista będzie zawierała dwie kopie tej liczby. Inaczej jest w przypadku struktury `Set`, która nie może zawierać duplikatów — jeśli więc wartość 3051 zostanie wstawiona do `Set<int>` dwukrotnie, reprezentowany zbiór i tak będzie zawierał tylko jedną jej kopię. Stwierdzenie, że `Set` „jest” `list` jest więc fałszywe, ponieważ istnieją własności poprawne dla obiektów klasy `list`, które nie są poprawne dla obiektów klasy `Set`.

Ponieważ omawiane dwie klasy są związane relacją inną niż „jest”, modelowanie rzeczywistej relacji nie powinno się opierać na mechanizmie publicznego dziedziczenia. Właściwym rozwiązaniem jest stwierdzenie, że obiekt klasy `Set` może być *implementowany z wykorzystaniem* obiektu klasy `list`:

```
// właściwy sposób wykorzystania struktury list dla Set
template<class T>
class Set {
public:
  bool member(const T& item) const;

  void insert(const T& item);
  void remove(const T& item);

  int cardinality() const;

private:
  list<T> rep;                // reprezentacja zbioru
};
```

Funkcje składowe klasy `Set` mogą w dużej mierze opierać się na funkcjonalności udostępnianej zarówno przez szablon `list`, jak i innych części standardowej biblioteki C++, zatem implementacja powyższej klasy nie jest trudna do napisania ani do późniejszego zrozumienia:

```
template<class T>
bool Set<T>::member(const T& item) const
{ return find(rep.begin(), rep.end(), item) != rep.end(); }

template<class T>
void Set<T>::insert(const T& item)
{ if (!member(item)) rep.push_back(item); }

template<class T>
void Set<T>::remove(const T& item)
{
  list<T>::iterator it = find(rep.begin(), rep.end(), item);
  if (it != rep.end()) rep.erase(it);
}

template<class T>
void Set<T>::cardinality() const
{ return rep.size(); }
```

Powyższe funkcje są na tyle proste, że warto rozważyć ich wbudowanie, chociaż zdaję sobie sprawę, że przed podjęciem takiej decyzji powinieneś przypomnieć sobie nasze rozważania ze sposobu 33. (wykorzystane w powyższym kodzie funkcje `find`, `begin`, `end`, `push_back` etc. są częścią udostępnianego przez standardową bibliotekę C++ modelu dla szablonów generujących pojemniki podobne do `list` — ogólny opis tego modelu znajdziesz w sposobie 49.).

Musimy także pamiętać, że interfejs klasy `Set` nie spełnia wymagań stawianych przed interfejsem kompletnym i minimalnym (patrz sposób 18.). Głównym zaniedbaniem w zakresie kompletności jest brak funkcjonalności obsługującej przechodzenie przez kolejne obiekty przechowywane w zbiorze, co w wielu programach może być niezbędne (i jest oferowane przez wszystkie podobne struktury standardowej biblioteki C++, włącznie z szablonem `set`). Kolejnym niedociągnięciem jest niezgodność klasy `Set` z przyjętymi w standardowej bibliotece konwencjami dotyczącymi klas pojemnikowych (patrz sposób 49.), co utrudnia użytkownikom struktury `Set` korzystanie z innych części tej biblioteki.

Wady interfejsu klasy `Set` nie powinny jednak przesłonić niekwestionowanej zalety tej struktury — relacji pomiędzy klasami `Set` i `list`. Nie jest to relacja „jest" (choć początkowo mogła na taką wyglądać), mamy w tym przypadku do czynienia z relacją implementacji z wykorzystaniem, zaś zastosowanie do jej implementacji mechanizmu podziału na warstwy może być źródłem uzasadnionej dumy u każdego projektanta klas.

Nawiasem mówiąc, w sytuacjach, kiedy wykorzystujesz podział na warstwy do modelowania relacji pomiędzy klasami, tworzysz łączącą je zależność czasu kompilacji. Informacje na temat niepożądanych skutków takiego działania oraz możliwości ich unikania znajdziesz w sposobie 34.

Sposób 41. Rozróżniaj dziedziczenie od stosowania szablonów

Przeanalizuj dwa poniższe problemy związane z projektowaniem.

- Jesteś sumiennym studentem informatyki i chcesz stworzyć klasy reprezentujące stosy obiektów. Będziesz potrzebował wielu różnych klas, ponieważ każdy stos musi być homogeniczny, co oznacza, że może zawierać obiekty tylko jednego typu. Przykładowo, możesz opracować klasę dla stosów liczb całkowitych typu `int`, klasę dla stosów łańcuchów znakowych typu `string` oraz klasę reprezentującą stosy stosów łańcuchów typu `string`. Planujesz jedynie opracowanie minimalnego interfejsu tej klasy (patrz sposób 18.), ograniczasz więc dostępne operacje do tworzenia stosu, niszczenia stosu, położenia obiektu na stosie, zdjęcia obiektu ze stosu oraz określenia, czy stos jest pusty. Rezygnujesz ze stosowania klas dostępnych w standardowej bibliotece C++ (włącznie z klasą `stack` — patrz sposób 49.), ponieważ pragniesz zdobyć doświadczenie w samodzielnym tworzeniu zaawansowanych struktur danych. Ponowne wykorzystywanie gotowych rozwiązań jest oczywiście doskonałym sposobem tworzenia oprogramowania, jeśli jednak Twoim celem jest dogłębna analiza pewnych zachowań, nie ma nic lepszego niż samodzielne ich kodowanie.
- Jesteś miłośnikiem kotów i chcesz opracować klasy reprezentujące koty. Będziesz potrzebował wielu różnych klas, ponieważ każda rasa kotów jest nieco inna. Jak wszystkie obiekty, koty mogą być tworzone i niszczone oraz — co jest oczywiste dla każdego miłośnika kotów — mogą dodatkowo wyłącznie jeść i spać. Koty każdej rasy jedzą i śpią w charakterystyczny dla siebie, ujmujący sposób.

Opisane specyfikacje problemów brzmią podobnie, jednak na ich podstawie należy opracować całkowicie odmienne projekty oprogramowania. Dlaczego?

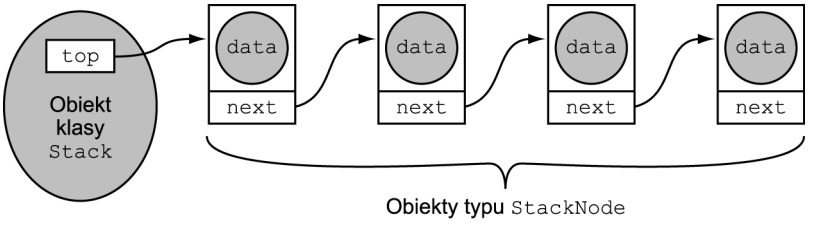
Odpowiedź wynika z relacji pomiędzy zachowaniami poszczególnych klas a *typami* przetwarzanych obiektów. Zarówno w przypadku stosów, jak i w przypadku kotów operujemy na zupełnie innych typach (stosy zawierają obiekty typu `T`, koty reprezentują

obiekty rasy T), jednak pytanie, które musimy sobie postawić, brzmi następująco: „Czy typ T wpływa na *zachowanie* tworzonej klasy?”. Jeśli T *nie* wpływa na zachowanie klasy, możemy zastosować szablon. Jeśli T *ma* wpływ na zachowanie projektowanej klasy, będziemy potrzebowali wirtualnych funkcji, co oznacza, że konieczne będzie wykorzystanie mechanizmu dziedziczenia.

Oto jak możemy zdefiniować opartą na liście jednokierunkowej implementację klasy Stack, zakładając, że przechowywane na stosie obiekty są typu T:

```
class Stack {
public:
  Stack();
  ~Stack();

  void push(const T& object);
  T pop();

  bool empty() const;        // czy stos jest pusty?

private:
  struct StackNode {         // element listy jednokierunkowej
    T data;                  // dane reprezentowane przez ten element
    StackNode *next;         // następny element listy

    // konstruktor StackNode inicjalizuje oba pola
    StackNode(const T& newData, StackNode *nextNode)
    : data(newData), next(nextNode) {}
  };

  StackNode *top;            // szczyt stosu

  Stack(const Stack& rhs);             // uniemożliwiają kopiowanie
  Stack& operator=(const Stack& rhs);  // i przypisywanie (patrz sposób 27.)
};
```

Obiekty klasy Stack będą więc budowały struktury przypominające poniższy schemat:



Sama lista jednokierunkowa składa się z obiektów typu StackNode, jest to jednak wyłącznie szczegół implementacyjny klasy Stack, zatem strukturę StackNode zadeklarowaliśmy jako prywatny typ tej klasy. Zauważ, że dla typu StackNode zdefiniowaliśmy konstruktor zapewniający właściwe inicjalizowanie wszystkich pól tej struktury. Twoje umiejętności umożliwiające tworzenie kodu dla list jednokierunkowy z zamkniętymi oczami nie mogą Ci przesłaniać korzyści płynących z wykorzystania tego typu konstruktorów.

Oto pierwsza próba zaimplementowania funkcji składowych klasy `Stack`. Jak większość prototypowych implementacji (dalekich od ostatecznej, handlowej wersji oprogramowania), poniższy kod nie zawiera instrukcji wykrywających błędy, ponieważ w świecie prototypów nic się nigdy nie psuje:

```
Stack::Stack(): top(0) {}       // inicjalizuje top wskaźnikiem zerowym

void Stack::push(const T& object)
{
  top = new StackNode(object, top); // umieszcza nowy element
}                                   // na początku listy

T Stack:pop()
{
  StackNode *topOfStack = top;  // zachowuje element ze szczytu
  top =  top->next;

  T data = topOfStack->data;    // zachowuje dane elementu
  delete topOfStack;

  return data;
}

Stack::~Stack()                 // usuwa wszystkie elementy ze stosu
{
  while (top) {
    StackNode *toDie = top;     // uzyskuje wskaźnik do elementu ze szczytu
    top = top->next;            // przechodzi do następnego elementu
    delete toDie;               // usuwa poprzedni element ze szczytu
  }
}

bool Stack::empty() const
{ return top == 0; }
```

Powyższe implementacje nie zawierają w sobie żadnych nowych, fascynujących elementów. W rzeczywistości jedynym interesującym szczegółem w powyższym kodzie jest to, że każda z zaprezentowanych funkcji składowych została opracowana *bez najmniejszej znajomości* docelowego typu `T` (zakładamy jedynie, że możemy wywoływać konstruktor kopiujący typu `T`, ale — jak wynika z treści sposobu 45. — mamy do tego prawo). Stworzony kod konstruujący i niszczący stos, kładący i zdejmujący elementy ze stosu oraz określający, czy stos jest pusty, jest całkowicie niezależny od typu `T`. Wyjątkiem jest założenie, że możemy dla tego typu wywoływać konstruktor kopiujący, jednak zachowanie obiektów zdefiniowanej powyżej klasy w żaden inny sposób nie jest uzależnione od `T`. Powyższy przykład doskonale obrazuje podstawową zasadę szablonów klas — zachowanie klasy nie zależy od typu przetwarzanych obiektów.

Przekształcenie klasy `Stack` w szablon jest zresztą tak proste, że mógłby to zrobić każdy:

```
template<class T> class Stack {
  ...                          // dokładnie to samo co w poprzednim kodzie
};
```

Wróćmy teraz do kotów. Dlaczego szablony nie będą dobrym rozwiązaniem dla klas reprezentujących koty?

Przeczytaj raz jeszcze specyfikację i zwróć uwagę na jedno z wymagań: „każda rasa kotów je i śpi w charakterystyczny dla siebie, ujmujący sposób". Oznacza to, że musimy zaimplementować *różne zachowania* dla poszczególnych typów (ras) kotów. Nie możemy po prostu napisać jednej funkcji obsługującej zachowania wszystkich kotów, możemy jedynie stworzyć *specyfikację interfejsu* dla funkcji, którą każdy typ kotów musi implementować. Aha! Możemy przekazać *wyłącznie interfejs* funkcji, deklarując jedynie czystą funkcję wirtualną (patrz sposób 36.):

```
class Cat {
public:
  virtual ~Cat();              // patrz sposób 14.

  virtual void eat() = 0;      // wszystkie koty jedzą
  virtual void sleep() = 0;    // wszystkie koty śpią
};
```

Podklasy klasy `Cat` — powiedzmy `Siamese` i `BritishShortHairedTabby` — muszą oczywiście ponownie zdefiniować dziedziczone interfejsy funkcji `eat` i `sleep`:

```
class Siamese: public Cat {
public:
  void eat();
  void sleep();
  ...
};

class BritishShortHairedTabby: public Cat {
public:
  void eat();
  void sleep();
  ...
};
```

Dobrze, wiemy już, dlaczego szablony są dobrym rozwiązaniem dla klasy `Stack`, i dlaczego nie powinniśmy ich stosować w przypadku klasy `Cat`. Wiemy także, dlaczego właściwym rozwiązaniem dla klasy `Cat` jest dziedziczenie. Pozostaje więc tylko pytanie, dlaczego nie powinniśmy stosować dziedziczenia dla klasy `Stack`. Aby sobie na to pytanie odpowiedzieć, spróbujmy zadeklarować najwyższą klasę (`Stack`) z hierarchii klas reprezentujących stosy — klasę bazową, po której wszystkie pozostałe klasy reprezentujące stosy będą dziedziczyły:

```
class Stack {        // jakakolwiek lista
public:
  virtual void push(const ??? object) = 0;
  virtual ??? pop() = 0;
  ...
};
```

Wszystko jest już jasne. Jaki typ powinniśmy zadeklarować dla czystych funkcji wirtualnych `push` i `pop`? Pamiętaj, że każda podklasa musi ponownie zadeklarować dziedziczone funkcje wirtualne z *dokładnie* tymi samymi typami parametrów i typami zwracanych wyników, z którymi funkcje te zostały zadeklarowane w klasie bazowej.

Niestety, stos liczb całkowitych typu `int` może obsługiwać operacje kładzenia i zdejmowania tylko wartości typu `int`, a np. stos typu `Cat` może obsługiwać operacje kładzenia i zdejmowania tylko obiektów klasy `Cat`. Jak więc możemy zadeklarować w klasie `Stack` jej czyste funkcje wirtualne w taki sposób, by klienci mogli tworzyć zarówno stosy liczb całkowitych, jak i stosy obiektów klasy `Cat`? Gorzka prawda jest taka, że nie możemy tego zrobić i właśnie dlatego dziedziczenie nie jest właściwym mechanizmem do tworzenia stosów.

Być może sądzisz, że jesteś sprytniejszy. Może Ci przyjść do głowy, że jesteś w stanie przechytrzyć swoje kompilatory, wykorzystując ogólne wskaźniki (`void*`). Okazuje się jednak, że w tym przypadku wskaźniki typu `void*` Ci nie pomogą. Obejście wymagania, by deklaracje wirtualnych funkcji w klasach potomnych były zgodne z deklaracjami w klasie bazowej, jest zwyczajnie niemożliwe. Ogólne wskaźniki mogą nam jednak pomóc w rozwiązaniu zupełnie innego problemu — związanego z efektywnością klas generowanych na podstawie szablonów (szczegóły znajdziesz w sposobie 42.).

Zakończmy nasze rozważania dotyczące stosów i kotów — spróbujmy teraz podsumować wnioski płynące z treści tego sposobu:

- Szablon powinien być wykorzystywany do generowania zbioru klas w sytuacji, gdy typ przetwarzanych obiektów *nie wpływa* na zachowania należących do definiowanej klasy funkcji.
- Dziedziczenie powinno być wykorzystywane dla zbioru klas w sytuacji, gdy typ przetwarzanych obiektów *wpływa* na zachowania należących do definiowanej klasy funkcji.

Połącz te dwa punkty, a poczynisz ogromny krok w kierunku mistrzostwa we właściwym dobieraniu mechanizmu dziedziczenia i szablonów.

Sposób 42. Dziedziczenie prywatne stosuj ostrożnie

Prezentując sposób 35., wykazałem, że C++ traktuje publiczne dziedziczenie jak relację „jest". Zademonstrowałem to na przykładzie sytuacji, w której kompilatory, mając hierarchię, w której klasa `Student` publicznie dziedziczy po klasie `Person`, niejawnie przekształcają obiekty klasy `Student` w obiekty klasy `Person`, gdy jest to niezbędne do poprawnej realizacji wywołania funkcji. Warto w tym momencie przypomnieć fragment tamtego przykładu z jedną zmianą — zamiast dziedziczenia publicznego zastosujemy dziedziczenie prywatne:

```
class Person { ... };

class Student:                  // tym razem stosujemy
  private Person { ... };       // dziedziczenie prywatne

void dance(const Person& p);    // każdy może tańczyć
void study(const Student& s);   // tylko studenci studiują
```

```
Person p;                    // p reprezentuje osobę
Student s;                   // s reprezentuje studenta

dance(p);                    // dobrze, p reprezentuje osobę
dance(s);                    // błąd! student nie jest osobą
```

Oczywisty wniosek jest taki, że dziedziczenie prywatne nie oznacza relacji „jest". Co więc faktycznie oznacza?

Myślisz pewnie, że zanim zajmiemy się rzeczywistym znaczeniem dziedziczenia prywatnego, powinniśmy przeanalizować zachowanie programów, w których wykorzystujemy ten mechanizm. Dobrze, pierwszą regułę rządzącą prywatnym dziedziczeniem mogliśmy właśnie zaobserwować — w przeciwieństwie do dziedziczenia publicznego, kompilatory w ogólności *nie* przekształcają obiektów klasy potomnej (np. klasy `Student`) w obiekty klasy bazowej (np. klasy `Person`), jeśli zdefiniowano między tymi klasami relację dziedziczenia prywatnego. Dlatego właśnie wywołanie funkcji `dance` dla obiektu `s` zakończyło się niepowodzeniem. Druga reguła określa, że składowe odziedziczone po prywatnej klasie bazowej stają się prywatnymi składowymi klasy potomnej, nawet jeśli w klasie bazowej zostały zadeklarowane jako składowe chronione lub publiczne. To wszystko, co możemy powiedzieć o zachowaniu kodu zawierającego dziedziczenie prywatne.

Przejdźmy więc do rzeczywistego znaczenia tej relacji. Dziedziczenie prywatne modeluje relację implementacji z wykorzystaniem. Kiedy deklarujemy klasę `D` prywatnie dziedziczącą po klasie `B`, robimy to dlatego, że chcemy w klasie `D` wykorzystać część kodu napisanego dla klasy `B`; pojęciowe relacje łączące obiekty typu `B` z obiektami typu `D` nie mają tutaj żadnego znaczenia. Oznacza to, że dziedziczenie prywatne ma wyłącznie charakter techniki implementacji klas. Posługując się językiem ze sposobu 36., możemy powiedzieć, że dziedziczenie prywatne oznacza, że dziedziczona powinna być *tylko* implementacja, interfejs powinien być całkowicie ignorowany. Jeśli klasa `D` dziedziczy prywatnie po klasie `B`, oznacza to tylko tyle, że obiekty klasy `D` są implementowane z wykorzystaniem obiektów klasy `B`. Dziedziczenie prywatne nie jest techniką wykorzystywaną w fazie *projektowania* oprogramowania, a jedynie podczas *implementowania* programów.

Fakt, że prywatne dziedziczenie oznacza w rzeczywistości relację implementacji z wykorzystaniem jest nieco mylący, ponieważ, prezentując sposób 40. stwierdziłem, że takie samo znaczenie ma rozmieszczanie obiektów w warstwach. Na jakiej podstawie powinniśmy więc wybierać właściwą technikę z tej pary? Odpowiedź jest prosta — stosuj podział na warstwy zawsze, gdy jest to możliwe; stosuj dziedziczenie prywatne tylko wtedy, gdy jest to jedyne rozwiązanie. Kiedy możemy mieć do czynienia z tą drugą sytuacją? Kiedy mamy do czynienia z chronionymi składowymi i (lub) wirtualnymi funkcjami.

W sposobie 41. opisałem sposób tworzenia szablonu `Stack`, którego zadaniem było generowanie klas przechowujących obiekty różnych typów. Być może powinieneś zapoznać się teraz z treścią tego sposobu. Szablony są jednym z najbardziej przydatnych elementów udostępnianych w języku C++, kiedy jednak zaczniesz je stosować regularnie, szybko odkryjesz, że tworząc wiele obiektów danego szablonu, prawdopodobnie zwielokrotniasz także jego *kod*. W przypadku szablonu `Stack` kod funkcji

składowych klasy `Stack<int>` będzie przecież zupełnie inny niż kod funkcji składowych klasy `Stack<double>`. W niektórych sytuacjach nie można tego uniknąć, jednak z podobną powtarzalnością kodu mamy prawdopodobnie do czynienia nawet w przypadkach, gdy funkcje szablonu w rzeczywistości mogłyby wykorzystywać wspólny kod. Wynikający z tego przyrost rozmiarów kodu obiektu ma swoją nazwę — spowodowane przez szablon *puchnięcie kodu*. Nie jest to oczywiście pozytywne zjawisko.

W przypadku niektórych typów klas możesz uniknąć tego zjawiska, stosując wskaźniki ogólne. Dotyczy to klas przechowujących *wskaźniki* zamiast obiektów i zaimplementowanych zgodnie z następującymi regułami:

1. Pojedyncza klasa zawiera wskaźniki typu `void*` do obiektów.
2. Istnieje dodatkowy zbiór klas, których jedynym celem jest egzekwowanie ścisłej kontroli typów. Wszystkie te klasy wykorzystują do normalnego działania ogólną klasę z punktu 1.

Oto przykład zastosowania klasy `Stack` ze sposobu 41. w wersji niebędącej szablonem; jedyna zmiana dotyczy przechowywania wskaźników ogólnych zamiast przechowywanych wcześniej obiektów:

```
class GenericStack {
public:
  GenericStack();
  ~GenericStack();

  void push(void *object);
  void * pop();

  bool empty() const;

private:
  struct StackNode {
    void *date;             // dane reprezentowane przez ten element
    StackNode *next;        // następny element listy

    StackNode(void *newData, StackNode *nextNode)
    : data(newData), next(nextNode) {}
  };

  StackNode *top;                          // szczyt stosu

  GenericStack(const GenericStack& rhs);   // uniemożliwia kopiowanie
  GenericStack&                            // i przypisywanie (patrz
    operator=(const GenericStack& rhs);    // sposób 27.)
};
```

Ponieważ powyższa klasa przechowuje wskaźniki zamiast obiektów, istnieje możliwość, że dany obiekt jest wskazywany przez więcej niż jeden stos (został położony na wielu stosach). Zasadnicze znaczenie ma w takiej sytuacji zapewnienie, by funkcja `pop` i destruktor klasy *nie* usuwały wskaźnika `data` w żadnym niszczonym obiekcie typu `StackNode` i jednocześnie nadal usuwały sam obiekt `StackNode`. Obiekty typu `StackNode` mają w końcu przydzielaną pamięć wewnątrz klasy `GenericStack`, zatem

także tam muszą być usuwane. Oznacza to, że implementacja klasy `Stack` ze sposobu 41. niemal w zupełności wystarcza także dla klasy `GenericStack`. Jedyne potrzebne zmiany to zastąpienie typu `T` typem `void*`.

Sama klasa `GenericStack` jest mało użyteczna — jest też zbyt prosta, by możliwe było jej niewłaściwe wykorzystanie. Klient mógłby jednak przez pomyłkę położyć na stosie przechowującym wyłącznie wskaźniki do liczb całkowitych typu `int` wskaźnik do obiektu klasy `Cat`, a kompilatory i tak zaakceptowałyby takie posunięcie. W końcu parametr będący wskaźnikiem typu `void*` może dotyczyć dowolnych obiektów.

Aby odzyskać bezpieczeństwo typów, do którego zdążyliśmy się już przyzwyczaić, musimy stworzyć *klasy interfejsu* do `GenericStack`:

```
class IntStack {            // klasa interfejsu dla wartości typu int
public:
  void push(int *intPtr) { s.push(intPtr); }
  int * pop() { return static_cast<int*>(s.pop()); }
  bool empty() const { return s.empty(); }

private:
  GenericStack s;           // implementacja
};

class CatStack {            // klasa interfejsu dla wartości typu Cat
public:
  void push(Cat *catPtr) { s.push(catPtr); }
  Cat * pop() { return static_cast<Cat*>(s.pop()); }
  bool empty() const { return s.empty(); }

private:
  GenericStack s;           // implementacja
};
```

Jak widać, zadaniem klas `IntStack` i `CatStack` jest wyłącznie zapewnienie ścisłej kontroli typów. Pierwsza klasa umożliwia umieszczanie na stosie i zdejmowanie ze stosu jedynie wartości całkowitoliczbowych typu `int`; druga zezwala na umieszczanie na stosie i zdejmowanie ze stosu wyłącznie obiektów klasy `Cat`. Obie klasy (zarówno `IntStack`, jak i `CatStack`) zostały zaimplementowane w powiązaniu z klasą `GenericStack` (tę relację wyrażamy za pomocą mechanizmu podziału na warstwy — patrz sposób 40.) i obie klasy wykorzystują kod napisany dla funkcji składowych klasy `GenericStack`, które w rzeczywistości implementuje ich zachowania. Co więcej, fakt, że wszystkie funkcje składowe klas `IntStack` i `CatStack` są (niejawnie) wbudowywane, oznacza, że koszt wykorzystania powyższych klas interfejsów w czasie wykonywania programu jest bliski zeru.

Co się jednak stanie, jeśli potencjalni klienci nie zdadzą sobie z tego sprawy? Co się stanie, kiedy błędnie przyjmą, że zastosowanie samej klasy `GenericStack` jest bardziej efektywne lub jeśli są na tyle lekkomyślni, że sądzą, iż tworzenie kodu bezpiecznie operującego typami jest domeną mięczaków? Jak możemy zmusić ich do stosowania pośredniczących klas `IntStack` i `CatStack`, zamiast bezpośrednich odwołań do składowych klasy `GenericStack`, które mogą prowadzić do tego rodzaju błędów typów, których starali się szczególnie uniknąć twórcy C++?

Nic, nic nie może zmusić do tego potencjalnych klientów Twojego kodu. A może jednak istnieje coś, co może pomóc?

Na początku tego sposobu wspomniałem, że alternatywnym sposobem ustanawiania pomiędzy klasami relacji implementacji z wykorzystaniem jest wiązanie ich dziedziczeniem prywatnym. W tym przypadku technika ta okazuje się lepsza niż dzielenie na warstwy, ponieważ umożliwia wyrażanie założenia, że klasa `GenericStack` nie jest wystarczająco bezpieczna dla ogólnych zastosowań i powinna być wykorzystywana wyłącznie w charakterze implementacji innych klas. Można to wyrazić, umieszczając funkcje składowe klasy `GenericStack` w bloku `protected`:

```
class GenericStack {
protected:
  GenericStack();
  ~GenericStack();

  void push(void *object);
  void * pop();

  bool empty() const;

private:
  ...                          // to samo co wcześniej
};

GenericStack s;                // błąd! konstruktor jest chroniony

class IntStack {
public:
  void push(int *intPtr) { GenericStack::push(intPtr); }
  int * pop() { return static_cast<int*>( GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};

class CatStack {
public:
  void push(Cat *catPtr) { GenericStack::push(catPtr); }
  Cat * pop() { return static_cast<Cat*>( GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};

IntStack is;                   // dobrze
CatStack cs;                   // także dobrze
```

Podobnie jak rozwiązanie oparte na podziale na warstwy, powyższa implementacja oparta na prywatnym dziedziczeniu pozwala uniknąć powielania kodu, ponieważ bezpieczne pod względem obsługi typów klasy interfejsów składają się wyłącznie z wbudowanych wywołań funkcji składowych klasy `GenericStack` (będących właściwą implementacją).

Budowa bezpiecznych pod względem obsługi typów interfejsów ponad klasą `GenericStack` jest sprytnym rozwiązaniem, jednak ręczne tworzenie klas interfejsów dla wszystkich możliwych typów byłoby bardzo pracochłonne. Na szczęście nie musimy

tego robić — możemy przecież wykorzystać szablon, który wygeneruje potrzebne klasy automatycznie. Oto oparty na prywatnym dziedziczeniu szablon generujący bezpiecznie operujące na typach interfejsy stosów:

```
template<class T>
class Stack: private GenericStack {
public:
  void push(T *objectPtr) { GenericStack::push(objectPtr); }
  T * pop() { return static_cast<T*>( GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};
```

Być może nie jest to dla Ciebie od razu takie oczywiste, jednak powyższy kod jest niesamowity — dzięki zastosowaniu szablonu kompilatory automatycznie wygenerują tyle klas interfejsów, ile będziemy potrzebować. Ponieważ generowane klasy są bezpieczne pod względem obsługi typów, popełniane przez klienta błędy w tym zakresie będą wykrywane już w czasie kompilacji. Ponieważ funkcje składowe klasy `GenericStack` są chronione oraz ponieważ klasy interfejsów wykorzystują `GenericStack` jako prywatną klasę bazową, klienci nie mogą obejść klas interfejsów i uzyskać bezpośredniego dostępu do klasy implementacji. Ponieważ każda funkcja składowa klasy interfejsu jest (niejawnie) deklarowana z atrybutem `inline`, stosowanie klas bezpiecznie obsługujących typy nie powoduje żadnych dodatkowych kosztów w trakcie wykonywania programu — wygenerowany kod jest identyczny jak kod obsługujący bezpośredni dostęp do składowych klasy `GenericStack` (zakładając, że kompilatory uwzględniają dyrektywy `inline` — patrz sposób 33.). Ponieważ w klasie `GenericStack` zastosowaliśmy wskaźniki `void*`, ponosimy koszty operowania na stosach przez dokładnie jedną kopię kodu, niezależnie od liczby różnych typów stosów wykorzystywanych w programie. Krótko mówiąc, zaprezentowany projekt zapewnia naszemu programowi maksymalną efektywność i maksymalne bezpieczeństwo typów. Niełatwo będzie skonstruować lepsze rozwiązanie.

Jednym z wniosków płynących z tej książki jest ten, że różne własności języka C++ wzajemnie na siebie oddziaływają niekiedy w sposób niezwykły. Myślę, że zgodzisz się ze mną, że powyższe rozwiązanie jest tego dobrym przykładem.

Nie moglibyśmy zrealizować omawianego przykładu za pomocą mechanizmu podziału na warstwy. Wynika to z faktu, że tylko mechanizm dziedziczenia daje możliwość dostępu do chronionych składowych i tylko dziedziczenie umożliwia ponowne definiowanie wirtualnych funkcji (przykład funkcji wirtualnych, których obecność może skłonić programistę do zastosowania prywatnego dziedziczenia, znajdziesz w sposobie 43.). W sytuacjach, w których mamy do czynienia z klasą zawierającą funkcje wirtualne i chronione składowe prywatne, dziedziczenie jest niekiedy jedynym sposobem wyrażania relacji implementacji z wykorzystaniem pomiędzy klasami. Nie powinieneś więc obawiać się stosowania dziedziczenia prywatnego, kiedy okaże się, że jest to najbardziej właściwa technika, jaką masz w danym przypadku do dyspozycji. Powinieneś jednak pamiętać, że lepszą techniką jest w ogólności dzielenie na warstwy, powinieneś więc stosować ją zawsze, kiedy możesz.

Sposób 43. Dziedziczenie wielobazowe stosuj ostrożnie

Różni programiści rozmaicie postrzegają technikę dziedziczenia wielobazowego — jedni sądzą, że jest dziełem samego Boga, inni twierdzą, że jest oczywistym dowodem na istnienie szatana.

Zwolennicy dziedziczenia wielobazowego utrzymują, że technika ta jest niezwykle istotnym elementem naturalnego modelowania problemów świata rzeczywistego; przeciwnicy przekonują natomiast, że jest wolna, trudna w implementacji i nie daje większych możliwości niż zwykłe dziedziczenie po pojedynczej klasie bazowej. Niestety, także świat obiektowych języków programowania jest w tym względzie podzielony — dziedziczenie wielobazowe jest możliwe w języku C++, Eiffel i Common LISP Object System (CLOS), nie jest dostępne w językach Smalltalk, Objective C i Object Pascal, natomiast Java obsługuje tę technikę w ograniczonej formie. W co biedny programista powinien więc wierzyć?

Zanim uwierzysz w cokolwiek, powinieneś uporządkować pewne fakty. Niezaprzeczalną cechą dotyczącą dziedziczenia wielobazowego w C++ jest fakt, że otwiera puszkę Pandory zawierająca mnóstwo komplikacji, które zwyczajnie nie mają miejsca w przypadku dziedziczenia zwykłego. Najprostszą z nich jest wieloznaczność wywołań dziedziczonych funkcji składowych (patrz sposób 26.). Jeśli klasa potomna dziedziczy składową o tej samej nazwie po więcej niż jednej klasie bazowej, każde odwołanie do tej nazwy jest niejednoznaczne; musisz więc jawnie określać, o którą składową Ci chodzi. Oto przykład oparty na naszych rozważaniach ze sposobu 50.:

```
class Lottery {
public:
  virtual int draw();
  ...
};

class GraphicalObject {
public:
  virtual int draw();
  ...
};

class LotterySimulation: public Lottery,
                         public GraphicalObject {
  ...                              // brak deklaracji składowej draw
};

LotterySimulation *pls = new LotterySimulation;

pls->draw();                       // błąd! niejednoznaczność
pls->Lottery::draw();              // dobrze
pls->GraphicalObject::draw();      // dobrze
```

Powyższe wywołania funkcji `draw` wyglądają dosyć niezgrabnie, ale przynajmniej działają prawidłowo. Niestety, taki wygląd wywołań jest stosunkowo trudny do wyeliminowania. Niejednoznaczności wywołań nie można wyeliminować, nawet definiując jedną z dziedziczonych funkcji jako prywatną (a więc niedostępną) — istnieje sensowne wytłumaczenie takiego zachowania; omówiłem je w sposobie 26.

Jawne kwalifikowanie wywoływanych składowych jest nie tylko niezgrabne, rodzi także pewne ograniczenia. Kiedy jawnie kwalifikujemy daną funkcję wirtualną z nazwą klasy, funkcja przestaje być traktowana jak wirtualna. Zamiast tego, wywoływana funkcja to dokładnie ta, którą wyznaczamy; nawet jeśli obiekt, dla którego jest wywoływana, jest egzemplarzem klasy potomnej:

```
class SpecialLotterySimulation: public LotterySimulation {
public:
  virtual int draw();
  ...
};

pls = new SpecialLotterySimulation;

pls->draw();                      // błąd! - nadal niejednoznaczność
pls->Lottery::draw();             // wywołuje Lottery::draw
pls->GraphicalObject::draw();     // wywołuje GraphicalObject::draw
```

Zauważ, że mimo iż w tym przypadku `pls` wskazuje na obiekt klasy `SpecialLotterySimulation`, nie mamy możliwości (bez rzutowania w dół hierarchii klas — patrz sposób 39.) wywołania funkcji `draw` zdefiniowanej w tej właśnie klasie.

Poczekaj, jest coś jeszcze. Zarówno wersja funkcji `draw` z klasy `Lottery`, jak i wersja z klasy `GraphicalObject` została zadeklarowana jako wirtualna po to, by podklasy tych klas mogły je ponownie definiować (patrz sposób 36.), co się jednak stanie, kiedy spróbujemy w klasie `LotterySimulation` zdefiniować ponownie *obie* wersje? Niestety, nie możemy tego zrobić, ponieważ klasa może zawierać tylko jedną bezargumentową funkcję składową nazwaną `draw` (istnieje szczególny wyjątek od tej reguły, kiedy jedna z funkcji jest stała, a druga nie — patrz sposób 21.).

Omawiany problem był uważany za tak istotny, że rozważano nawet wprowadzenie odpowiednich zmian w języku C++. Chodziło o wprowadzenie możliwości „zmiany nazw” dziedziczonych funkcji wirtualnych, jednak szybko zdano sobie sprawę, że problem można wyeliminować dodając parę nowych klas:

```
class AuxLottery: public Lottery {
public:
  virtual int lotteryDraw() = 0;
  virtual int draw() { return lotteryDraw(); }
};

class AuxGraphicalObject: public GraphicalObject {
public:
  virtual int graphicalObjectDraw() = 0;
  virtual int draw() { return graphicalObjectDraw(); }
};
```

```
class LotterySimulation: public Lottery,
                         public GraphicalObject {
public:
  virtual int lotteryDraw();
  virtual int graphicalObjectDraw();
  ...
};
```

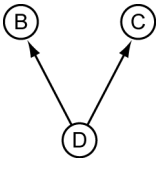
Każda z dwóch nowych klas, `AuxLottery` i `AuxGraphicalObject`, deklaruje w istocie nową nazwę dla dziedziczonej funkcji `draw`. Nowa nazwa przyjmuje postać czystej funkcji wirtualnej (w tym przypadku, odpowiednio `lotteryDraw` i `graphicalObjectDraw`), co sprawia, że konkretne podklasy muszą je ponownie definiować. Co więcej, każda z klas ponownie definiujących dziedziczoną funkcję wywołuje nową czystą funkcję wirtualną. W rezultacie wewnątrz obu nowych klas należących do omawianej hierarchii pojedyncza, niejednoznaczna nazwa funkcji `draw` została faktycznie rozbita na dwie, jednoznaczne, ale funkcjonalnie równoważne nazwy funkcji: `lotteryDraw` i `graphicalObjectDraw`:

```
LotterySimulation *pls = new LotterySimulation;

Lottery *pl = pls;
GraphicalObject *pgo = pls;

// wywołanie funkcji LotterySimulation::LotteryDraw
pl->draw();

// wywołanie funkcji LotterySimulation::graphicalObjectDraw
pgo->draw();
```

Powinieneś dobrze zapamiętać powyższą strategię, w której sprytnie zastosowaliśmy czyste funkcje wirtualne, proste funkcje wirtualne i funkcje z atrybutem `inline` (patrz sposób 33.). Po pierwsze, rozwiązuje to problem, z którym możesz się pewnego dnia spotkać. Po drugie, przypomina o komplikacjach wynikających ze stosowania techniki dziedziczenia wielobazowego. Tak, zaprezentowane rozwiązanie działa poprawnie, zastanów się jednak, czy naprawdę chcesz wprowadzać nowe klasy tylko po to, by umożliwić sobie ponowne definiowanie wirtualnych funkcji. Klasy `AuxLottery` i `AuxGraphicalObject` mają podstawowe znaczenie dla poprawnego funkcjonowania hierarchii, nie odpowiadają jednak ani abstrakcji na poziomie definicji problemu, ani abstrakcji na poziomie implementacji jego rozwiązania. Są tylko i wyłącznie narzędziem umożliwiającym nam implementację pewnego modelu. Wiesz już, że dobre oprogramowanie powinno być niezależne od tego typu narzędzi. Ta zasada ma zastosowanie także w tym przypadku.

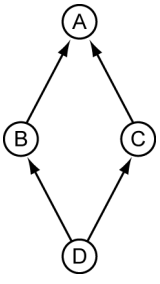
Problem abstrakcji — choć interesujący — może znacznie ograniczać nasze możliwości wykorzystywania techniki dziedziczenia wielobazowego. Jak wynika z obserwacji, kolejnym problemem jest fakt, że hierarchia dziedziczenia wielobazowego w postaci:

```
class B { ... };
class C { ... };
class D: public B, public C { ... };
```



wykazuje niepokojącą tendencję do ewoluowania w kierunku hierarchii w postaci:

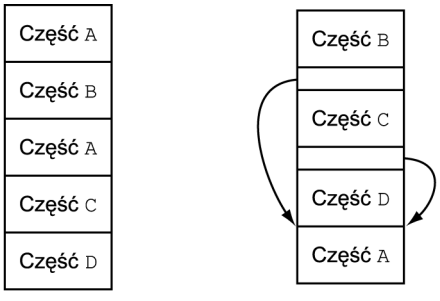
```
class A { ... };
class B: virtual public A { ... };
class C: virtual public A { ... };
class D: public B, public C { ... };
```



Niezależnie od tego, czy prawdą jest, że diamenty są najlepszymi przyjaciółmi kobiety, z pewnością zaprezentowana powyżej hierarchia dziedziczenia w kształcie diamentu nie jest przyjacielem programisty. Kiedy tworzymy podobną hierarchię, na samym początku musimy sobie odpowiedzieć na pytanie, czy `A` powinna być wirtualną klasą bazową (czyli, czy dziedziczenie po tej klasie powinno być wirtualne). W praktyce odpowiedź niemal zawsze powinna być twierdząca; tylko w szczególnych przypadkach będziemy chcieli, by obiekt klasy `D` zawierał wiele kopii danych będących składowymi klasy `A`. W powyższym przykładzie w klasach `B` i `C` zadeklarowaliśmy `A` jako wirtualną klasę bazową.

Niestety, kiedy definiujemy klasy `B` i `C`, możemy nie wiedzieć, czy jakakolwiek inna klasa będzie jednocześnie dziedziczyła po nich obu i w rzeczywistości do poprawnego zdefiniowania tych klas taka wiedza nie powinna nam być potrzebna. Stawia nas to w bardzo trudnym położeniu, przynajmniej jako projektantów tych klas. Jeśli *nie* zadeklarujemy `A` jako wirtualnej klasy bazowej klas `B` i `C`, późniejsi projektanci klasy `D` będą być może zmuszeni do zmodyfikowania definicji klas `B` i `C`, by umożliwić sobie ich efektywne wykorzystanie. Takie rozwiązanie jest zazwyczaj nie do przyjęcia, zwykle dlatego, że definicje klas `A`, `B` i `C` są dostępne tylko do odczytu. Może to wynikać np. z faktu, że klasy `A`, `B` i `C` znajdują się w bibliotece, a klasa `B` jest tworzona przez klienta tej biblioteki.

Z drugiej strony, jeśli *zadeklarujemy* `A` jako wirtualną klasę bazową dla klas `B` i `C`, będziemy musieli zazwyczaj ponieść dodatkowe koszty zarówno w wymiarze wykorzystywanej przestrzeni pamięciowej, jak i czasu działania programów klientów tych klas. Wynika to z faktu, że wirtualne klasy bazowe są zwykle implementowane jako *wskaźniki* do obiektów, nie zaś jako same obiekty. Rozmieszczanie obiektów w pamięci zależy zwykle od konkretnych działań poszczególnych kompilatorów, jednak faktem jest, że obiekt klasy `D` z niewirtualną klasą bazową `A` jest zazwyczaj umieszczany w szeregu przylegających komórek pamięci, natomiast obiekt klasy `D` z wirtualną klasą bazową `A` jest niekiedy umieszczany w szeregu przylegających komórek pamięci, z których dwa zawierają wskaźniki do komórek zawierających składowe z danymi wirtualnej klasy bazowej:



Popularny sposób rozmieszczania w pamięci obiektu klasy D, gdzie A jest jego niewirtualną klasą bazową

Sposób, w jaki niektóre kompilatory rozmieszczają w pamięci obiekt klasy D, gdzie A jest wirtualną klasą bazową

Nawet kompilatory niestosujące tej konkretnej strategii implementacji w ogólności nałożą na program klienta dodatkowy koszt związany ze zwiększonym wykorzystaniem pamięci przez wirtualnie dziedziczące klasy.

Mając na uwadze powyższą analizę, wygląda na to, że projektowanie efektywnych klas wykorzystujących technikę dziedziczenia wielobazowego wymaga od projektantów bibliotek zdolności jasnowidztwa. Widząc, jak rzadką cechą jest w naszych czasach zdrowy rozsądek, przesadne poleganie na własnościach języka, które wymagają od projektantów nie tylko zwykłego przewidywania przyszłych zastosowań, ale także zdolności wróżbiarskich, jest bardzo ryzykowne.

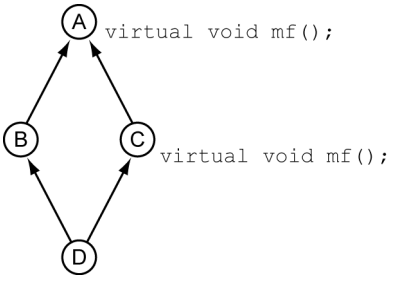
To samo można oczywiście powiedzieć o wyborze pomiędzy funkcjami wirtualnymi a niewirtualnymi w klasie bazowej, istnieje jednak zasadnicza różnica. W sposobie 36. wyjaśniłem, że funkcja wirtualna ma dokładnie zdefiniowane wysokopoziomowe znaczenie, które jest inne od odpowiedniego, równie dokładnie zdefiniowanego wysokopoziomowego znaczenia funkcji niewirtualnej. Dokonanie właściwego wyboru pomiędzy tymi dwiema możliwościami jest więc możliwe w oparciu o to, co chcemy przekazać autorom potencjalnych podklas. Podejmując decyzję odnośnie wirtualnej lub niewirtualnej klasy bazowej nie mamy jednak do dyspozycji tak dobrze zdefiniowanych znaczeń wysokiego poziomu. Decyzję musimy więc opierać zwykle na strukturze całej hierarchii dziedziczenia, co oznacza, że odpowiednie kroki nie mogą być podejmowane do momentu jej zaprojektowania. Jeśli musisz znać dokładne zastosowania swojej klasy, zanim przystąpisz do jej poprawnego zdefiniowania, projektowanie efektywnych klas staje się bardzo trudne.

Kiedy już poradzisz sobie z problemem niejednoznaczności i odpowiesz na pytanie, czy dziedziczenie po klasie bazowej (lub klas bazowych) powinno być wirtualne, nadal czeka Cię wiele komplikacji. Zamiast nad nimi rozpaczać, wspomnę jedynie o dwóch problemach, na które powinieneś zwracać szczególną uwagę:

- **Przekazywanie argumentów konstruktora do wirtualnych klas bazowych**. W przypadku zastosowania techniki niewirtualnego dziedziczenia argumenty konstruktora klasy bazowej są wyznaczane za pomocą list inicjalizacji składowych klas pośredniczących w dziedziczeniu po klasie bazowej.

Ponieważ hierarchie pojedynczego dziedziczenia wymagają wyłącznie niewirtualnych klas bazowych, argumenty są przekazywane w górę hierarchii dziedziczenia w sposób zupełnie naturalny — klasy na *n*-tym poziomie hierarchii przekazują argumenty do klas na poziomie ($n - 1$). W przypadku konstruktorów wirtualnej klasy bazowej argumenty są jednak wyznaczane za pomocą list inicjalizacji składowych klas *najbardziej potomnych* względem klasy bazowej. W efekcie klasa inicjalizująca wirtualną klasę bazową może być od niej dowolnie oddalona w hierarchii dziedziczenia i może się zmieniać wraz z dodawaniem do hierarchii nowych klas. Dobrym sposobem ominięcia tego problemu jest wyeliminowanie potrzeby przekazywania argumentów konstruktora do wirtualnych klas bazowych. Najprostszym sposobem jest oczywiście unikanie umieszczania danych składowych w tych klasach. Przykładem takiego rozwiązania jest język Java — definiowane tak wirtualne klasy bazowe (nazywane „interfejsami") zwyczajnie nie mogą zawierać żadnych danych.

- **Przewaga funkcji wirtualnych**. Zaraz po tym, gdy stwierdziliśmy, że jesteśmy w stanie właściwie identyfikować wszystkie niejednoznaczności stosowanych wywołań, zmieniły się istotne reguły ich zachowania. Rozważmy ponownie przykład przypominającego diament grafu dziedziczenia dla klas `A`, `B`, `C` i `D`. Przypuśćmy, że klasa `A` definiuje wirtualną funkcję składową `mf`, która jest ponownie definiowana w klasie `C`, ale nie jest już definiowana w klasach `B` i `D`:



Na podstawie wniosków płynących z naszych wcześniejszych analiz, wydawać by się mogło, że będziemy mieli do czynienia z niejednoznacznością:

```
D *pd = new D;
pd->mf();                    // A::mf czy C::mf?
```

Która wersja funkcji `mf` powinna być wywołana dla obiektu klasy `D`? Ta bezpośrednio dziedziczona po klasie `C`, czy też dziedziczona pośrednio (przez klasę `B`) po klasie `A`? Oto odpowiedź: *to zależy od sposobu, w jaki klasy `B` i `C` dziedziczą po klasie `A`*. W szczególności, jeśli `A` jest niewirtualną klasą bazową dla klasy `B` lub `C`, przedstawione wywołanie jest niejednoznaczne; jeśli jednak `A` jest wirtualną klasą bazową zarówno dla klasy `B`, jak i `C` mówimy, że ponowna definicja funkcji `mf` w klasie `C` *dominuje* nad oryginalną definicją z klasy `A` — wywołanie funkcji `mf` za pośrednictwem wskaźnika `pd` będzie wówczas (jednoznacznie) dotyczyło wersji `C::mf`. Jeśli dokładnie przeanalizujesz teraz to zachowanie, okaże się, że właśnie tego szukałeś, jednak dokładne prześledzenie wszystkich aspektów tego zachowania może być szalenie trudne.

Być może przyznasz teraz, że dziedziczenie wielobazowe może prowadzić do wielu komplikacji. Być może jesteś przekonany, że nigdy nie będziesz zmuszony wykorzystać tej techniki dziedziczenia. Być może jesteś gotowy zaproponować międzynarodowej komisji standaryzacji C++ usunięcie dziedziczenia wielobazowego z tego języka lub przynajmniej zaproponować szefowi swojego projektu, by zakazał programistom stosowania tej techniki.

Być może jesteś zbyt porywczy.

Pamiętaj, że projektanci C++ nie stworzyli techniki dziedziczenia wielobazowego, która byłaby trudna w stosowaniu, dopiero później okazało się, że w połączeniu z innymi — bardziej lub mniej sensownymi — elementami ten typ dziedziczenia pociąga za sobą pewne komplikacje. W powyższych rozważaniach mogłeś zauważyć, że większość tych komplikacji pojawia się dopiero w połączeniu ze stosowaniem wirtualnych klas bazowych. Jeśli więc możesz uniknąć ich stosowania (jeśli możesz zrezygnować z tworzenia morderczych grafów dziedziczenia), większość problemów po prostu przestanie istnieć.

Przykładowo, w sposobie 34. opisałem *klasę protokołu* istniejącą wyłącznie po to, by definiować interfejs klasy potomnej — omówiona klasa nie zawierała żadnych składowych danych, nie definiowała też żadnych konstruktorów; zawierała wyłącznie wirtualny konstruktor (patrz sposób 14.) i pełniący rolę specyfikacji interfejsu zbiór czystych funkcji wirtualnych. Klasa protokołu `Person` mogłaby mieć postać:

```
class Person {
public:
  virtual ~Person();

  virtual string name() const = 0;
  virtual string birthDate() const = 0;
  virtual string address() const = 0;
  virtual string nationality() const = 0;
};
```

Klienci tej klasy muszą wykorzystywać w swoich programach wskaźniki i referencje do `Person`, ponieważ nie można tworzyć obiektów abstrakcyjnych klas.

Aby utworzyć obiekt, który można wykorzystać jak obiekty klasy `Person`, klienci tej klasy muszą wykorzystać specjalne *funkcje fabryczne* (patrz sposób 34.) tworzące obiekty konkretnych podklas klasy `Person`:

```
// funkcja fabryczna tworząca obiekt klasy Person na podstawie
// unikalnego identyfikatora w bazie danych
Person * makePerson(DatabaseID personIdentifier);
DatabaseID askUserForDatabaseID();

DatabaseID pid = askUserForDatabaseID();
Person *pp = makePerson(pid);    // tworzy obiekt obsługujący
                                 // interfejs klasy Person

...                              // operuje na *pp za pośrednictwem
                                 // funkcji składowych klasy Person

delete pp;                       // usuwa niepotrzebny już obiekt
```

Jak jednak funkcja makePerson może tworzyć obiekty wskazywane przez zwracane wskaźniki? To proste, musi istnieć jakaś konkretna klasa potomna względem klasy Person, której obiekty będą mogły być tworzone wewnątrz funkcji makePerson.

Przypuśćmy, że taka klasa nosi nazwę MyPerson. Jako konkretna klasa, MyPerson musi zapewniać implementację dziedziczonych po klasie Person czystych funkcji wirtualnych. Można je napisać od początku, jednak zgodnie z zaleceniami inżynierii oprogramowania lepszym rozwiązaniem będzie wykorzystanie istniejących komponentów, których większość lub wszyscy programiści używali już w przeszłości. Przykładowo załóżmy, że dla naszej starej bazy danych istnieje już klasa PersonInfo, która zabezpiecza najważniejsze potrzeby klasy MyPerson:

```
class PersonInfo {
public:
  PersonInfo(DatabaseID pid);
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  virtual const char * theAddress() const;
  virtual const char * theNationality() const;

  virtual const char * valueDelimOpen() const;       // patrz
  virtual const char * valueDelimClose() const;      // poniżej
  ...
};
```

Możesz pomyśleć, że powyższa klasa jest stara, ponieważ jej funkcje składowe zwracają łańcuchy typu const char* zamiast obiektów typu string. Jeśli buty pasują, dlaczego nie mielibyśmy ich nosić? Nazwy funkcji składowych powyższej klasy sugerują, że efekt prawdopodobnie będzie dla nas satysfakcjonujący.

Dochodzimy wreszcie do odkrycia, że klasa PersonInfo została jednak zaprojektowana po to, by ułatwiać wypisywanie z bazy danych pól z danymi w różnych formatach, gdzie każde pole jest z góry i z dołu ograniczone specjalnymi łańcuchami. Domyślnymi ogranicznikami otwierającymi i zamykającymi wartości pól są nawiasy kwadratowe, zatem wartość pola „lemur gruboogoniasty” będzie reprezentowana przez łańcuch:

```
[Lemur gruboogoniasty]
```

Mając na uwadze fakt, że nawiasy kwadratowe nie są uniwersalnymi ogranicznikami odpowiadającymi wszystkim klientom klasy PersonInfo, wirtualne funkcje valueDelimOpen i valueDelimClose umożliwiają klasom potomnym wyznaczanie własnych łańcuchów pełniących rolę ograniczników otwierających i zamykających wartości. Implementacje należących do klasy PersonInfo funkcji theName, theBirthDate, theAddress i theNationality wywołują te wirtualne funkcje, by dodać właściwe ograniczniki do zwracanych wartości. Przykładowo, kod funkcji PersonInfo::theName może wyglądać następująco:

```
const char * PersonInfo::valueDelimOpen() const
{
  return "[";                  // domyślny ogranicznik otwierający
}
```

```
const char * PersonInfo::valueDelimClose() const
{
  return "]";                  // domyślny ogranicznik zamykający
}

const char * PersonInfo::theName() const
{
  // tworzy bufor dla zwracanej wartości; ponieważ tablica jest
  // statyczna, zostanie zainicjalizowana samymi zerami
  static char value[MAX_FORMATTED_FIELD_VALUE_LENGTH];

  // zapisuje ogranicznik otwierający
  strcpy(value, valueDelimOpen());

  dodaj do łańcucha value zawartość pola reprezentującego nazwisko

  // zapisuje ogranicznik zamykający
  strcat(value, valueDelimClose());

  return value;
}
```

Można oczywiście w powyższej definicji funkcji `PersonInfo::theName` doszukiwać się wad (szczególnie w zastosowaniu bufora o stałym rozmiarze — patrz sposób 23.), powinniśmy jednak odłożyć te rozważania na bok i skupić się na czymś innym — funkcja `theName` wywołuje funkcję `valueDelimOpen` celem wygenerowania ogranicznika otwierającego zwracany łańcuch, następnie funkcja generuje samą wartość reprezentującą nazwisko i wywołuje funkcję `valueDelimClose`. Ponieważ `valueDelimOpen` i `valueDelimClose` są funkcjami wirtualnymi, wynik zwracany przez funkcję `theName` jest uzależniony nie tylko od definicji klasy `PersonInfo`, ale także od wszystkich klas potomnych względem tej klasy.

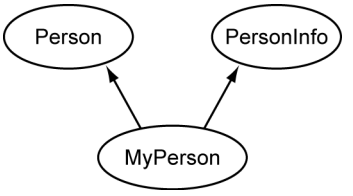
Dla programisty implementującego klasę `MyPerson` jest to dobra wiadomość, ponieważ uważnie analizując funkcje wypisujące z bazy danych wartości klasy `Person`, odkryliśmy, że zadaniem funkcji `theName` i jej siostrzanych funkcji składowych jest zwracanie nienaruszonych wartości (pozbawionych ograniczników). Oznacza to, że jeśli dana osoba pochodzi z Madagaskaru, po wywołaniu dla tej osoby funkcji zwracającej wartość pola `nationality` powinniśmy otrzymać łańcuch `"Madagaskar"`, a nie `"[Madagaskar]"`.

Relacja łącząca klasy `MyPerson` i `PersonInfo` polega na tym, że klasa `PersonInfo` zawiera niekiedy funkcje, dzięki którym implementacja klasy `MyPerson` jest łatwiejsza. To wszystko, nie jest to więc relacja „jest” ani „ma”. Oznacza to, że musimy mieć do czynienia z relacją implementacji z wykorzystaniem, o której wiemy, że może być reprezentowana na dwa sposoby — za pomocą podziału na warstwy (patrz sposób 40.) lub za pomocą prywatnego dziedziczenia (patrz sposób 42.). W sposobie 42. stwierdziłem, że technika podziału na warstwy jest w ogólności lepszym rozwiązaniem, jednak w przypadku, gdy konieczne jest ponowne definiowane funkcji wirtualnych, musimy zastosować dziedziczenie prywatne. W omawianym przykładzie klasa `MyPerson` musi zawierać nową definicję funkcji `valueDelimOpen` i `valueDelimClose`, zatem zastosowanie podziału na warstwy jest niemożliwe — `MyPerson` musi więc prywatnie dziedziczyć po klasie `PersonInfo`.

Klasa `MyPerson` musi jednak także implementować interfejs klasy `Person`, co wiąże się z publicznym dziedziczeniem. Prowadzi nas to do ciekawego przykładu dziedziczenia wielobazowego — połączenia publicznego dziedziczenia interfejsu z prywatnym dziedziczeniem implementacji:

```
class Person {                  // klasa wyznacza interfejs, który
public:                         // ma być implementowany
  virtual ~Person();

  virtual string name() const = 0;
  virtual string birthDate() const = 0;
  virtual string address() const = 0;
  virtual string nationality() const = 0;
};

class DatabaseID { ... };       // wykorzystywana poniżej (szczegóły
                                // są dla nas nieistotne)

class PersonInfo {              // klasa zawiera funkcje przydatne
public:                         // podczas implementacji klasy Person
  PersonInfo(DatabaseID pid);
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  virtual const char * theAddress() const;
  virtual const char * theNationality() const;

  virtual const char * valueDelimOpen() const;       // patrz
  virtual const char * valueDelimClose() const;      // poniżej
  ...
};

class MyPerson: public Person,            // zwróć uwagę na
                private PersonInfo {      // dziedziczenie wielobazowe
public:
  MyPerson(DatabaseID pid): PersonInfo(pid) {}

  // ponowne definicje odziedziczonych wirtualnych funkcji ograniczników
  const char * valueDelimOpen() const { return ""; }
  const char * valueDelimClose() const { return ""; }

  // implementacje wymaganych funkcji składowych klasy Person
  string name() const
  { return PersonInfo::theName(); }

  string birthDate() const
  { return PersonInfo::theBirthDate(); }

  string address() const
  { return PersonInfo::theAddress(); }

  string nationality() const
  { return PersonInfo::theNationality(); }
};
```

Graficznie można to przedstawić następująco:



Powyższy przykład pokazuje, że technika dziedziczenia wielobazowego może być przydatna i zrozumiała, chociaż nieprzypadkowo nie mamy w tym przypadku do czynienia z przerażającymi grafami dziedziczenia w kształcie diamentów.

Nadal jednak musimy opierać się pokusie pochopnego stosowania dziedziczenia wielobazowego. Możemy niekiedy wpaść w pułapkę nieprzemyślanego wykorzystania tej techniki do szybkiego poprawienia hierarchii dziedziczenia, która w rzeczywistości wymaga głębszych zabiegów projektowych. Przykładowo, przypuśćmy, że pracujemy nad hierarchią klas reprezentujących postacie z animowanych kreskówek. Przynajmniej na poziomie pojęciowym sensownym rozwiązaniem jest umożliwienie każdej z postaci tańczenia i śpiewania, jednak sposób realizacji tych czynności różni się dla poszczególnych bohaterów. Co więcej, domyślnym zachowaniem podczas śpiewania i tańczenia jest brak jakichkolwiek działań.

Możemy to wyrazić w języku C++ w następujący sposób:

```
class CartoonCharacter {
public:
  virtual void dance() {}
  virtual void sing() {}
};
```

Naturalnym sposobem modelowania wymagania dotyczącego tańczenia i śpiewania wszystkich obiektów klasy `CartoonCharacter` jest wykorzystanie funkcji wirtualnych. Domyślne zachowanie polegające na braku operacji wyrażamy za pomocą pustej definicji tych funkcji wewnątrz klas (patrz sposób 36.).

Przypuśćmy, że jednym z konkretnych typów postaci w kreskówce jest konik polny, który tańczy i śpiewa w charakterystyczny dla siebie sposób:

```
class Grasshopper: public CartoonCharacter {
public:
  virtual void dance();        // definicja znajduje się gdzieś indziej
  virtual void sing();         // definicja znajduje się gdzieś indziej
};
```

Przypuśćmy teraz, że po zaimplementowaniu klasy `Grasshopper` decydujemy, że będziemy także potrzebowali klasy dla świerszczy:

```
class Cricket: public CartoonCharacter {
public:
  virtual void dance();
  virtual void sing();
};
```

Kiedy zabierzesz się za implementowanie klasy `Cricket`, uświadomisz sobie, że możesz ponownie wykorzystać większość kodu napisanego wcześniej dla klasy `Grasshopper`. Kody funkcji należących do obu klas muszą się jednak w paru szczegółach różnić — uwzględniamy w ten sposób różnice pomiędzy technikami tańczenia i śpiewania koników polnych i świerszczy. Nagle przychodzi nam do głowy sprytny sposób ponownego wykorzystania istniejącego kodu — zaimplementujemy klasę `Cricket` *z wykorzystaniem* klasy `Grasshopper` i wykorzystamy wirtualne funkcje, które umożliwią klasie `Cricket` modyfikowanie zachowań z klasy `Grasshopper`!

Od razu powinniśmy się zorientować, że połączenie obu wymagań — relacji implementacji z wykorzystaniem z możliwością ponownego definiowania wirtualnych funkcji — oznacza, że klasa `Cricket` musiałaby prywatnie dziedziczyć po klasie `Grasshopper`, jednak świerszcz pozostałby oczywiście postacią z kreskówki, zatem klasę `Cricket` musielibyśmy zdefiniować w taki sposób, by dziedziczyła zarówno po klasie `Grasshopper`, jak i `CartoonCharacter`:

```
class Cricket: public CartoonCharacter,
               private Grasshopper {
public:
  virtual void dance();
  virtual void sing();
};
```

Dochodzimy teraz do momentu, w którym musimy wprowadzić niezbędne modyfikacje do klasy `Grasshopper`. W szczególności musimy zadeklarować kilka nowych wirtualnych funkcji, które będą ponownie definiowane w klasie `Cricket`:

```
class Grasshopper: public CartoonCharacter {
public:
  virtual void dance();
  virtual void sing();

protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};
```

Tańczenie koników polnych możemy teraz zdefiniować w następujący sposób:

```
void Grasshopper::dance()
{
  wykonaj typowe operacje dla tańczenia;
  danceCustomization1();
  wykonaj kolejne typowe operacje dla tańczenia;
  danceCustomization2();
  wykonaj końcowe operacje typowe dla tańczenia;
}
```

Podobnie powinniśmy zaimplementować zachowanie koników polnych podczas śpiewania.

Jest oczywiste, że musimy zaktualizować klasę `Cricket` w taki sposób, by uwzględniała nowe wirtualne funkcje, które musi ponownie definiować:

```
class Cricket: public CartoonCharacter,
               private Grasshopper {
public:
  virtual void dance() { Grasshopper::dance(); }
  virtual void sing() { Grasshopper::sing(); }

protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};
```

Wygląda na to, że wszystko powinno działać prawidłowo. Kiedy obiekt klasy `Cricket` ma zatańczyć, wykona wspólny kod funkcji `dance` z klasy `Grasshopper`, wykona właściwy tylko do świerszczy kod funkcji `dance` z klasy `Cricket`, wykona kod funkcji `Grasshopper::dance` itd.

Zaprezentowany projekt zawiera jednak poważną wadę — ślepo dążąc do celu złamałeś bowiem zasadę zwaną *brzytwą Ockhama*[1]. Ockhamizm głosi, że bytów nie należy mnożyć bez konieczności, odrzuca tym samym wszelkie byty, do których uznania nie zmusza doświadczenie. W tym przypadku tymi bytami są relacje dziedziczenia. Jeśli sądzisz, że dziedziczenie wielobazowe jest bardziej skomplikowane od zwykłego dziedziczenia (mam nadzieję, że tak właśnie sądzisz), zaproponowany projekt klasy `Cricket` jest niepotrzebnie tak skomplikowany.

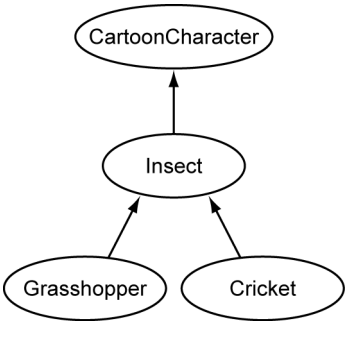
Zasadniczy problem polega na tym, że *nieprawdą* jest, że klasa `Cricket` jest zaimplementowana z wykorzystaniem klasy `Grasshopper`. Klasy `Cricket` i `Grasshopper` mają po prostu trochę *wspólnego kodu*. W szczególności wykorzystują wspólny kod definiujący te zachowania podczas tańczenia i śpiewania koników polnych i świerszczy, które dla obu typów postaci są identyczne.

Dziedziczenie jednej klasy po drugiej nie jest dobrym sposobem wyrażania ich zależności polegającej na wykorzystywaniu wspólnego kodu — w takim przypadku *obie* klasy powinny dziedziczyć po jednej wspólnej klasie bazowej. Wspólny kod dla koników polnych i świerszczy nie powinien należeć ani do klasy `Grasshopper`, ani do klasy `Cricket`; powinien należeć do nowej klasy bazowej, po której obie wymienione klasy powinny dziedziczyć, powiedzmy do klasy `Insect`:

```
class CartoonCharacter { ... };

class Insect: public CartoonCharacter {
public:
  virtual void dance();        // wspólny kod dla koników polnych
  virtual void sing();         // i świerszczy

protected:
  virtual void danceCustomization1() = 0;
  virtual void danceCustomization2() = 0;
```

[1] Zasada sformułowana przez średniowiecznego mnicha i teologa franciszkańskiego, angielskiego przedstawiciela późnej scholastyki, Wilhelma Ockhama (właściwie William of Occam), 1300 – 1349 — *przyp. tłum.*

```
  virtual void singCustomization() = 0;
};

class Grasshopper: public Insect {
protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};

class Cricket: public Insect {
protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};
```



Zwróć uwagę na prostotę tego projektu. Wykorzystujemy wyłącznie technikę pojedynczego dziedziczenia. Co więcej, stosujemy wyłącznie dziedziczenie *publiczne*. Klasy `Grasshopper` i `Cricket` definiują jedynie funkcje charakterystyczne dla reprezentowanych przez siebie postaci — wspólny kod funkcji `dance` i `sing` dziedziczą po klasie `Insect`. Wilhelm Ockham byłby z nas dumny.

Mimo że nowy projekt jest prostszy od omawianego wcześniej wymagającego dziedziczenia wielobazowego, może początkowo robić wrażenie bardziej skomplikowanego. W porównaniu z wcześniejszą koncepcją (wykorzystującą technikę dziedziczenia wielobazowego), proponowana architektura z pojedynczym dziedziczeniem wiąże się z koniecznością wprowadzenia zupełnie nowej klasy, która wcześniej nie była konieczna. Po co wprowadzać dodatkową klasę, skoro nie jest potrzebna?

Ten przykład demonstruje uwodzicielski charakter techniki dziedziczenia wielobazowego. Dziedziczenie wielobazowe z zewnątrz wygląda na łatwiejsze — nie wymaga stosowania dodatkowych klas i chociaż wiąże się z wywoływaniem kilku nowych wirtualnych funkcji z klasy `Grasshopper`, nowe funkcje i tak muszą zostać gdzieś zdefiniowane.

Wyobraźmy sobie teraz programistę konserwującego wielką bibliotekę klas C++, do której należy dodać nową klasę (`Cricket`) do istniejącej hierarchii `CartoonCharacter`-`Grasshopper`. Programista wie, że z istniejącej hierarchii korzysta mnóstwo klientów,

zatem im większe zmiany wprowadzi do biblioteki, tym większe będzie ich niezadowolenie. Programista stawia sobie jednak za cel zminimalizowanie tego zjawiska. Dokładnie analizując wszystkie możliwości, dochodzi do wniosku, że jeśli doda relację pojedynczego dziedziczenia po klasie `Grasshopper` do nowej klasy `Cricket`, hierarchia nie będzie wymagała żadnych dodatkowych modyfikacji. Jego radość jest uzasadniona — udało mu się znacząco zwiększyć funkcjonalność biblioteki kosztem minimalnego zwiększenia jej złożoności.

Wyobraź sobie teraz, że to Ty jesteś tym programistą. Nie daj się więc skusić technice dziedziczenia wielokrotnego.

Sposób 44. Mów to, o co czym naprawdę myślisz. Zdawaj sobie sprawę z tego, co mówisz

We wstępie do tej części, poświęconej dziedziczeniu i projektowaniu zorientowanemu obiektowo, podkreśliłem wagę właściwego zrozumienia, co poszczególne konstrukcje obiektowe języka C++ naprawdę *oznaczają*. Nie chodzi tylko o zwykłą znajomość reguł tego języka programowania. Przykładowo, reguły dla C++ mówią, że jeśli klasa `D` publicznie dziedziczy po klasie `B`, istnieje standardowa konwersja ze wskaźnika do obiektu klasy `D` do wskaźnika do obiektu klasy `B`; publiczne funkcje składowe klasy `B` są dziedziczone jako publiczne funkcje składowe klasy `D` itd. Wszystkie te cechy są oczywiście prawdziwe, jednak ta wiedza jest niemal bezużyteczna, kiedy próbujemy przełożyć nasz projekt na kod w C++. Musimy więc zdać sobie sprawę z faktu, że publiczne dziedziczenie w rzeczywistości oznacza relację „jest” — jeśli klasa `D` publicznie dziedziczy po klasie `B`, każdy obiekt klasy `D` jest także obiektem klasy `B`. Jeśli więc w swoim projekcie wprowadzasz relację „jest”, wiesz, że w implementacji powinieneś zastosować dziedziczenie publiczne.

Właściwe określenie, co mamy na myśli, jest jednak dopiero połową sukcesu. Drugą, równie ważną połowę stanowi właściwe rozumienie efektów naszych decyzji projektowych. Przykładowo, deklarowanie niewirtualnych funkcji przed przeanalizowaniem związanych z tym ograniczeń dla podklas jest nieodpowiedzialne, jeśli nie całkowicie niemoralne. Deklarując niewirtualną funkcję, w rzeczywistości sygnalizujesz, że dana funkcja reprezentuje działanie niezależne od specjalizacji — jeśli nie zdajesz sobie z tego sprawy, efekt może być katastrofalny.

Równoważności publicznego dziedziczenia i relacji „jest” oraz niewirtualnych funkcji składowych i niezależności od specjalizacji są przykładami sposobu, w jaki konkretne konstrukcje języka C++ odpowiadają rozwiązaniom na poziomie projektu. Poniższa lista jest podsumowaniem najważniejszych odpowiedniości tego typu:

- **Wspólna klasa bazowa oznacza wspólne cechy klas potomnych**.
  Jeśli zarówno klasa `D1`, jak i klasa `D2` deklaruje `B` jako swoją klasę bazową, klasy `D1` i `D2` dziedziczą wspólne dane składowe i (lub) wspólne funkcje składowe po klasie `B` (patrz sposób 43.).

- **Publiczne dziedziczenie jest równoważne z relacją „jest”**. Jeśli klasa `D` publicznie dziedziczy po klasie `B`, każdy obiekt typu `D` jest także obiektem typu `B`, ale nie na odwrót (patrz sposób 35.).
- **Prywatne dziedziczenie jest równoważne z relacją implementacji z wykorzystaniem**. Jeśli klasa `D` prywatnie dziedziczy po klasie `B`, obiekty typu `D` są po prostu implementowane z wykorzystaniem obiektów typu `B`; pomiędzy obiektami klasy `B` i `D` nie istnieje żadna relacja pojęciowa (patrz sposób 42.).
- **Podział na warstwy jest równoważny z relacją implementacji z wykorzystaniem**. Jeśli klasa `A` zawiera składową daną typu `B`, obiekty typu `A` albo zawierają elementy typu `B`, albo są zaimplementowane z wykorzystaniem obiektów typu `B` (patrz sposób 40.).

Poniższe stwierdzenia dotyczą sytuacji, w których wykorzystywana jest technika publicznego dziedziczenia:

- **Istnienie w klasie czystej funkcji wirtualnej oznacza, że dziedziczony będzie wyłącznie interfejs tej klasy**. Jeśli klasa `C` deklaruje czystą funkcję wirtualną `mf`, podklasy klasy `C` muszą dziedziczyć interfejs tej funkcji, a konkretne podklasy klasy `C` muszą dostarczyć własną implementację funkcji `mf` (patrz sposób 36.).
- **Deklaracja prostej funkcji wirtualnej oznacza, że dziedziczony będzie zarówno interfejs tej funkcji, jak i jej domyślna implementacja**. Jeśli klasa `C` deklaruje prostą (nie czystą) funkcję wirtualną `mf`, podklasy klasy `C` muszą dziedziczyć interfejs tej funkcji, mogą także — jeśli jest to korzystne — dziedziczyć jej domyślną implementację (patrz sposób 36.).
- **Deklaracja niewirtualnej funkcji oznacza, że dziedziczony będzie zarówno interfejs tej funkcji, jak i jej wymagana implementacja**. Jeśli klasa `C` deklaruje prostą (nie czystą) funkcję wirtualną `mf`, podklasy klasy `C` muszą dziedziczyć zarówno interfejs tej funkcji, jak i jej implementację. Oznacza to, że zachowanie funkcji `mf` jest niezależne od specjalizacji (patrz sposób 36.).