Abstrakcja Danych i Hierarchia

Barbara Liskov
MIT Pracownia Informatyczna
Cambridge, Ma. 02139

Abstrakt

Abstrakcja danych to cenna metoda organizacji programów w celu ułatwienia ich modyfikacji i utrzymania. Dziedziczenie umożliwia powiązanie jednej implementacji abstrakcji danych z inną hierarchicznie. Ten dokument bada przydatność hierarchii w tworzeniu programów i podsumowuje, że chociaż abstrakcja danych jest pojęciem ważniejszym, hierarchia, w niektórych przypadkach, poszerza jej użyteczność.

Badania te wspierane były przez profesurę NEC w dziedzinie nauk programowania i inżynierii

1. Wprowadzenie

Ważnym celem w projektowaniu jest zidentyfikowanie struktury programu, która upraszcza zarówno utrzymywanie programu, jak i modyfikacje wprowadzone by wspierać zmieniające się wymagania. Abstrakcje danych są dobrym sposobem na osiągniecie tego celu. Pozwalają nam na odejście od sposobu w jaki struktury danych są implementowane, do zachowania, na którym polegać mogą inne programy. Umożliwiają lokalną zmianę fragmentu danych, bez wpływu na programy, które z nich korzystają. Są szczególnie ważne, ponieważ ukrywają skomplikowane rzeczy (struktury danych), które prawdopodobnie ulegną zmianie w przyszłości. Upraszczają również strukturę programów, które z nich korzystają, ponieważ prezentują wyższy stopień interfejsu. Zmniejszają, na przykład, liczbę argumentów do procedur, ponieważ przekazywane są abstrakcyjne obiekty zamiast ich reprezentacji.

Programowanie obiektowe jest główną techniką abstrakcji danych i z tego wynika największe źródło jego mocy. Niemniej jednak, można rozwinąć tę technikę dzięki pojęciu "dziedziczenia". Dziedziczenie może być użyte na wiele różnych sposobów, niektóre z nich mogą spotęgować moc abstrakcji danych. W takich przypadkach, dziedziczenie stanowi jej przydatne uzupełnienie.

Ten dokument omawia związek między abstrakcją danych a programowaniem obiektowym. W drugiej części pochylamy się nad definicją abstrakcji danych i jej rolą w procesie twórczym oprogramowania. Następnie, w trzeciej części, omawiamy dziedziczenie i określamy dwa rodzaje jego wykorzystania: hierarchię implementacji i hierarchię typów. Spośród tych dwóch metod hierarchia typów naprawdę dodaje coś do abstrakcji danych, dlatego w części czwartej rozważamy zastosowania hierarchii typów w projektowaniu i tworzeniu programów. Dalej, omawiamy niektóre problemy, pojawiające się podczas implementacji hierarchii typów. Kończymy podsumowaniem naszych wyników.

2. Abstrakcja danych

Celem abstrakcji w programowaniu jest oddzielenie zachowania od implementacji. Pierwszym abstrakcyjnym mechanizmem w programowaniu była procedura. Procedura wykonuje pewne zadanie lub funkcję; inne części programu łączą się z procedurą w celu wykonania zadania. Aby skorzystać z procedury, programista dba tylko o to, jakie czynności ona wykonuje, a nie jak jest zaimplementowana. Każda implementacja, zapewnia potrzebną funkcję pod warunkiem, że implementuje się ją poprawnie i jest ona wystarczająco wydajna.

Procedury są użytecznymi mechanizmami abstrakcji, lecz na początku lat siedemdziesiątych niektórzy badacze zdali sobie sprawę z tego, iż nie są one wystarczające [15, 16, 7] i zaproponowali nowy sposób organizacji programów wokół „połączeń" między modułami. Z nich narodziła się koncepcja *abstrakcji danych* lub *abstrakcyjnych typów danych* [5, 12].

Abstrakcje danych zapewniają te same korzyści co procedury ale w przypadku danych. Przypomnijmy, że główną ideą jest oddzielenie tego, czym jest abstrakcja, od tego, jak jest wdrażana, tak aby implementacje tej samej abstrakcji mogły być swobodnie zastępowane. Implementacja obiektu danych skupia się na tym jak obiekt przedstawiany jest w pamięci komputera; informacja ta nazywana jest reprezentacją. Aby umożliwić zmianę implementacji bez wpływu na użytkowników,

potrzebujemy sposobu na zmianę reprezentacji bez konieczności zmiany wszystkich programów, które ją wykorzystują. Osiąga się to poprzez enkapsulacje reprezentacji za pomocą zestawu operacji, które nią manipulują i ograniczenie używających jej programów, tak aby nie mogły manipulować one bezpośrednio ale by zamiast tego wywoływały jej operacje. Wtedy do implementacji lub reimplementacji abstrakcji danych, konieczne jest zdefiniowanie reprezentacji i zaimplementowanie operacji pod tym kątem, by zmiana nie wpływała na używający ją kod.

Abstrakcja danych jest zatem zbiorem obiektów, którymi można bezpośrednio manipulować jedynie za pomocą zestawu operacji. Przykładem abstrakcji danych są liczby całkowite: obiekty to 1, 2, 3 itd. oraz operacje dodawania dwóch liczb całkowitych, testowania ich pod kątem równości i innych tego typu operacji. Programy używające liczb całkowitych manipulują nimi za pomocą operacji i są chronione przed szczegółami implementacji, takimi jak to, czy reprezentacja jest uzupełnieniem dwójkowym. Innym przykładem są ciągi znaków z elementami takimi jak „a" czy „xyz" i operacje do selekcji znaków z ciągu i łączenia ich. Ostatnim przykładem są zestawy liczb całkowitych, z elementami takimi jak { } (zbiór pusty) i {3, 7} oraz operacje wstawiania elementów i sprawdzenia czy dana liczba całkowita jest częścią zbioru. Zwróćmy uwagę na to, że liczby całkowite i ciągi znaków są wbudowanymi typami danych większości języków programowania, natomiast zbiory i inne abstrakcje danych skupionych na aplikacji, takie jak stosy czy tablice symboli, już nie. Mechanizmy językowe, które pozwalają na implementację abstrakcyjnych typów danych zdefiniowanych przez użytkownika omówione zostały w Rozdziale 2.2.

Abstrakcja lub abstrakcja procedur jest określana przez *specyfikację* i zaimplementowana przez moduł programu napisanego w danym języku programowania. Specyfikacja opisuje działania abstrakcji, ale pomija wszelkie informacje o tym, jak jest ona zaimplementowana. Pomijając taki szczegół, zezwalamy na wiele różnych implementacji. Implementacja jest poprawna, jeśli zapewnia zachowanie zdefiniowane w specyfikacji. Poprawność można udowodnić matematycznie, jeśli specyfikacja napisana jest w języku o precyzyjnej semantyce; w przeciwnym razie ustalamy poprawność poprzez rozumowanie przyczynowo-skutkowe lub przez nieco niezadowalającą technikę testowania. Prawidłowe implementacje różnią się od siebie sposobem działania, tzn. mogą mieć różną wydajność w zależności od tego jakich algorytmów używają. Każda poprawna implementacja jest dopuszczalna przez wywołującego pod warunkiem, że spełnia jego wymagania dotyczące wydajności. Zauważmy, iż poprawne implementacje nie muszą być identyczne; chodzi o to, aby umożliwić implementacją na różnice między sobą zapewniając jednocześnie, że pozostają one takie same tam, gdzie jest to niezbędne. To specyfikacja określa co jest niezbędne.

Aby abstrakcja działała, implementacje muszą być enkapsulowane. Jeśli implementacja jest zaenkapsulowana wtedy żaden inny moduł nie może zależeć od szczegółów jej implementacji. Enkapsulacja gwarantuje, że moduły mogą być implementowane i reimplementowane niezależnie; wiąże się to z zasadą „przesłaniania informacji" zalecaną przez Parnas [15].

2.1. Lokalność

Gdy abstrakcja wspierana jest przez specyfikacje i enkapsulację, zapewnia *lokalność* w programie. Lokalność pozwala na implementację, zrozumienie lub modyfikację programu po jednym module na raz:

1. Osoba implementująca abstrakcję wie, co jest potrzebne, ponieważ jest to opisane w specyfikacji. W związku z tym nie musi ona wchodzić w interakcje z programistami innych modułów (lub przynajmniej interakcje te są bardzo ograniczone).

2. Podobnie, osoba używająca modułu wie czego spodziewać się ma od abstrakcji, ponieważ jej zachowanie opisane jest przez specyfikacje.

3. Do określenia, jaką funkcję pełni program i czy pełni ją poprawnie potrzebna jest jedynie analiza lokalna. Program analizowany jest po jednym module na raz. W każdym przypadku zwracamy uwagę czy moduł spełnia swoją funkcję, czyli czy jest zgodny ze specyfikacją. Możemy jednak ograniczyć naszą uwagę tylko do określonego modułu i zignorować moduły, które go używają i moduły, których używa. Moduły używające go można zignorować, ponieważ zależą one tylko od specyfikacji tego modułu, a nie jego kodu. Używane moduły mogą być ignorowane dzięki zrozumieniu jakie funkcje pełnią, poprzez specyfikacje zamiast ich implementacji. Dzięki temu możemy zaoszczędzić ogromną ilość wysiłku, ponieważ specyfikacje są znacznie mniejsze niż implementacje. Na przykład, gdybyśmy musieli spojrzeć na kod wywoływanej abstrakcji, musielibyśmy zwrócić uwagę nie tylko na jej kod, ale także na kod wszelkich modułów, z których korzysta, i tak dalej.

4. Wreszcie, modyfikacja programu może być wykonywana moduł po module. Jeśli konkretną abstrakcję należy reimplementować, aby zapewnić lepszą wydajność, poprawić błąd lub rozszerzyć udogodnienia, stary moduł można wymienić na nowy bez wpływania na inne moduły.

Lokalność zapewnia solidną podstawę do szybkiego tworzenia prototypów. Zazwyczaj istnieje kompromis między wydajnością algorytmu a szybkością, z jaką jest on projektowany i wdrażany. Wstępna implementacja może być prosta i mieć słabą wydajność. Może być zastąpiona później inną implementacją z lepszą wydajnością. Pod warunkiem, że obie implementacje są poprawne, nie wpłyną one na poprawność programu.

Lokalność wspiera również rozwój programu. Abstrakcje mogą być użyte do enkapsulacji potencjalnych modyfikacji. Załóżmy na przykład, że chcemy, aby program działał na różnych urządzeniach. Możemy osiągnąć to przez stworzenie abstrakcji maskującej różnice między dwoma urządzeniami, aby przenieść więc program na inne urządzenie reimplementowana musi zostać tylko ta abstrakcja. Dobra zasada projektowania polega na przemyśleniu przyszłych modyfikacji i projektowaniu przy użyciu abstrakcji, która enkapsuluje zmiany.

Korzyści płynące z lokalności są szczególnie ważne w przypadku abstrakcji danych. Struktury danych często są skomplikowane, dlatego prostszy abstrakcyjny widok dostarczany przez specyfikację pozwala na uproszczenie reszty programu. W miarę rozwoju programów prawdopodobne są również zmiany w strukturach danych; można zminimalizować wpływ tych zmian poprzez enkapsulacje ich wewnątrz abstrakcji danych.

2.2. Wsparcie językowe dla abstrakcji danych

Abstrakcje danych są wspierane przez mechanizmy językowe w kilku językach. Najwcześniejszym takim językiem była Simula 67 [3]. Poniżej omówiono dwie główne odmiany, te w CLU i Smalltalku.

CLU [8, 11] zapewnia mechanizm zwany *klastrem* do implementacji abstrakcyjnego typu. Szablon klastra pokazano na rysunku 2-l. Nagłówek identyfikuje implementowany typ danych, zawiera również listę operacji danego typu; służy do określenia, jakie definicje procedur wewnątrz klastra można wywołać z zewnątrz. Linia „rep =" definiuje sposób reprezentacji obiektów danego typu; w przykładzie implementujemy zestawy jako połączone listy. Reszta klastra składa się z procedur; każda operacja musi mieć swoją procedurę, ponadto mogą istnieć pewne procedury, które można zastosować jedynie wewnątrz klastra.

```
int_set = cluster is create, insert, is_in, size, ...
        rep = int_list
        create = proc ... end create
        insert = proc ... end insert
        ...
end int_set
```

Rysunek 2-1: Szablon Klastra CLU

W Smalltalku [4], abstrakcje danych są implementowane przez *klasy*. Klasy można układać hierarchicznie, ale na razie to pominiemy. Klasa implementuje abstrakcję danych podobnie jak klaster. Zamiast „rep =" reprezentacja opisana jest sekwencją zadeklarowanych zmiennych; są to *zmienne instancji*[1]. Pozostała część klasy składa się z metod, które są definicjami procedur. Dla każdej operacji typu danych zaimplementowanej przez klasę przypisana jest metoda. (Ponieważ nie można wykluczyć użycia metody poza klasą.) Metody wywoływane są przez „wysyłanie wiadomości", co ma taki sam efekt, jak wywoływanie operacji w CLU.

Zarówno CLU, jak i Smalltalk wymuszają enkapsulację, ale CLU sprawdza typ w czasie kompilacji, podczas gdy Smalltalk używa sprawdzania w czasie wykonywania programu. Sprawdzanie w czasie kompilacji jest o wiele lepsze, ponieważ pozwala na wyłapanie błędów klasy przed uruchomieniem programu i pozwala na generowanie wydajniejszego kodu przez kompilator. (Sprawdzanie w czasie kompilacji może ograniczyć moc ekspresji chyba, że język programowania ma wystarczający rozbudowany system typów; kwestię tę omówiono szerzej w rozdziale 5.) Inne języki obiektowe na przykład [1, 13], w ogóle nie wymuszają enkapsulacji. Prawdą jest, że w przypadku braku obsługi przez język enkapsulacji poprawność kodu może być sprawdzona ręcznie, ale techniki te są podatne na błędy i chociaż sytuacja może być w pewnym stopniu możliwa do opanowania, w przypadku nowo wdrażanego programu, w miarę wprowadzania modyfikacji może ulec szybkiej degradacji. W przypadku automatycznego sprawdzania w czasie wykonywania lub w czasie kompilacji można bez obaw polegać na tych mechanizmach, bez potrzeby ręcznego przeglądania kodu.

---

[1] Pomijamy tutaj zmienne klas, ponieważ nie są one ważne dla rozróżnień, które próbujemy wprowadzić. Mechanizm CLU jest analogiczny: klaster może mieć pewne „własne" zmienne [9].

Kolejną różnicę między CLU i Smalltalk można znaleźć w semantyce obiektów danych. W Smalltalku, operacje są częścią obiektu i mają dostęp do zmiennych instancji, które składają się na reprezentację obiektu, ponieważ zmienne te są również częścią obiektu. W CLU operacje nie należą do obiektu, zamiast tego są przynależne do typu. Daje im to specjalne przywileje w odniesieniu do obiektów ich typu, których inne części programu nie posiadają, mianowicie, mogą zobaczyć reprezentację tych obiektów. (Pierwszy raz zostało to opisane przez Morrisa'a [14].) Widok CLU działa lepiej w przypadku operacji manipulujących kilkoma obiektami jednocześnie, ponieważ operacje mogą widzieć wiele reprezentacji obiektów na raz. Przykładami takich operacji jest dodawanie dwóch liczb całkowitych lub tworzenie sumy dwóch zbiorów. Widok Smalltalku nie obsługuje również wspomnianych wcześniej operacji, ponieważ operacja może znajdować się tylko w jednym obiekcie. Z drugiej strony, widok Smalltalku działa lepiej, gdy chcemy uruchomić kilka implementacji tego samego typu w ramach tego samego programu. W CLU operacja ma wgląd do reprezentacji dowolnego obiektu danego typu, a zatem musi być ona zakodowana by poradziła sobie z wieloma reprezentacjami. Smalltalk unika tego problemu, ponieważ operacja może zobaczyć reprezentację tylko jednego obiektu.

3. Dziedziczenie i hierarchia

W tej sekcji omówiono dziedziczenie i sposób, w jaki wspiera hierarchię. Zaczynamy od omówienia, jak skonstruować program przy użyciu dziedziczenia. Następnie, omówimy dwa główne zastosowania dziedziczenia, hierarchię implementacji i hierarchię typów; patrz [18] gdzie omówiono podobny temat. Tylko hierarchia typów, dodaje coś nowego do abstrakcji danych.

3.1. Dziedziczenie

W języku z dziedziczeniem, abstrakcję danych można zaimplementować w kilku powiązanych ze sobą elementach. Chociaż różne języki zapewniają różne mechanizmy łączenia elementów, wszystkie są do siebie podobne. W ten sposób możemy je zobrazować, badając pojedynczy mechanizm, mechanizm podklasy w Smalltalku.

W Smalltalku klasę można zadeklarować jako *podklasę* innej klasy[2], która jest jej *nadklasą*. Pierwsza rzeczą, jaką musimy zrozumieć o tym mechanizmie jest to jaki kod jest wynikiem takiej definicji. To pytanie jest ważne dla zrozumienia jaką funkcję pełni podklasa. Na przykład, gdybyśmy chcieli sprawdzić jej poprawność musielibyśmy spojrzeć na ten kod.

Z punktu widzenia kodu wynikowego, stwierdzenie, że jedna klasa jest podklasą innej jest po prostu skróconą notacją do budowania programów. Stworzony program, zależy od reguł języka, np. kiedy metody podklasy nadpisują metody nadklasy. Dokładne szczegóły tych zasad nie są ważne dla naszej dyskusji (chociaż są one niezmiernie ważne, jeśli język ma być sensowny i użyteczny). Chodzi o to, że wynik jest równoznaczny z bezpośrednią implementacją klasy zawierającej zmienne instancji i metody, które wynikają z zastosowania reguł.

Załóżmy na przykład, że klasa T posiada operacje o_1 i o_2 , zmienną instancji v_1 oraz że klasa S, zadeklarowana jako podklasa klasy T, posiada operacje o_1 i o_3 oraz zmienną instancji v_2. Wynik w Smalltalku jest zatem klasą z dwiema zmiennymi instancji v_1 i v_2, oraz trzema operacjami o_1, o_2, o_3, gdzie kod o_2 jest dostarczany przez T, a kod pozostałych dwóch operacji jest dostarczany przez S.

[2] Ignorujemy wielokrotne dziedziczenie, aby uprościć dyskusję.

Połączony kod musi być zrozumiany lub zmodyfikowany, jeśli S zostanie reimplementowane, chyba że S jest w pewien sposób ograniczone, jak omówiono to poniżej.

Jednym z problemów związanym z prawie wszystkimi mechanizmami dziedziczenia jest to, że w pewnym stopniu ograniczają one abstrakcję danych. W językach z dziedziczeniem: implementacja abstrakcji danych (tzn. klasa) ma dwa rodzaje użytkowników. Są to użytkownicy "zewnętrzni", którzy używają obiektów, przez wywołanie operacji oraz użytkownicy „wewnętrzni". Są to podklasy, które mogą zazwyczaj naruszać enkapsulację. Istnieją trzy sposoby naruszania enkapsulacji [18]: podklasa może uzyskać dostęp do zmiennej instancji swojej superklasy, wywołać prywatną operację swojej superklasy lub odnieść się bezpośrednio do superklasy swojej superklasy. (ostatni przykład nie jest możliwy w Smalltalku).

Gdy enkapsulacja nie jest naruszona, możemy wnioskować o działaniach superklasy za pomocą jej specyfikacji, możemy również zignorować reprezentacje danej superklasy. Gdy enkapsulacja zostaje naruszona tracimy wówczas korzyści z lokalności. Musimy wziąć pod uwagę połączony kod pod- i superklasy w rozumowaniu o podklasie, a jeśli superklasa musi zostać ponownie zaimplementowana, być może będziemy musieli ponownie zaimplementować także jej podklasy. Byłoby to na przykład konieczne, gdyby zmieniłaby się zmienna instancji superklasy lub jeśli podklasa odnosi się bezpośrednio do superklasy swojej superklasy T, a następnie T zostaje ponownie zaimplementowane, aby pozbyć się swojej superklasy.

Naruszenie enkapsulacji może być przydatne w szybkim tworzeniu prototypu, ponieważ umożliwia tworzenie kodu poprzez rozszerzenie i modyfikację istniejącego kodu. Nie możemy jednak oczekiwać, że modyfikacje implementacji superklasy mogą być automatycznie propagowane do podklasy. Propagacja jest przydatna tylko wtedy, gdy kod wynikowy działa, co oznacza, że wszystkie oczekiwania podklasy o superklasie muszą być zgodne z nową implementacją. Oczekiwania te można osiągnąć przez dostarczenie innej specyfikacji dla superklasy; jest to inna specyfikacja niż dla postronnych, ponieważ zawiera dodatkowe ograniczenia. Korzystając z tej dodatkowej specyfikacji, programista może określić czy proponowana zmiana w superklasie może być użytecznie wykorzystana do propagowania do podklasy. Warto pamiętać, że im bardziej precyzyjna dodatkowa specyfikacja szczegółów poprzedniej implementacji superklasy, tym mniej prawdopodobne, że spełni ją nowa implementacja superklasy. Ponadto sytuacja nie będzie do opanowania, jeśli każda z podklas opierać się będzie na innej specyfikacji superklasy. Jednym z możliwych rozwiązań jest zdefiniowanie jednej specyfikacji, z której korzystać będą wszystkie podklasy, a która zawierać będzie więcej szczegółów niż specyfikacja dla postronnych ale dalej będzie abstrakcyjna dla wielu implementacyjnych szczegółów. Podobna praca poruszająca ten temat opisana jest w [17].

3.2. Hierarchia implementacji

Pierwszym sposobem wykorzystania dziedziczenia jest technika implementacji typów danych, które są podobne do innych, istniejących już typów. Załóżmy na przykład, że chcemy zaimplementować zestawy liczb całkowitych z operacjami, aby stwierdzić, czy element należy do zbioru i określić aktualny rozmiar zbioru. Załóżmy dalej, że typ danych listy został już zaimplementowany, i że zapewnia *operację przynależności do niej* jak i *operację rozmiaru*, a także dogodny sposób reprezentowania zbioru. Moglibyśmy wtedy zaimplementować zestaw jako podklasę listy; możemy sprawić, by lista zawierała elementy zestawu bez duplikacji, tzn. jeśli element zostałby dodany do zestawu dwukrotnie, w liście pojawiłby się tylko raz. Nie musielibyśmy wtedy zapewniać implementacji dla elementu ani rozmiaru ale musielibyśmy wdrożyć inne operacje, takie jak te,

wstawiające nowy element do zestawu. Powinniśmy także pominąć niektóre operacje, takie jak *zwracanie pierwszego elementu z listy,* aby były niedostępne, ponieważ nie mają one znaczenia dla zestawów. (Jest to możliwe do zrobienia w Smalltalku przez implementacje w podklasie dla niedostępnych operacji; taka implementacja zasygnalizuje wyjątek, jeśli zostanie wywołana.)

Innym sposobem na osiągnięcie tego samego celu jest użycie jednego (abstrakcyjnego) typu jako reprezentacji innego typu. Możemy na przykład, zaimplementować zestawy, używając listy jako reprezentacji. Napisanie implementacji dla tych dwóch operacji mimo prostego kodu to więcej pracy niż pisząc w nich cokolwiek. Z drugiej strony nie musimy nic robić, aby usunąć niepożądane operacje takie jak *zwracanie pierwszego elementu z listy.*

Ponieważ hierarchia implementacji nie pozwala nam na zrobienie niczego więcej czego nie moglibyśmy osiągnąć za pomocą abstrakcji danych na tym więc zakończmy nasze rozważania. Pozwala nam na naruszenie enkapsulacji, która zarówno przynosi korzyści jak i problemy. W razie potrzeby, umiejętność ta może być również dostępna w reprezentacji.

3.3. Hierarchia typów

Hierarchia typów składa się z podtypów i nadtypów. Intuicyjna idea *podtypu* składa się z obiektów zapewniających wszystkie zachowania obiektów innego typu (*nadtypu*) i czegoś dodatkowego. Pożądane jest tutaj coś na wzór zasady podstawienia [6]: Jeżeli dla każdego obiektu o_1 typu S występuje obiekt o_2 typu T taki, że dla wszystkich programów P zdefiniowanych za pomocą T, zachowanie P jest niezmienne gdy o_1 jest zastąpione przez o_2 wtedy S jest podtypem T. (Zobacz również podobne prace na ten temat [2, 17].)

Używamy tutaj słów „podtyp" i „nadtyp", aby podkreślić, że przechodzimy teraz do omówienia różnic semantycznych. Natomiast „podklasa" i „superklasa" są po prostu pojęciami języków programowania umożliwiającymi budowanie programów w określony sposób. Mogą być używane do implementacji podtypów, ale także w inny sposób, jak wspomniano powyżej.

Zaczniemy od kilku przykładów typów, które nie są swoimi podtypami. Po pierwsze, zestaw nie jest podtypem listy ani na odwrót. Jeśli ten sam element zostanie dodany do zestawu dwukrotnie, wynik będzie taki sam jakby został on dodany tylko raz, jest on również liczony tylko raz przy obliczaniu rozmiaru zbioru. Jeśli jednak ten sam element zostanie dodany do listy dwukrotnie, na liście pojawi się dwa razy. Tak więc program oczekujący na listę może nie działać jeśli przekażemy mu zestaw; podobnie program oczekujący na zestaw może nie działać jeśli przekażemy mu listę. Innym przykładem typów, które nie są swoimi podtypami są stosy i kolejki. Stosy to LIFO; kiedy element jest usuwany ze stosu, ostatni dodany element jest usuwany. Kolejki natomiast to FIFO. Działający program prawdopodobnie zauważy różnicę między tymi dwoma typami.

Powyższe przykłady ignorują prostą różnicę między parami typów, ze zbieżnymi nazwami operacji podtyp musi mieć wszystkie operacje[3] swojego nadtypu, w przeciwnym razie program używający ich nie mógłby użyć operacji, na których polega. Jednak samo posiadanie operacji o właściwych nazwach i sygnaturach nie wystarczy. (Sygnatura operacji definiuje liczby i typy jej argumentów wejściowych i wyjściowych.) Operacje muszą również spełniać te same funkcje. Na przykład stosy i kolejki mogą mieć operacje o tych samych nazwach, np. *add_e1,* aby dodać element,

[3] Potrzebuje metody instancyjne, a nie metody klasy.

czy rem_el, aby go usunąć ale nadal nie są one swoimi podtypami, ponieważ ich operacje różnią się definicją.

Podajmy teraz kilka przykładów hierarchii podtypów. Pierwszym z nich są indeksowane kolekcje, posiadające operacje pozwalające na dostęp do elementów po indeksie, np. aby pobrać zaindeksowany element kolekcji musi istnieć operacja pobierania. Również wszystkie podtypy mają te operacje, ale dodatkowo każdy może zapewniać dodatkowe operacje. Przykładami podtypów są tablice, sekwencje i zaindeksowane zestawy, np. sekwencje mogą być połączone, a tablice można modyfikować, przechowując nowe obiekty w elementach.

Drugi przykład to abstrakcyjne urządzenia, które ujednolicają wiele różnych rodzajów urządzeń wejściowych i wyjściowych. Poszczególne urządzenia mogą zapewniać dodatkowe operacje. W takim przypadku operacje abstrakcyjnych urządzeń to te, które obsługiwane są przez wszystkie urządzenia, np. test „end-of-file", podczas gdy operacje na podtypach byłyby specyficzne dla danego urządzenia. Na przykład drukarka posiadałaby operacje modyfikujące takie jak *put_char* ale nie operacje odczytujące jak *get_char.* Inną możliwością jest to, że abstrakcyjne urządzenia miałyby wszystkie możliwe operacje np. zarówno *put_char*, jak i *get_char* a zatem wszystkie podtypy miałyby ten sam zestaw operacji. W takim przypadku operacje, których nie są w stanie wykonać wszystkie rzeczywiste urządzenia, muszą być sprecyzowane w sposób ogólny co pozwalałoby na wyłapanie wyjątków. Na przykład, *get_char* sygnalizowałaby wyjątek, kiedy zostałaby wywołana na drukarce.

Do implementacji hierarchii podtypów można użyć mechanizmu dziedziczenia. Istniałaby klasa, która implementowała by nadklasę oraz osobna klasa do implementacji każdego z podtypów. Klasa implementująca podtyp zadeklarowałby klasę nadtypu jako swoją nadklasę.

4. Korzyści z hierarchii typów

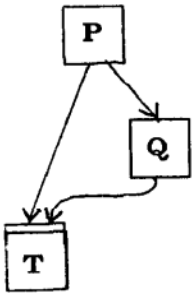
Abstrakcja danych sama w sobie jest potężnym narzędziem. Hierarchia typów jest przydatnym dodatkiem do abstrakcji danych. W tej sekcji omówiono, w jaki sposób można wykorzystać podtypy w tworzeniu programu. (Szczegółowe omówienie projektowania opartego na abstrakcji danych można znaleźć w [11].) Omówiono również zastosowanie podtypów w organizowaniu biblioteki programów.

4.1. Projektowanie Przyrostowe

Abstrakcje danych są zwykle opracowywane stopniowo w miarę postępu projektu. We wczesnych stadiach znamy tylko niektóre operacje abstrakcji danych i część ich zachowań. Taki etap projektu przedstawiony jest na rysunku 4-1a. Projekt przedstawiony jest na wykresie ilustrującym jak wygląda program podzielony na moduły. Istnieją dwa rodzaje węzłów; węzeł z pojedynczym paskiem na górze reprezentuje abstrakcje procedury, a węzeł z podwójnym paskiem na górze abstrakcję danych. Strzałka poprowadzona z jednego węzła do drugiego oznacza, że abstrakcja pierwszego węzła zostanie zaimplementowana za pomocą abstrakcji drugiego węzła. Rysunek pokazuje więc dwie procedury, P i Q oraz jedną abstrakcję danych, T. P zostanie zaimplementowane za pomocą Q (tzn. kod P wywołuje Q) oraz T (tzn. jego kod używa obiektów typu T). (Rekurencja jest zaznaczona na wykresie cyklem. Jeśli więc oczekiwalibyśmy, że implementacja P wywoła P, wskazywałaby na to strzałka od P do P).

Ilustracja reprezentuje wczesny etap projektu, w którym projektant wie jak zaimplementować P oraz wymyślił już Q i T. W tym momencie zidentyfikowano niektóre operacje T, a projektant zadecydował, że do komunikacji pomiędzy P i Q zostanie użyty obiekt typu T.

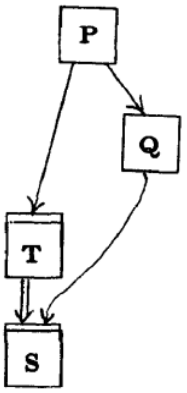
Kolejnym etapem projektowania jest zbadanie, jak zaimplementować Q. (Nie ma sensu patrzeć w tym momencie na implementację T, ponieważ nie znamy jeszcze wszystkich jego operacji).



Rysunek 4-1 Początek projektu

W trakcie badania Q jesteśmy w stanie zdefiniować dodatkowe operacje dla T. Można na to spojrzeć jak na adaptacje T tak, aby stało się podtypem S, jak pokazano na Rysunku 4-2. Podwójna strzałka wskazuje tutaj od nadtypu do podtypu; podwójne strzałki mogą łączyć tylko abstrakcje danych (i nie może być cykli obejmujących jedynie podwójne strzałki).

.



Rysunek 4-2 Ciąg dalszy projektu

Rodzaj rafinacji przedstawionej na rysunku może zdarzyć się kilka razy, np. S może mieć z kolei podtyp R i tak dalej. Ponadto, jeden typ może mieć kilka podtypów, reprezentujących potrzeby różnych podczęści programu.

Śledzenie tych różnic jako podtypów jest lepsze niż traktowanie ich jako grupy typów z kilku powodów. Po pierwsze, może ograniczyć skutki błędów projektowych. Załóżmy na przykład, że inwestygacja doprowadziła do problemu w interfejsie S. Kiedy pojawia się problem tego rodzaju,

konieczne jest aby: spojrzeć na każdą abstrakcję, która używa zmienionej abstrakcji. W przypadku rysunku oznacza to, że musimy spojrzeć na Q. Pod warunkiem, że interfejs T jest nienaruszony, nie musimy patrzeć na P. Gdyby S i T były traktowane jako jeden typ, wtedy P również musiałoby zostać sprawdzone.

Kolejną zaletą rozróżniania typów jest to, że pomaga nam w uzasadnieniu decyzji projektowych. Dokument dotyczący decyzji projektowych opisuje decyzje podjęte w poszczególnych punktach projektu i omawia, dlaczego zostały one wykonane i jakie istnieją alternatywy. Poprzez zachowanie hierarchii w podejmowaniu powstających z czasem decyzji możemy uniknąć nieporozumień i być bardziej precyzyjni. Jeśli błąd zostanie wykryty później, potrafimy dokładnie określić, w którym momencie projektu wystąpił.

Wreszcie, rozróżnienie może pomóc podczas implementacji, np. jeśli nie T ale S ma zostać reimplementowane. Może się jednak zdarzyć, że w implementacji nie zostanie zachowana hierarchia. Często koniec projektu to tylko jeden typ, ostatni wymyślony podtyp, ponieważ wdrażanie pojedynczych modułów jest wygodniejsze niż posiadanie oddzielnych modułów dla nadtypów i podtypów. Mimo to jednak rozróżnienie pozostaje przydatne po implementacji, ponieważ skutki zmian nadal można zlokalizować w specyfikacji, nawet jeśli zmiany implementacji już nie. Na przykład zmiana w specyfikacji S, a nie T oznacza, że musimy reimplementować Q, a nie P. Jeśli jednak S i T są zaimplementowane jako pojedynczy moduł musimy reimplementować oba z nich, zamiast samego S.

4.2. Pokrewne typy.

Drugie zastosowanie podtypów dotyczy pokrewnych typów. Projektant może rozpoznać, że program będzie używał kilku abstrakcji danych podobnych do siebie ale różnych. Różnice reprezentują warianty tego samego uogólnionego założenia, gdzie podtypy mogą mieć ten sam zestaw operacji, a niektóre z nich mogą poszerzać nadtyp. Przykładem jest wspomniane wcześniej uogólnione abstrakcyjne urządzenie. Aby pomieścić pokrewne typy w projektowaniu, projektant wprowadza nadtyp w momencie, gdy cały zestaw typów jest w trakcie tworzenia, a następnie wprowadza podtypy, które są potrzebne w późniejszym etapie projektowania.

Powiązane typy powstają na dwa różne sposoby. Czasami relacja jest określona z góry, przed wynalezieniem jakiegokolwiek typu; jest to sytuacja omówiona powyżej. Ewentualnie, relacja może nie zostać rozpoznana dopóki nie istnieje kilka pokrewnych typów. Dzieje się tak z powodu chęci zdefiniowania modułu, który działa na każdym z powiązanych typów, ale zależy tylko od niewielkiej wspólnej części. Na przykład, moduł może być procedurą sortowania, opierającą się na argumencie „kolekcja", by umożliwić mu pobieranie elementów oraz aby polegał na samym typie elementu, dla zapewnienia operacji „ < ".

Kiedy relacja zdefiniowana jest z góry, hierarchia jest dobrym sposobem na jej opisanie i prawdopodobnie będziemy chcieli użyć dziedziczenia jako mechanizmu implementacji. Pozwala nam to na implementacje tylko raz (w nadtypie) wszystkiego, co da się zrobić w sposób niezależny od podtypów. Moduł podtypu jest zainteresowany tylko konkretnym zachowaniem danego podtypu i jest niezależny od modułów implementujących inne podtypy. Posiadanie oddzielnych modułów dla nad- i podtypów zapewnia lepszą modularność niż użycie jednego modułu do realizacji implementacji ich wszystkich. Ponadto, jeśli później zostanie dodany nowy podtyp, żadna z istniejących implementacji nie musi być zmieniona.

Gdy relacja zostanie rozpoznana po zdefiniowaniu typów, hierarchia może nie być właściwym sposobem tworzenia programu. Ten problem poruszony został w Rozdziale 5.1.

4.3. Organizowanie Biblioteki Typów

Istnieje jeszcze inny sposób na wykorzystanie hierarchii, który ma pomóc w organizacji biblioteki typów. Od dawna wiadomo, że programowanie jest bardziej efektywne, jeśli można je wykonywać w kontekście, który zachęca do ponownego wykorzystania modułów programu zaimplementowanych przez innych. Aczkolwiek, aby dany kontekst nadawał się do użytku musi być łatwy w nawigowaniu, aby określić, czy szukane moduły istnieją. Hierarchia jest przydatna jako sposób organizacji biblioteki programów, aby ułatwić wyszukiwanie, zwłaszcza w połączeniu z narzędziami do wyszukiwania, np. w środowisku Smalltalku.

Hierarchia umożliwia grupowanie podobnych typów. Tak więc, jeśli użytkownik potrzebuje określonego rodzaju „kolekcji" abstrakcji istnieje duża szansa, że szukana abstrakcja, jeśli w ogóle istnieje, może być znaleziona za pomocą pozostałych kolekcji. Stosowana hierarchia jest albo hierarchią podtypów, albo prawie hierarchią podtypów (tzn. podtyp nieznacznie różni się od rozszerzenia nadtypu). Chodzi o to, że typy są pogrupowane na podstawie ich zachowania, a nie tego, jak są wykorzystywane do wzajemnej implementacji.

W wyszukiwaniu typów kolekcji, typów liczbowych lub jakiegokolwiek innego typu pomocne są dwie rzeczy. Po pierwsze, trzeba wziąć pod uwagę, że cała biblioteka wywodzi się z jednego źródła lub źródeł, należy również zapewnić użytkownikowi narzędzie do poruszania się w hierarchii. Po drugie, należy rozsądnie dobierać nazwy głównych kategorii, aby użytkownik mógł rozpoznać, że „kolekcja" jest częścią pożądanej hierarchii.

Używanie hierarchii jako sposobu na organizowanie biblioteki jest dobrym pomysłem ale nie trzeba ją łączyć z mechanizmem podklasy w języku programowania. Zamiast tego interaktywny system wspierający budowę i przeglądanie biblioteki pomógłby w tworzeniu biblioteki.

5. Hierarchia typów i dziedziczenie

Nie wszystkie przypadki użycia hierarchii typów wymagają obsługi przez język programowania. Wsparcie nie jest wymagane dla biblioteki programów; zamiast tego wystarczy użyć pojęcia hierarchii typów jako zasady organizacji. Jak wspomniano wcześniej, najbardziej prawdopodobnym wynikiem tej techniki projektowania jest finalny typ (ostatni wprowadzony podtyp), którym jest implementowany jako jednostka. Dlatego każdy język obsługujący abstrakcję danych jest tutaj odpowiedni, chociaż dziedziczenie może być przydatne do wprowadzania dodatkowych operacji utworzonych później.

Dla pokrewnych typów może być jednak potrzebna specjalna obsługa języka. To wsparcie omówione jest w rozdziale 5.1. Rozdział 5.2 omawia relację dziedziczenia z wieloma implementacjami typu.

5.1. Polimorfizm

Procedura polimorficzna lub abstrakcja danych to taka, która działa dla wielu różnych typów. Rozważmy na przykład procedurę sortującą. W wielu językach taka procedura została zaimplementowana by działała na tablicy liczb całkowitych; gdybyśmy chcieli później posortować tablicę ciągów znaków, potrzebna byłaby inna procedura. Jest to niefortunne. Idea sortowania jest niezależna od danego typu elementu na tablicy, pod warunkiem, że można porównać elementy w celu określenia, które z nich są mniejsze od innych. Powinniśmy być w stanie zaimplementować jedną procedurę sortowania, która działałaby dla wszystkich typów. Procedura ta byłaby polimorficzna.

Ilekroć w programie występują pokrewne typy, prawdopodobnie występuje również polimorfizm. Nawet gdy związek zdefiniowany jest z góry może wystąpić poliformizm. W takim przypadku nadtyp jest często *wirtualny*: nie ma własnych obiektów, ale jest po prostu typem zastępczym w hierarchii dla rodziny pokrewnych typów. W tym przypadku każdy moduł używający nadtypu jest polimorficzny. Z drugiej strony, jeśli nadtyp posiada własne obiekty niektóre moduły mogą używać tylko ich i żadnego z jego podtypów.

Używanie hierarchii jako wsparcie polimorfizmu oznacza, że moduł polimorficzny jest postrzegany jako używający nadtypu, a każdy typ, który ma być używany przez ten moduł, staje się podtypem nadtypu. Hierarchia jest dobrym sposobem na uchwycenie relacji, kiedy nadtypy są wprowadzane przed podtypami. Nadtyp jest dodawany do uniwersum typów wtedy, kiedy jest on tworzony a jego podtypy później.

Jeśli typy istnieją przed relacją, hierarchia również nie działa. W tym przypadku wprowadzenie nadtypu komplikuje uniwersum typów: należy dodać nowy typ (nadtyp), wszystkie typy używane przez moduł polimorficzny muszą zostać jego podrzędnymi, a klasy implementujące podtypy muszą być zmienione w celu odzwierciedlenia hierarchii (i ponownie skompilowane w systemie, który wykonuje kompilację). Na przykład, musielibyśmy dodać nowy typ „sortowalny” do uniwersum i sprawić, by każdy element typu był jego podtypem. Zauważmy, że każdy taki nadtyp musi być brany pod uwagę za każdym razem, kiedy tworzymy nowy typ: każdy nowy typ musi być podtypem nadtypu jeśli istnieje jakakolwiek szansa, że będziemy chcieli użyć jego obiektów w module polimorficznym. Ponadto supertyp może być bezużyteczny tak długo jeśli dotyczy on współdzielonego kodu, ponieważ może nie mieć nic co można byłoby zaimplementować.

Alternatywnie można po prostu pozwolić modułowi polimorficznemu na użycie dowolnego typu, który dostarcza potrzebne operacje. W tym przypadku nie podejmuje się próby powiązania typów. Zamiast tego obiekt należący do dowolnego z powiązanych typów może być przekazany jako argument modułu polimorficznego. W ten sposób otrzymujemy taki sam efekt ale bez konieczności komplikowania uniwersum typów. Podejście to będziemy nazywać *grupowaniem*.

Te dwa podejścia różnią się tym, co jest brane pod uwagę do uzasadnienia poprawności programu. W obu przypadkach wymagamy, aby obiekty argumentów miały operacje z odpowiednią sygnaturą i zachowaniem. Wymagania dotyczące sygnatury można sprawdzić podczas kontroli zgodności typów, co może mieć miejsce w czasie wykonywania lub w czasie kompilacji. Sprawdzanie podczas wykonywania nie wymaga specjalnego mechanizmu; obiekty są po prostu przekazywane do modułu polimorficznego, a błędy typu zostaną zauważone, jeśli potrzebna operacja nie jest obecna. Sprawdzanie w czasie kompilacji wymaga odpowiedniego systemu typów. Jeśli używane jest podejście hierarchiczne, wtedy język musi łączyć sprawdzanie typu w czasie kompilacji z

dziedziczeniem, przykład takiego języka omówiono w [17]. Jeśli używamy grupowania potrzebujemy sposobu na wyrażenie ograniczeń operacji w czasie kompilacji. Na przykład, oba języki CPU oraz Ada [19] są w stanie to zrobić. W CLU nagłówkiem procedury sortowania może być

sort = **proc** [T: type] (a: array[T])

where T has lt: **proctype** (T, T) **returns** (bool)

Ten nagłówek ogranicza parametr T do typu z operacją o nazwie *It* ze wskazaną sygnaturą; specyfikacja *sort* wyjaśniałaby, że lt musi być sprawdzeniem „mniej niż". Przykładem wywołania *sort* jest

sort[int](x)

Kompilując takie wywołanie, kompilator sprawdza, czy typ x to tablica[int], ponadto, że int ma operacja o nazwie lt z wymaganą sygnaturą. Moglibyśmy nawet zdefiniować bardziej polimorficzną rutynę sortowania w CLU, która działałaby dla wszystkich kolekcji, które są "indeksowalne", tzn. których elementy mogą być indeksowane jak i pobierane za pomocą indeksu.

Wymagania dotyczące zachowania muszą być sprawdzone przez jakąś formę weryfikacji programu. Wymagane zachowanie musi być częścią specyfikacji, a specyfikacja znajduje się w różnych miejscach dwóch metod. W przypadku hierarchii specyfikacja należy do nadtypu; w powiązanych typach jest częścią specyfikacji modułu polimorficznego. Sprawdzanie zachowania odbywa się również w różnym czasie. W hierarchii, sprawdzanie ma miejsce za każdym razem, gdy programista stwarza podtyp nadtypu; w grupowaniu, dzieje się tak za każdym razem, gdy programista pisze kod, który używa modułu polimorficznego.

Zarówno grupowanie, jak i hierarchia mają ograniczenia. Większa elastyczność w zastosowaniu modułu polimorficznego jest wskazana. Na przykład, jeśli *sort* zostanie wywołany z operacją, która wykonuje test „większego niż", doprowadzi to do innego sortowania tablicy, ale to inne sortowanie może być właśnie tym, czego oczekujemy. Dodatkowo, mogą wystąpić konflikty między typami przeznaczonymi do użycia w module polimorficznym:

1. Nie wszystkie typy zapewniają wymaganą operację.
2. Typy używają różnych nazw operacji.
3. Niektóre typy używają nazwy wymaganej operacji dla innej operacji, np. nazwa *It* jest używana w typie T do identyfikacji operacji „*długości*".

Jednym ze sposobów osiągnięcia większej ogólności jest po prostu przekazanie potrzebnych operacji jako argumentów procedury, np. *sort* w rzeczywistości przyjmuje dwa argumenty, tablicę i procedurę używaną do określenia kolejności[4]. Ta metoda jest ogólna, ale może być niewygodna. Metody, które pozwalają uniknąć niedogodności w połączeniu z podejściem grupowania istnieją w takich językach jak Argus [10] i Ada.

Podsumowując, gdy powiązane typy zostaną odkryte na wczesnym etapie projektowania, hierarchia jest dobrym sposobem na wyrażenie związku. W przeciwnym razie albo podejście

[4] Oczywiście to rozwiązanie nie sprawdziłoby się dobrze w Smalltalku, ponieważ procedury nie mogą być wygodnie definiowane jako pojedyncze podmioty ani traktowane jako obiekty.

grupowania (z odpowiednim wsparciem językowym) albo procedury takie jak argumenty mogą być lepsze.

5.2. Wielokrotne implementacje

Często przydatne jest posiadanie wielu implementacji tego samego typu. Na przykład dla niektórych macierzy używamy rzadkich reprezentacji, a dla innych reprezentacji nierzadkich. Co więcej, czasami pożądane jest używanie obiektów tego samego typu, ale różnych reprezentacji w tym samym programie.

Wydaje się, że języki obiektowe umożliwiają użytkownikom symulowanie wielu implementacji przez dziedziczenie. Każda implementacja byłaby podklasą innej klasy, która implementuje typ. Ta druga klasa byłby prawdopodobnie wirtualna; na przykład istniałaby wirtualna klasa implementująca macierze i podklasy implementujące rzadkie i nierzadkie macierze.

Korzystanie z dziedziczenia w ten sposób pozwala nam na kilka implementacji tego samego typu w tym samym programie, ale koliduje to z hierarchią typów. Załóżmy na przykład, że wymyślimy podtyp macierzy, macierz rozszerzoną. Naszym celem jest implementacja rozszerzonych macierzy z klasą, która dziedziczy z macierzy, a nie z konkretnej implementacji macierzy, ponieważ pozwoliłoby to nam na połączenie z dowolną implementacją macierzy. Nie jest to jednak możliwe. Zamiast tego klasa rozszerzonej macierzy musi być wyraźnie wskazana w tekście programu jako podklasa rzadkiej lub nie rzadkiej macierzy.

Problem pojawia się, ponieważ dziedziczenie jest używane do dwóch różnych rzeczy: do implementacji typu i wskazania, że jeden typ jest podtypem innego. Zastosowania te powinny się łączyć. Dopiero wtedy otrzymalibyśmy to co chcemy: dwa typy (macierz i rozszerzoną macierz), jeden podtyp drugiego, gdzie każdy posiada kilka implementacji oraz możliwość łączenia implementacji podtypu z implementacjami nadtypu na różne sposoby.

6. Podsumowanie

Abstrakcja, a zwłaszcza abstrakcja danych, jest ważną techniką do tworzenia programów, które są stosunkowo łatwe w utrzymaniu i modyfikowaniu w miarę zmieniających się wymagań. Abstrakcje danych są szczególnie ważne, ponieważ ukrywają skomplikowane rzeczy (struktury danych), które prawdopodobnie ulegną zmianie w przyszłości. Pozwalają nam na lokalną zmianę reprezentacji danych bez wpływu na programy korzystające z tych danych.

Dziedziczenie to mechanizm implementacji, który umożliwia powiązanie jednego typu z innym hierarchicznie. Jest używany na dwa sposoby: do implementacji typu pochodzącego z implementacji innego typu i zdefiniowania podtypów. Stwierdziliśmy, że pierwsze użycie jest nieciekawe, ponieważ możemy osiągnąć ten sam wynik, używając jednego typu jako reprezentacji drugiego. Z drugiej jednak strony podtypy stwarzają nowe możliwości. Zidentyfikowano trzy zastosowania podtypów. Podczas projektowania przyrostowego pozwalają na ograniczenie wpływu zmian na projekt i lepsze uporządkowanie dokumentacji. Zapewniają również sposób na grupowanie powiązanych typów, szczególnie gdy nadtyp został wymyślony przed jakimikolwiek podtypami. Kiedy związek zauważony jest po zdefiniowaniu kilku typów inne metody, takie jak grupowanie lub argumenty procedury są

prawdopodobnie lepsze niż hierarchia. Wreszcie, hierarchia jest wygodnym i sensownym sposobem organizacji biblioteki typów. Hierarchia jest albo hierarchią podtypów, albo prawie nią jest; podtypy mogą nie pasować do naszej ścisłej definicji, ale są podobne do nadtypu w pewnym intuicyjnym sensie.

Dziedziczenie może służyć do implementacji hierarchii podtypów. Jest to potrzebne przede wszystkim w przypadku powiązanych typów, gdzie nadtyp wymyślany jest jako pierwszy, ponieważ wygodniej jest zaimplementować wspólne cechy tylko raz w nadklasie, a następnie zaimplementować rozszerzenia osobno dla każdego podtypu.

Dochodzimy do wniosku, że chociaż abstrakcja danych jest ważniejsza, hierarchia typów rozszerza jej użyteczność. Ponadto dziedziczenie jest czasami potrzebne do wyrażenia hierarchii typów i dlatego jest przydatnym mechanizmem do wprowadzenia w języku programowania.

Podziękowania
Wiele osób zgłosiło uwagi i sugestie, które poprawiły treść tego dokumentu. Autor z wdzięcznością docenia tę pomoc, a zwłaszcza wysiłki Toby'ego Blooma i Gary'ego Leavensa.

Bibliografia

1. Bobrow, D., et al. “CommonLoops: Merging Lisp and Object-Oriented Programming”. Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications, SIGPLAN Notices 21, 11 (November 1986).

2. Bruce, K., and Wegner, P. “An Algebraic Model of Subtypes in Object-Oriented Languages (Draft)“. SIGPLAN Notices 21, 10 (October 1986).

3. Dahl, O.-J., and Hoare, C. A. R. Hierarchical Program Structures. In Structured Programming, Academic Press, 1972.

4. Goldberg, A., and Robson, D. Smalltalk-80: The Language and its Implementation. Addison- Wesley, Reading, Ma., 1983.

5. Hoare, C. A. R. “Proof of correctness of data representations'”. Acta In Informatica 4 (1972), 271-281.

6. Leavens, G. Subtyping and Generic Invocation: Semantics and Language Domain Design. Ph.D., Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, forthcoming.

7. Liskov, B. A Design Methodology for Reliable Software Systems. In Tutorial on Software Design Techniques, P. Freeman and A. Wasserman, Eds., 1977. Also published in the Proc. of the Fall Joint Computer Conference, 1972.

8. Liskov, B., Snyder, A., Atkinson, R. R., and Schaffert, J. C. “Abstraction mechanisms in CLU* . Comm. of the ACM 20, 8 (August 1977), 564-576.

9. Liskov, B., et al.. CLU Reference Manual. Springer-Verlag, 1984.

10. Liskov, B., et al. Argus Reference Manual. Technical Report MIT/LCS/TR-400, M.I.T. Laboratory for Computer Science, Cambridge, Ma., 1987.

11. Liskov, B., and Guttag, J.. Abstraction and Specification in Program Development. MIT Press and McGraw Hill, 1986.

12. Liskov, B., and Zilles, S. “Programming with abstract data types”. Proc. of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices 9 (1974).

13. Moon, D. “Object-Oriented Programming with Flavors”. Rot. of the ACM Con jerance on Object- Oriented Programming Systems, Languages, and Applications, SIGPLAN Notices 21, 11 (November 1986).

14. Morris, J. H. “Protection in Programming Languages”. Comm. of the ACM 16, 1 (January 1973).

16. Parnas, D. Information Distribution Aspects of Design Methodology. In Fkoceedinge oj IFIP Congress, North Holland Publishing Co., 1971.

16. Parnas, D. “On the Criteria to be Used in Decomposing*Systems into Modules". Comm. of the ACM 15, 12 (December 1972).

17. Schaffert, C., et al. “An Introduction to Trellis/Owl”. Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, SIGPLAN Notices 21, 11 (November 1986).

18. Snyder, A. “Encapsulation and Inheritance in Object-Oriented Programming Languages". Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, SIGPLAN Notices 21, 11 (November 1986).

19. U. S. Department of Defense. Reference manual for the Ada programming language. 1983. ANSI standard Ada.